Mapping of Modal Applications given Throughput and Latency Constraints

Stefan J. Geuns[§], Joost P.H.M. Hausmans[§], Marco J.G. Bekooij[§¶]
[§]University of Twente, The Netherlands, [¶]NXP Semiconductors, The Netherlands
stefan.geuns@utwente.nl, joost.hausmans@utwente.nl, marco.bekooij@nxp.com

***Abstract*—Real-time applications such as software defined radios have different reception modes and their real-time requirements are a result of periodic sources and sinks in the form of ADCs and DACs. Tools are under development that automatically translate a sequential specification of a radio application, that often includes nested while loops to describe the modes, into a parallel task graph and map this task graph onto an embedded multiprocessor system. However the specification of strict periodic sources and sinks together with input and output buffers that can respectively overflow or underrun is currently not possible in a sequential programming language.**

In this paper we will introduce a nested loop program (NLP) language extension that enables the specification of periodic sources and sinks and their buffers in a sequential program. We show that parallelization of such a sequential program poses challenges because the order in which different tasks access the input and output buffers should be maintained in the parallel program. Furthermore, the buffers at the sources and sinks allow destructive writes and non-destructive reads, which causes non-deterministic functional behavior in case the throughput and latency constraint of the application are not met. The other buffers in the task graph block in case no data or space is available. Therefore, the system internals remain functionally deterministic which significantly simplifies debugging and analysis.

Furthermore, to guarantee real-time requirements, we show that it is possible to conservatively model an application with nested while loops as a Cyclo-Static Dataflow (CSDF) model. Using this model we can compute a mapping of the task graph, which includes a task to processor assignment, suitable scheduler settings and buffer capacities. By making use of this CSDF model, we can guarantee that sources and sinks can run periodically under the assumption that the used execution times of the tasks are upper bounds.

I. Introduction

Today's embedded systems often execute a number of stream processing applications simultaneously. Each applications samples its input streams periodically and produces, after processing, a periodic output stream. For example Software Defined Radio (SDR) systems can execute simultaneously a Digital Video Broadcasting (DVB) application and a Global Positioning System (GPS) application. Furthermore, for performance reasons, these applications are often executed on a multi-core platform.

However, programming of a multi-core platform introduces a number of issues. Communication is performed between tasks on different processing cores, which introduces the need for synchronization between the tasks. Streaming applications often also have real-time requirements in terms of throughput and latency constraints.

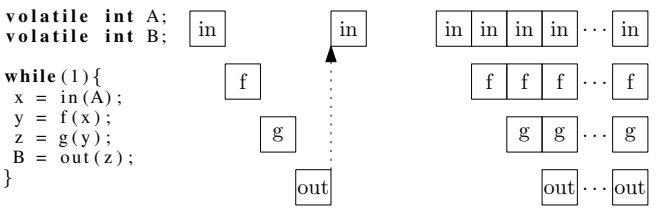
Communication between the environment and the application is done via ports. We distinguish two port types, ports which are only read and ports which are only written. Ports which are read are called sources and ports which are written are called sinks. These sources and sinks are often shared between different tasks in the system. In a parallel approach, the order in which they are read or written must be the same as in the sequential specification. For example in Figure 3a multiple functions read from the source A. If they read their input data in a different order, systems behavior could clearly be affected.

Ports can also be distinguished in terms of their timing behavior. Ports such as an analogue-digital convertor (ADC) operate periodically. Ports such as keyboards operate aperiodically, often event based. This paper focuses on periodic ports because these are the type of ports that are typically used in real-time stream processing systems.

The equivalent for a port in a traditional sequential language, such as C, is to declare the corresponding variable as volatile. However, due to the communication with peripherals, volatiles have side-effects and should therefore be executed in the order as specified in the sequential program [11]. This requirement precludes pipeline parallelism over loop iterations, which is illustrated with the example in Figure 1. Before the in function in the loop in Figure 1a can be executed a second time, the out function must be executed first. The result is the schedule shown in Figure 1b, where the dotted arrow between out and in shows the constraint imposed by the sequential ordering. The schedule from Figure 1c shows the desired pipelined execution which would result from a more relaxed language requirement than volatiles.

Therefore, a language extension must be defined to our nested loop programs (NLPs) language, which already allows while loops to support modes, for interfacing with the environment. The language extension must allow for pipelining and it should be possible to derive a corresponding analysis model from an NLP.

This paper introduces a parallelization and analysis method in which a sequentially described NLP that contains periodic sources and sinks, is parallelized into a task graph for which pipelined execution is possible. It is shown that this task graph can be analyzed using a corresponding Cyclo-Static Dataflow (CSDF) model [6]. The CSDF model allows for buffer capacity calculations using a given throughput constraint, which is imposed by the periodic sources and sinks. Also a given



(a) Input program (b) Schedule as produced by the C semantics. (c) Schedule as produced by our approach.

Fig. 1. Pipelining is not allowed under the C semantics, while our approach does allow for pipelining.

latency constraint between sources and sinks can be verified using this method.

If execution times are optimistic, it cannot be guaranteed that the throughput constraint will be met. Therefore, non-determinism is allowed at the buffers connected to the sources and sinks by letting them respectively overflow or underrun. A specific Circular Buffer (CB) type is introduced that supports this. This CB also support the sequential access ordering to be preserved after parallelization.

The CBs can be first-in first-out (FIFO) buffers or CBs with sliding windows [4] or overlapping windows [5]. To preserve functional correctness, synchronization statements are inserted into the generated parallel task graph using the method described in [13]. Using the placement of these synchronization statements, a corresponding CSDF model is generated. The CSDF model is used to determine, for given throughput and latency constraints, buffer capacities for the buffers between the tasks in the task graph.

The remainder of this paper is organized as follows. Section II discusses related work. Section III provides a justification of our system design approach which uses either time-triggered periodic or data-driven aperiodic sources and sinks on the system boundary in combination with data-driven execution of tasks. Section IV details the language extension for the sources and sinks. Section IV-A describes the parallelization of an application with sources and sinks and Section IV-B describes the new buffer type. Section V describes the derivation of the corresponding CSDF analysis model. The applicability is illustrated using a WLANp receiver in Section VI. The conclusion is stated in Section VII.

II. Related Work

Existing languages that allow for the specification of periodic ports are for example the synchronous languages, such as Esterel [1] and Lustre [10]. Both languages contain constructs that allow for a periodic execution of statements. These synchronous languages are based on an abstraction in which all events occur simultaneously. An important difference with our approach is that the input specification of synchronous programs is a concurrent specification whereas we have a sequential input specification. The synchronous programs are generally compiled to sequential code, although there are a couple of attempts to support distributed systems [2], [9].

In ADA [16] time is specified using a delay statement. The delay statement blocks the execution of the current task until a specified absolute or relative deadline. Similar to the synchronous languages, ADA also starts from a parallel description of the program while we start from a sequential specification. As a consequence, compilers for ADA can not always detect the presence or absence of deadlock in the parallel program. Our sequential specification is deadlock free by definition and this property is preserved after parallelization. Guaranteed deadlock freedom is for us one of the most important arguments to start from a sequential input specification.

Similar to ADA, the RTC++ [14] approach allows for a specification of time in the language. The RTC++ approach is an extension to C++ in which time is added to the language. RTC++ contains an exception construction which allows for the specification of, amongst others, timeouts. The disadvantage is that RTC++ allows the introduction of non-determinism inside the language. Our approach only allows for non-determinism at the border of the system and the behavior is only non-deterministic in case execution times are underestimated. An important advantage is that an NLP internally has a deterministic behavior which simplifies debugging and analysis significantly.

An approach which also starts from a sequential program and converts this to a parallel system is PN [18]. The disadvantage of this approach is that nested while loops are not allowed in the input specification, whereas these while loops are needed to describe modes in stream processing applications.

III. System Overview

A system can execute using a number of different execution styles. When a system reacts to events it receives from the environment it is called event-triggered. A disadvantage is that event-flooding can occur because often no minimum distance between events is known at design time. As a consequence, minimum buffer sizes can not be determined and therefore it can occur that events have to be discarded. Also a pure event-triggered system can not detect the absence of events.

If a system executes tasks according to a predetermined schedule, the system is called time-triggered. The main disadvantage of such systems is that it requires that the worst-case execution time of the tasks are known, otherwise outdated data might be read by a task inside the application which may result in an undefined functional behavior of the application. However, in many systems data caches are applied which usually result in a significant overestimation of the worst-case execution times. As a consequence, designers of non-safety critical systems typically resort to measurement of the execution times instead and derive from these measured execution times upper bounds of the execution times of the tasks. However these upper bounds might be lower than the worst-case execution times of the tasks.

A system can also execute its tasks data-driven. This means that tasks are only executed when data is available at their

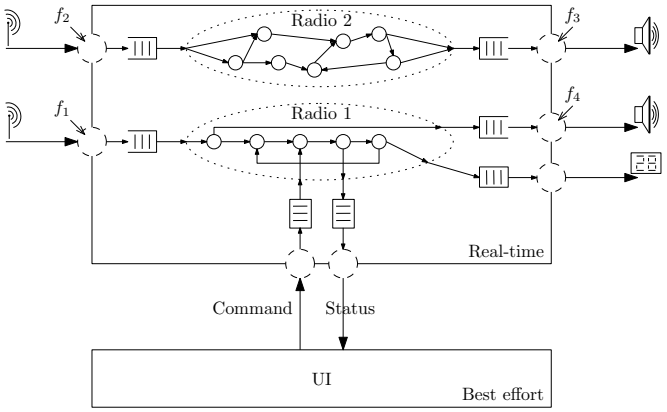


Fig. 2. System overview.

input buffers and space is available at the output buffers to write the results to. A data-driven approach can cope with variable execution times since tasks execute when there is data available at their input buffers. If the execution time of one task's execution is too long and another execution is shorter, the total time can still be low enough to meet the deadlines. The disadvantage is that the tasks can not execute strictly periodic without an external periodic trigger. Also the absence of events can not be detected because nothing is executed without data.

In our approach a hybrid of the time-triggered and data-driven approaches is applied to make the system more robust against potentially underestimated execution times and event flooding, but it allows the use of periodic sources and sinks. The processing in our approach is done data-driven and also the aperiodic sources and sinks execute data-driven whereas periodic sources and sinks execute time-triggered. As a consequence, external signals are always sampled and therefore even the absence of events can be detected.

Figure 2 shows an overview of an example system that adheres to our approach. The two radio applications execute data-driven and independently of each other. They read their data from buffers connected to the two periodic sources, on the left in the figure, and write the results to buffers connected to the periodic and aperiodic sinks, on the right in the figure.

The buffers connected to the sources and sinks are blocking for the internal tasks but periodic sources can always write to these buffers and periodic sinks can always read from them. The aperiodic sources can only write data if there is space available in the buffers and the sinks can only read data if there is data available in the buffers.

Commands are received from the best-effort part of the system, this includes for example a User-Interface (UI). The ports between the real-time system and the best-effort system are executed data-driven. As a consequence these ports execute aperiodically. The aperiodic execution is determined by the real-time system such that no system overload can occur in the real-time part as a consequence of too many external events in a time-interval from the best-effort part of the system. These data-driven sources sample when there is empty buffer space available and the data-driven sinks write data when there is enough data available in their input buffer.

The data-driven output ports can also be used in systems that produce burst of output data but need to sample their inputs continuously. An example of such a system is the WLANp receiver application that is described in Section VI.

A. Non-Determinism

An application is functionally deterministic if each execution of the application using the same input produces the same output. A program is functionally non-deterministic if different outputs are produced given the same inputs.

Our approach follows the same ideas on non-determinism as described in [7]. The sequential specification is completely deterministic and after parallelization the application remains functionally deterministic by construction. The only allowed non-determinism is at the boundary of the system and is specified explicitly by the use of sources and sinks. Requiring determinism in the internal tasks of the system simplifies the analysis of the real-time requirements and also simplifies debugging.

Non-determinism is caused by the buffers isolating the time-triggered tasks from the data-driven tasks. In case of system overload these buffers can overflow, causing destructive writes, or underrun, requiring non-destructive reads. This behavior causes non-determinism as it is dependent on the execution times which values are written or read from these buffers.

Non-determinism is also a natural result of the sampling of periodic ports [8]. The logical time of the parallel task graph is discrete while the actual time of the environment is continuous. Infinitely small differences in sampling moments can cause different output results.

The buffers between the sources and sinks and the system internals cause non-determinism, but also isolate non-determinism from the system internals. As soon as the system internals start processing a data item, i.e. the data item is read from the source buffer, the outcome is completely deterministically determined by the application and the results are written into the sink buffer.

In contrast to the differentiation made in [7], our approach is a combination of both a language based approach, a compiled-based approach and a software run-time approach. A language extension is required to explicitly introduce non-determinism in the program while compiler support is required to guarantee determinism in the derived parallel task graph. The software run-time environment in turn ensures that the generated synchronization primitives actually adhere to the memory-consistency model that is supported by the hardware.

B. Memory Consistency

A memory consistency model defines the order in which updates of variables become visible. The memory consistency model can be seen as the contract between the programmer, the compiler and the hardware, from which the programmer can derive the functional behavior of a program. Because memory consistency models define the order in which variable updates

become visible, they also define the reordering freedom of load and store operations. Therefore, they can have a significant impact on the available parallelism in an application. An example of a memory consistency model that offers little reordering freedom but which is relatively easy to use by the programmer is sequential consistency [15]. Sequential consistency is supported if the order in which stores become visible is an interleaving order that might occur if all loads and stores complete in the order as specified by the sequential program. As a consequence, sequential consistency offers minimal reordering freedom which usually limits the available parallelism severely.

For our sequential NLPs we rely on a simple memory consistency model which defines that a variable must be written before *the same* variable is read. Because there are no ordering constraints imposed by this memory consistency model between different variables there is a significant reordering freedom of loads and stores. There is also a constraint on the life-time of a value written in a variable by means of a Single Assignment Section (SAS) [13]. Basically a SAS states that a value stored in a variable is lost after every execution of the while loop iteration in which the value is written.

Our parallelization approach can exploit that there is no ordering constraint between writes and reads of different variables in the NLP, assuming the memory consistency model used in the parallel task graph also supports this. A memory consistency model that supports this reordering freedom is streaming consistency [17]. Streaming consistency requires that all shared variables are encapsulated by acquire and release statements. These acquire and release statement guarantee mutual exclusive access of the shared variables in the critical section. Given streaming consistency, only the acquire and release statements of a specific CB, which is related to one particular variable in the NLP, are not allowed to be reordered but acquire and release statements related to different variables can be reordered.

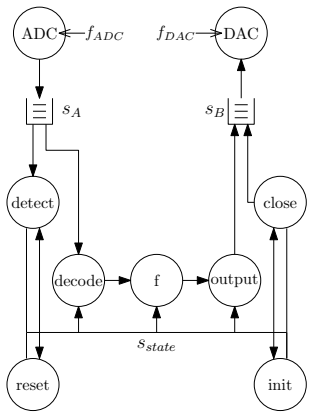
The parallelization approach from [4] encapsulates accesses of shared variables with acquire and release statements and also the extension with sources and sinks described in this paper does not violate the encapsulation requirement. A differentiation is made between acquire and release statements for tasks writing a variable and tasks reading a variable. A buffer location is acquired for write access using the $acqWW$ function and released using the $relWW$ function. Acquiring a location for read access is done via $acqRW$ and releasing it is done via $relRW$. All functions take the buffer as a parameter. The functions $acqRW$ and $relWW$ also take the array index as an additional argument.

Because each source and sink has a different variable associated to it, statements that access the source or sink are not related unless they access the same source or sink. Therefore, they are allowed to be reordered according to the streaming memory consistency model. Now assuming that the example code from Figure 1a was implemented using our NLP language, thus declaring A as a source and B as a sink, the optimal schedule from Figure 1c is a valid schedule after parallelization.

```
int source A = ADC() @ 5 KHz;
int sink B = DAC() @ 5 KHz;
start B 4 ms after A;

init(out state);
loop{
 switch(state){
 case 0:
  detect(A, out state');
 case 1:
  loop{
   x = decode(A);
   y = f(x);
   output(y, out B);
  } while(...);
  reset(out state');
 case 2:
  close(out B, out state');
 }
} while(1);
```

(a) Sequential NLP



(b) Parallel task graph

Fig. 3. Parallelization of an NLP with a source A and a sink B.

IV. Sources and Sinks

Ports provide the means for the system to communicate with the environment. There are two types of ports, ports which are read, called sources and ports which are written, called sinks. These sources and sinks are specified in the NLP using a name and a function. Functions in our NLPs can be implemented in another language, for instance C. The function corresponding to a port defines how a value is retrieved from or written to the corresponding port. The name of the source or sink can be used by statements in the NLP to read or write to that port. Periodic sources and sinks also have an additional property, their execution frequency. Aperiodic ports do not have a frequency. Ports in the system do not have a relation with each other, unless specified otherwise by the data dependencies. Therefore, the execution of the ports can be reordered with respect to each other.

Besides the periodic sources and sinks all functions in the system, including the function corresponding to the aperiodic sources and sinks, are executed data-driven, meaning they execute when there is data available on their inputs and space available on their outputs. The time triggering of the ports is performed by a timer. The period is the inverse of the specified frequency at which the ports operates. An example NLP with a port for which the frequency is specified is shown in Figure 3a. In this example there is a source and a sink, which both operate at a frequency of 5 KHz.

Sources and sinks often have latency constraints with respect to one another. For example after a periodic sink starts, there must always be data available at its input buffer. If not, noticeable glitches can occur. A minimal latency between the source and the sink can prevent this.

Latency constraints can be specified by means of a minimum time $d \in \mathbb{R}$ that one port should start after another. Note that also a negative delay can be specified to indicate that the sink should start before the source. In the example it is specified that the n-th execution of the sink should start 4 ms after the n-th execution of the source.

```
int source A = f();
int sink B = g();

while(1){
  if(c == 0)
    B = h(A);
  else
    B = k(A);
}
```

(a) Valid NLP.

```
int source A = f();
int sink B = g();

while(1){
  if(c == 0)
    B = h(A);
}
```

(b) Invalid NLP.

```
int source A = f();
int sink B = g();

while(1){
  if(c == 0)
    B = h(A);
  if(c != 0)
    B = k(A);
}
```

(c) Invalid NLP.

Fig. 4. An example where it can be verified if the communication rate equals the synchronization rate (a) and two examples where it can not (b), (c).

In the case that statements reading from a source and statements writing to a sink are in the same while loop, source and sink rate inconsistencies can be detected automatically at compile-time. For example the $decode$ function reads equally often from source A as the $output$ function writes to sink B. Therefore, the corresponding source and sink should operate at the same frequency.

If a port is accessed inside an if or switch statement, it can often not be verified at design time how often data is actually written to a sink or read from a source. In this case only the synchronization rates of the tasks are verified with the CSDF model but not the communication rates. In order to automatically verify that the communication rate is equal to the synchronization rate, an extra constraint on the sequential input program must be added. This extra constraint requires that if a source is read or a sink is written conditionally, the source or sink must be read or written in every conditional branch of that conditional statement.

Consider the example in Figure 4 which shows three code examples of NLPs, all having one source and one sink. The first example in Figure 4a reads the source and writes to the sink in all branches of the if-statement whereas the example in Figure 4b only accesses the source and sink in the if the variable c equals 0. The last example from Figure 4c accesses all ports in all branches. However, in general it can not be detected if two conditions are always equal or not to each other and therefore we always reject this construct.

A. Parallelization

From an NLP a parallel task graph is extracted. The resulting task graph allows task-level parallelism to be exploited during execution. The extracted parallel task graph should preserve the read/write ordering of ports as dictated by the order of statements in the sequential input specification. For instance if there are any side-effects on the sources or sinks, not preserving this sequential ordering results in a different functional behavior.

An NLP including sources and sinks is parallelized using the following method. The parallelism from the program body is extracted and modeled using the techniques presented in [5], [13]. A task is extracted from each statement in the program body. A CB is created for every variable in the NLP. The values from variables needed for the execution of a task is stored and retrieved from these CBs. Synchronization statements are inserted to ensure that a read values are available in the CBs and space is available in the CB to store values produced by the tasks.

For each specified source and sink a task is extracted which is not part of the endless loop body of the NLP. For a source, this task reads data from the source and writes the read data into a buffer. For a sink this scheme is reversed, data is read from a buffer and written to a sink. The buffers are of a special type and are described in more detail in the next section.

The tasks extracted from the source and sink functions ensure that the sequential read/write order is preserved after parallelization. Because there is only a single task reading from a source or writing to a sink the sequential ordering is trivially preserved. The CBs in combination with the method for the insertion of synchronization also results in a parallel task graph that is functionally equivalent to the sequential specification, as is shown in [13].

An example of the parallelization approach can be found in Figure 3. The input NLP contains a periodic source A and a periodic sink B with corresponding functions ADC and DAC. The extracted task graph is shown in Figure 3b. Our tool can automatically extract and generate a task graph from a sequential NLP.

B. Buffers

The buffers between the periodic sources and sinks and the data-driven tasks are based on the CBs as introduced in [5]. A CB from [5] contains windows in which a task can access a buffer location. Windows are used to support out-of-order access, skipping locations and reading locations multiple times. However in our approach, at the side of the sources and sinks only non-blocking FIFO access is allowed. FIFO access can be implemented using a window of size one location.

For robustness against overload, a protocol must be defined which defines how the buffers react during system overload situations. For buffers connected to sources, we see three options. The first option is to overwrite the oldest data. The disadvantage is that all read windows must be shifted to ensure that the newly written data is read after all older data is read. This causes multiple tasks updating the same windows, therefore needing atomic operations, which are not always available in embedded multi-core systems. A second option is to overwrite the newest data. A simple implementation using this scheme is keeping the write window at same place. A last alternative, closely related to the previous alternative, is to not write the currently sampled data, instead of overwriting the previously written data. In our implementation we selected the second option because this was the easiest option to implement, however the most suitable option is application and hardware dependent.

For the buffers connected to the sink, we see two alternatives. Either read the last read data again or use a default value. For both alternatives is a simple implementation, but also here the most suitable alternative is application dependent.

The usage of these protocols can introduce functional non-determinism. However, because these buffers are always at the

boundary of the tasks graph, the internal task graph remains deterministic, which simplifies algorithm specification and debugging significantly. Furthermore, applications must often be made robust against potentially corrupt input data anyway. An example is a radio channel decoder that must be able to handle errors caused by distortion in the wireless channel.

In the task graph from Figure 3b these special buffers are depicted by a open end on the side of the external ports, indicating non-blocking access, and with a closed end on the system side, indicating blocking access.

V. Modeling an NLP as a CSDF

After parallelism is extracted and synchronization statements are inserted, a CSDF model is created to determine buffer capacities and to verify whether the real-time requirements can be satisfied.

Data flow models, such as CSDF, can not model the time-triggered execution of the periodic sources and sinks because actors in a data flow model execute when tokens are available at their input edges. Tokens can also not be generated periodically and therefore no strictly periodic execution is possible. In our approach the periodic tasks are modeled as if they execute data-driven. After analysis it is guaranteed that, if a data driven execution is possible for a given throughput, a time-triggered execution is also possible.

The CSDF model is based on the placement of the synchronization statements that are inserted by the code generator of our parallelization tool. The placement of the synchronization statements is such that the synchronization is unconditional, i.e. whether a synchronization statement is executed, is not dependent on input data. The only exception are the while loops, for which we will show that particular synchronization between task can be modeled as if it is unconditional because other synchronization statements already enforces the same synchronization constraints.

A while loop in our NLPs has an unknown iteration upper bound by definition. A consequence is that in general it is impossible to derive how much time is spent inside the while loop. To guarantee that a periodic source is read in time and a periodic sink receives its data in time, synchronization must be added inside each while loop. In other words, a periodic port must be accessed by at least one statement inside each while loop.

Every while loop is modeled in the CSDF model as if it always executes one iteration. For the synchronization rate it is irrelevant which task actually accesses the port because synchronization and communication are decoupled. A sketch of the proof why modeling the execution of one iteration is sufficient can be found in the next section.

It is always possible to derive a CSDF model using the method described above if an NLP contains only scalar variables or arrays with manifest array access patterns [3]. An access pattern is manifest if it is possible to derive, at compile time, the trace of indices for arrays that will occur at run-time. An access pattern is non-manifest if such a trace can not be derived. Because we assume that functions are deterministic,

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

(a) NLP

```
do{
 acqWW(y);
  write(y,0,k());
 relWW(y,0);
} while(1);
```

(b) Task 1

```
do{
 acqWW(x);
  write(x,0,f());
 relWW(x,0);
 do{
  acqRW(b,0);
   t = read(b,0);
  relRW(b);
  acqWW(x);
 } while(t);
} while(1);
```

(c) Task 2

```
do{
 acqWW(x);
 acqRW(y,0);
 do{
  acqWW(x);
   write(x,0,
    g(read(y)));
  relWW(x,0);
  acqRW(b,0);
   t = read(b,0);
  relRW(b);
 } while(t);
 relRW(y);
} while(1);
```

(d) Task 3

```
int t;
do{
 do{
  acqRW(x,0);
   t = h(read(x,0));
  relRW(x);
  acqWW(b);
   write(b,0,t);
  relWW(b,0);
 } while(t);
 relRW(x);
} while(1);
```

(e) Task 4

Fig. 5. Example NLP in (a) and the tasks from the corresponding parallelized task graph in (b)-(e).

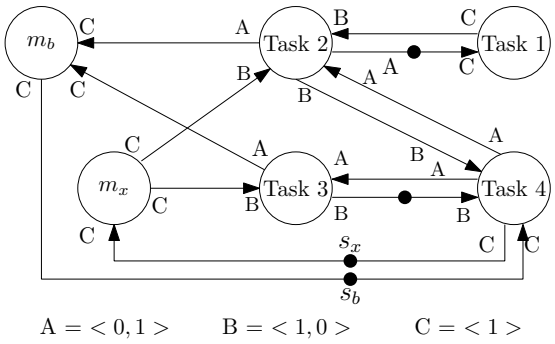


Fig. 6. CSDF model for the NLP in Figure 5a.

non-manifest behavior can only be a result of the data values produced by the sources. In the case that an access pattern is non-manifest, the CSDF model must be made conservatively and may cause the model to deadlock whereas the task graph will not deadlock, as shown in [13].

Figure 5a contains an example NLP where the variable x is written both in the statements before and inside a nested while loop. The variable y is only written in the statements before the inner while loop. After parallelization, the generated code from Figure 5b-5e is obtained. The CB b is introduced to store the result of the evaluation of the condition from the inner while loop. The variable t is an internal variable to temporarily store values from a CB. As can be seen in the figure, also Task 2 must move its window for s_x, despite that this task does not write to x in the inner while loop. This is needed because all windows must be moved an equal number of times [13], otherwise one window might prevent that other windows can move. The variable y is only read by the statements in the inner while loop, in the example the only statement in the inner while loop is the function g. Therefore, the synchronization is added outside of the inner while loop, as is shown in Task 3.

Based on the inserted synchronization, the CSDF model from Figure 6 is generated. Each task in the task graph is modeled as an actor in the CSDF model. An edge is added from each task which writes a variable to each task which reads from the same variable. Because the variable x is written by different tasks, an extra actor is required, the so called merging actor. The merging actor for variable x is named m_x in the figure. This merging actor is required to ensure that windows do not overtake each other by more than one iteration. An edge is added from each merging actor to all tasks that write to the corresponding variable.

If there are multiple tasks that read from the same variable, a merging actor is also required for the same reasons as on the writing side. Here an edge is added from all tasks which read from a variable to the corresponding merging actor. In the example there is only one task that reads from x so there is no need for a merging actor. For the variable b the situation is reversed as opposed to x, a merging actor for the reading tasks is required, named m_b, but not for the writing tasks because there is only a single task which writes to b.

Because all while loops are always executed and the tasks synchronize unconditionally, we have that each task synchronizes at least once for every variable in every while loop body. The CSDF model is created such that only one execution of a while loop is modeled. This can be seen in Figure 6 where all tasks synchronize equally often even though there is a rate difference between the variables x and y.

A. Correctness Sketch

The throughput and latency requirements are specified for each source and sink. Therefore, the schedule of the tasks internal to the system is irrelevant as long as we can determine with the CSDF model that data arrives in time and space becomes available in time [12].

The base case for throughput concerns is that each task that corresponds with a function in a while loop, executes only once. This case is modeled in the CSDF model, see for example Figure 6. It must now be shown that if a while loop executes more than one iteration, and therefore also the corresponding tasks execute more often, that than still space becomes available in time for the sources and data becomes available in time for the sinks. This requires us to provide an argument why space and data do not become available later in case the while loops execute more than once.

As explained above, every periodic source sink must be read and every periodic sink must be written in every while loop. This forces the synchronization for each source or sink variable to be inside this while loop. The tasks that correspond with the statements in the while loop potentially communicate with tasks that correspond with statements outside the while loop. If the while loop is executed more than once, the tasks that correspond with statements inside a while loop synchronize only the first iteration or the last iteration with tasks that correspond with functions outside the while loop. That tasks need to synchronize less corresponds in an unfolded data flow model with less dependencies between actors. Less dependencies can only result in an earlier firing of actors and an earlier production of tokens. Because the data flow abstraction is a conservative representation of the task graph executed on our multiprocessor system [20], we can conclude that data will be produced earlier and space will become available earlier than in the case of a single while loop iteration.

```
int source ADC = readADC() @ 250 KHz;
packet_t sink packet = writePacket();

init(out state);
loop{
  switch(state){
    case 0:{
      get11aModeParam(ADC, out nSym', out state');
    }
    case 1:{
      loop{
        getFEsamples(ADC, out yData);
        hAfcFft(yData, out freqData);
        ofdmDeMap(freqData, out symData, out symPilot);
        phaseTrackComp(symData, symPilot, out symTracked);
        qamSoftDeMapper(symTracked, out packet);
        i' = i + 1;
      } while(i <= nSym);
      reset(out state');
    }
  }
} while(1);
```

Fig. 7. NLP of a WLANp receiver.

Using the method as described in [19], sufficient buffer capacities for a given throughput constraint can be efficiently calculated given the generated CSDF model.

B. Latency

Periodic sinks should be started some time after the periodic sources because data is only available at the sink after some processing time. The analysis method as presented in [19] can take these latency constraints into account by adding additional constraints to the given linear program (LP) formulation. The LP formulation uses the start times of tasks to determine buffer capacities for a given throughput. The latency constraint dictates a minimum difference in start time between the source and the sink.

Therefore, the LP is extended as follows. If there is a latency constraint between the tasks t_A and t_B of L time units, than the constraint can be formulated in terms of start time as:

$$s_B - s_A \geq L$$

Here the variable s_A denotes that start time of task t_A and s_B denotes the start time of task t_B.

VI. Case-Study

An NLP specification of a WLANp receiver is shown in Figure 7. The WLANp receiver receives its data from the periodic source ADC and writes its data to the aperiodic sink $packet$. The aperiodic sink executes when there is data available in the connected buffer. In other words, the sink only executes if data from the source is processed completely. The periodic source is required to run at 250 KHz, so the input stream is sampled every 4 μs.

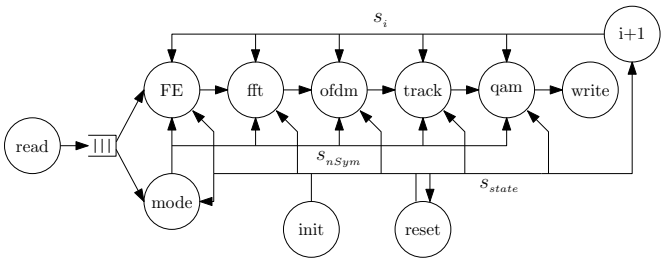


Fig. 8. Task graph of the WLANp receiver from Figure 7.

The receiver has different reception modes. In the first mode the signal is acquired such that the properties of the data signal are known, such as the number of symbols which make up a packet. When this is successful, a mode switch occurs and a whole packet is received. Since a packet has a variable length, the second mode contains a while loop which only terminates if the reception of a packet is complete. Since both modes need the input signal, both modes read data from the source.

After parallelism is extracted from the sequential NLP, the task graph from Figure 8 is obtained. The task graph is reasonably complex due to modes, even though the input program is fairly simple. However, despite the modes in the receiver and the large number of edges, the task graph can still be executed using pipeline parallelism. In the figure it can be seen that a pipeline is formed between the tasks in the while loop from mode 1. During execution, maximum pipelining is only possible if the CBs s_{nSym} and s_i, created from the variables $nSym$ and i, are at least six locations large and the CB s_{state}, created from the variable $state$, is nine locations. Each statement can than read/write to a different location in the CB. The data flow analysis can also decide that smaller buffers are sufficient if the system is fast enough to meet the 250 KHz requirement with less pipelining.

VII. Conclusion

This paper presented an approach for the automatic parallelization of NLPs containing sources and sinks. These sources and sinks can be accessed by statements in different modes and can be either executed strictly periodic or data-driven. After parallelization, synchronization statements are inserted into the extracted task graph to ensure the same functional behavior as the sequential application.

Between the sources and sinks and the application, buffers are inserted which ensure that the periodicity of the sources and sinks is never disturbed in case the execution time estimates that were used at design time, were optimistic. Periodicity of the sources and sinks is ensured by allowing these buffers to overflow and underrun. This introduces potentially local non-determinism in the system.

From the parallelized application, it is shown that a conservative CSDF model can be generated. The generated CSDF model is based on the synchronization in the task graph. Given throughput and latency constraints on the sources and sinks, buffer capacities can be calculated using the CSDF model. It is shown that a CSDF model can be generated by assuming only a single execution of the while loops. In case the while loops are executed more often, it can be shown that data and space will not arrive later than will be derived with the model.

The presented approach is intended for non-safety critical applications that require pipelined execution for performance reasons. The described issues and proposed solutions are our first but potentially interesting contribution to address the issues introduced by sources and sinks in sequential programs in a systematic way.

References

[1] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. 1988.

[2] G. Berry and E. Sentovich. Multiclock esterel. *Correct Hardware Design and Verification Methods*, pages 110–125, 2001.

[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 *SCOPES '08: Proc. of the 11th international workshop on Software & compilers for embedded systems*, pages 33–42. ACM, 2008.

[5] T. Bijlsma, M.J.G. Bekooij, and G.J.M. Smit. Circular Buffers with Multiple Overlapping Windows for Cyclic Task Graphs. 2011.

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

[7] R.L. Bocchino Jr, V.S. Adve, S.V. Adve, and M. Snir. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, pages 4–4. USENIX Association, 2009.

[8] P. Caspi. Embedded control: From asynchrony to synchrony and back. In *Embedded Software*, pages 80–96. Springer, 2001.

[9] P. Caspi and A. Girault. Execution of distributed reactive systems. *EURO-PAR'95 Parallel Processing*, pages 13–26, 1995.

[10] P. Caspi, D. Pilaud, N. Halbwachs, and J.A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *Conference Record of the 14th Annual ACM Symp. on Principles of Programming Languages*, 1987.

[11] International Organization for Standardization. ISO/IEC 9899:TC2: Programming Languages - C. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf, 2005.

[12] Marc Geilen, Stavros Tripakis, and Maarten Wiggers. The earlier the better: A theory of timed actor interfaces. In *14th International Conference on Hybrid Systems: Computation and Control (HSCC'11)*, April 2011.

[13] S.J. Geuns, M.J.G. 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), Grenoble, France*, 2011.

[14] Y. Ishikawa and H. Tokuda. Object-oriented real-time language design: Constructs for timing constraints. In *ACM Sigplan Notices*, volume 25, pages 289–298. ACM, 1990.

[15] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess program. *IEEE transactions on computers*, pages 690–691, 1979.

[16] S.T. Taft and R.A. Duff. *Ada 95 reference manual: language and standard libraries: international standard ISO/IEC 8652: 1995 (E)*. Springer Verlag, 1997.

[17] J.W. van den Brand and M. Bekooij. Streaming consistency: a model for efficient MPSoC design. In *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*, pages 27–34. IEEE, 2007.

[18] S. Verdoolaege, H. Nikolov, and T. Stefanov. PN: a tool for improved derivation of process networks. *EURASIP Journal on Embedded Systems*, 2007(1):19–19, 2007.

[19] M.H. Wiggers, M.J.G. Bekooij, and G.J.M. Smit. Efficient computation of buffer capacities for cyclo-static dataflow graphs. In *Proceedings of the 44th annual Design Automation Conference*, pages 658–663. ACM, 2007.

[20] M.H. Wiggers, M.J.G. Bekooij, and G.J.M. Smit. Monotonicity and run-time scheduling. In *Proceedings of the seventh ACM international conference on Embedded software*, pages 177–186. ACM, 2009.