Utilization Improvement by Enforcing Mutual Exclusive Task Execution in Modal Stream Processing Applications

Guus Kuiper
University of Twente
g.kuiper@utwente.nl

Stefan J. Geuns
University of Twente
s.j.geuns@utwente.nl

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

ABSTRACT

Real-time dataflow analysis techniques for multiprocessor systems ignore that the execution of tasks belonging to different operation modes are mutually exclusive. This results in more resources being reserved than strictly needed and a low resource utilization.

In this paper we present a dataflow analysis approach which takes into account that tasks belonging to different modes often execute mutually exclusive. Therefore less resources need to be reserved to satisfy a throughput constraint and a higher processor utilization can be obtained. Furthermore, we introduce a lock which is used to enforce mutual exclusive execution of tasks during a mode transition when beneficial. The effects of mutual exclusive execution are included in a Structured Variable-Rate Phased Dataflow (SVPDF) temporal analysis model which is used to determine whether adding a lock results in satisfaction of the throughput constraint. This model is generated from a sequential input specification of the application such that deadlock-free execution, even after the addition of locks, is guaranteed.

The applicability and benefits of the approach are demonstrated using a WLAN 802.11g application which switches between a detection and a decoding mode. It is shown that the use of two locks improves the worst-case response times of 3 tasks such that they can share the same processor, which improves the utilization of this processor and frees 2 other processors.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*complexity measures, performance measures*

General Terms

Design, Theory, Verification

SCOPES '15, June 01 - 03, 2015, Sankt Goar, Germany
© 2015 ACM. ISBN 978-1-4503-3593-5/15/06. . .$15.00
DOI: http://dx.doi.org/10.1145/2764967.2764970

Keywords

Real-time, Mutual Exclusion, Dataflow, Automatic Parallelization

1. INTRODUCTION

Dataflow analysis is suitable for the derivation of the minimum throughput of modal real-time stream processing applications that are executed on run-time scheduled multiprocessor systems [25]. An example of modal applications are packet-based software-defined radio applications that contain a detection and a decoding mode. An application can remain in a mode for an indefinite amount of time.

In these applications the tasks that execute, often differ per application mode. However, after a mode switch some tasks of the previous mode might still execute a few iterations as a result of data from a previous mode that is still in the First-In-First-Out (FIFO) buffers and because a pipelined scheduling strategy is employed.

The effects of mode changes on the schedulability of a task set executed on single processor systems have been extensively studied [8, 15, 16, 19, 22]. These works require that mode change control software in the kernel of the operating system takes care that a new mode change is not started during a mode change. An exception are dataflow analysis techniques [5, 6, 17, 26] which do allow the start of mode changes during a mode change. Furthermore, these techniques are intended for multiprocessor systems.

Tasks in different modes are often not active at the same time and can therefore share resources without interfering with each other. Dataflow analysis techniques can be used to analyze this resource sharing for systems in which budget schedulers [18, 25] are applied. These budget schedulers guarantee a minimum budget during a replenishment interval and thereby guarantee that always, thus also during mode transitions, at least a minimum amount of processor time is reserved for the execution of each task. As a result a single Worst-Case Response Time (WCRT) can be derived per task at design time. Throughput analysis of the task graph is based on these WCRTs. The reservation of the resources simplifies this analysis drastically. However, dataflow analysis techniques neglect that resources can be shared when tasks do not interfere which can result in a low utilization of the processors and often more processors will be reserved than what would be needed when the mutual exclusive execution of tasks in different modes would have been taken into account.

In this paper we present a dataflow analysis approach which takes into account that tasks execute mutually exclusive in different modes, which results in a lower resource reservation and a higher processor utilization. To enforce mutual exclusive execution when it is beneficial, a new lock is introduced. This lock allows parallel execution of a group of tasks, but enforces serial execution between groups. The lock is inserted by a compiler, which transforms a sequential specification of the application into a parallel task graph and an SVPDF model [6]. The lock is inserted in such a way that all tasks can still execute in the same order as in the sequential input specification; therefore deadlock free execution is guaranteed. The generated SVPDF model is a dynamic dataflow model in which mutual exclusion can be expressed. This SVPDF model is used to determine whether addition of a lock results in satisfaction of the throughput constraint and is used to compute the required buffer capacities.

The remainder of this paper is outlined as follows. First we position our contribution relative to related work in Section 2. In Section 3 we present the basic idea behind our approach. The different types of mutual exclusivity that we distinguish are described in Section 4. That mutual exclusivity results in tighter WCRTs is shown in Section 5. The realization of our lock is described in Section 6. Furthermore, it is shown that the generated parallel task graph that includes locks is always deadlock-free. Modeling mutual exclusivity in an SVPDF model of the application is explained in Section 7. The applicability and benefits in terms of throughput and processor utilization for a WLAN application are discussed in Section 8. Finally, the conclusions are stated in Section 9.

2. RELATED WORK

In this section we compare our lock with locks described in literature and discuss other approaches to handle and analyze mode switches.

Mutual exclusive access to resources is obtained by making use of locks. Such locks are usually implemented with atomic read-modify-write operations such as test-and-set and load-link-store conditional [4]. These locks are unsuitable for real-time systems because they are based on a retry mechanism which makes them non-starvation-free [10]. Starvation-free versions of locks do exist but often incur a much higher overhead. Examples of starvation-free locks are the Bakery lock [11] and Szymanski's mutual exclusion algorithm [20]. The lock proposed in this paper is starvation-free but does not introduce a high overhead. Key differences with other locks are that the proposed lock enforces an order in which groups of task can acquire the lock.

Ordinary load and store operations are used in the Bakery lock and Szymanski's mutual exclusion algorithm to guarantee mutual exclusive access. These locks require that sequential consistency [12] is supported as the memory consistency model by the multiprocessor hardware. FIFO buffers can also be realized using ordinary load and store operations [14]. However they require a much weaker memory consistency model [23] that only guarantees that writes issued by a processor complete in the order that they are issued and that read and writes that access the same memory do not overtake each other. Circular buffer [2] implementations have been proposed that can be seen as a generalization of these FIFO buffers because they allow multiple readers and writers. The lock proposed in this paper is based on the same principles as these circular buffers.

Techniques for the prevention of overloads that result in violation of the WCRTs during the transition between modes on single processor systems have been deeply investigated in the real-time literature [8, 16, 22] and a survey of mode change protocols for static priority preemptive scheduled systems can be found in [15]. Most of these protocols delay the start of tasks during a mode change. Some of these works are geared towards servers of which budget schedulers are a subclass [19]. All these approaches assume that a subsequent mode change is not started before the previous mode change completes. Furthermore, they are only applicable for acyclic task graphs while the method described in this paper can handle cyclic task graphs and overlapping mode changes are supported. Cycles in the task graphs are a result of data dependencies and of the use of buffers with a bounded capacity.

Classical dataflow models such as Homogeneous Synchronous Dataflow (HSDF), Synchronous Dataflow (SDF) [13] and Cyclo-Static Dataflow (CSDF) [3] can only model static applications, i.e. applications of which their synchronization behavior is independent of the input data. Therefore, these models are unsuitable for the modeling of modal applications. However, the recently introduced Variable-Rate Phased Dataflow (VPDF) [26] and Scenario-Aware Dataflow (SADF) [17, 21] models are suitable to describe modal applications. Generation of a parallel task graph and a corresponding structured version of the VPDF model, which is called SVPDF, is presented in [6]. An advantage of this approach is that the task graph and the corresponding analysis model are deadlock-free. In this paper we present the modeling of mutual exclusivity execution in the SVPDF model. This model is used to show that an *admissible* worst-case schedule exists that satisfies the throughput constraint imposed by the periodic source of the application. A schedule is admissible if all tasks do not execute before sufficient data and space is available. The production moments of the data in this worst-case schedule are upper bounds on the production moments of the tasks in all possible (aperiodic) run-time schedules. The SVPDF model is generated by a compiler from a sequential description of the application in the OIL language.

The dataflow analysis techniques discussed in this paper are applicable in combination with budget schedulers [25]. Budget schedulers are a subclass of servers that guarantee a minimum cycle budget in a replenishment interval. In [25] it has been shown that the worst-case effects of run-time task scheduling by budget schedulers can be included in firing durations of the actors of dataflow graphs. A budget scheduler with priorities is introduced in [18] which allows reducing the WCRT of one task at the cost of an increased WCRT of the other tasks. Most budget schedulers are work-conserving and as a result allocated budget for a task that is not used by this task becomes available for other tasks.

The budget schedulers used in this paper are a subclass of the servers used in [19]. The approach in [19] adapts the parameters of the server during a mode transition by a separate mode change controller in the scheduling kernel. The approach described in this paper does apply a different approach in which parameters of the scheduler are implicitly adapted due to the fact that resources are not used anymore by some of the tasks after a mode transition. Furthermore, the activation and deactivation of tasks is a responsibility of the tasks themselves and is part of the description of the application. This description is a sequential program with while-loops and if-conditions instead of tasks with schedul-

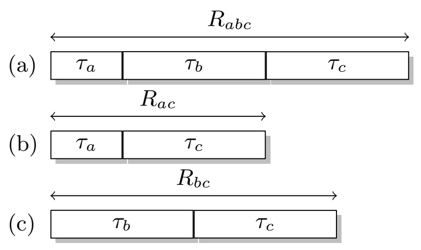


Figure 1: Replenishment interval of mutually exclusive tasks

ing parameters and a separate description of the behavior of a mode change controller. Another important observation is that we are dealing with a multiprocessor system and therefore the implementation of a (centralized) mode change controller would not be straightforward.

3. BASIC IDEA

In this section we present the basic idea behind our approach. With a didactic example we illustrate that the WCRTs of repetitively executed tasks can be reduced and a higher minimum throughput is obtained if information about mutually exclusive execution is taken into account. We derive these tasks from a sequential program in which if-conditions and while-loops are used to describe modes and mode transitions. Tasks execute data-driven such that variations in execution time can be exploited [9]. We show the counterintuitive effect that tasks resulting from different while-loops or branches of an if-then-else statement do not necessarily execute mutually exclusive. We explain why mutual exclusivity can be enforced with locks without introducing deadlock as a result of that the tasks in the task graph are generated from a sequential program. We furthermore explain the modeling of the sequence constraints that result from a lock in an SVPDF model. This model is used for throughput analysis.

That the WCRTs of tasks can be reduced by taking into account that tasks execute mutually exclusive is illustrated with Figure 1. In Figure 1a one replenishment interval is shown of a budget scheduler in which three slices are available for respectively the execution of tasks τ_a, τ_b, and τ_c. These tasks execute until they exhaust their budget after which they continue their execution in the subsequent replenishment interval. A task voluntarily suspends its execution in case it is not enabled because there is insufficient input data or output space to start the execution of the task after which a task switch occurs. In our example we will assume a budget of one, two, and two time units for tasks τ_a, τ_b, and τ_c respectively.

The replenishment interval is reduced in case τ_a and τ_b execute mutually exclusive. The length of the replenishment interval becomes three in case only task τ_a executes and four in case that only τ_b executes, as shown in Figure 1b and Figure 1c respectively. A reduction of the replenishment intervals of the tasks results in a reduction of the WCRTs, as directly follows from the WCRT equation [24] in Equation 1. In this equation ψ_i is the WCRT, R_i the replenishment interval, x_i the Worst-Case Execution Time (WCET), and B_i the budget of τ_i.

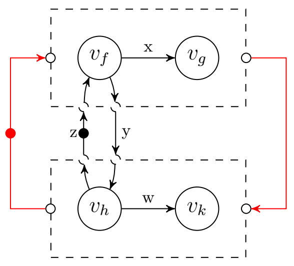
$$\psi_i = x_i + (R_i - B_i)\left\lceil \frac{x_i}{B_i} \right\rceil \tag{1}$$

During a mode transition it might occur that τ_a finishes at the end of its slice and immediately after that a mode switch occurs after which τ_b starts its execution. However, this does not result in a longer WCRT for τ_a and τ_b than R_{ac} and R_{bc} respectively, as will be explained in Section 5.

```
z = 0;
loop {
  loop{
    f(out x, out y, z);
    g(x);
  } while(x);

  loop{
    h(out w, out z, y);
    k(w);
  } while(w);
} while(1);
```

(a) Code example



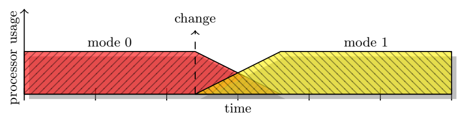
(b) SVPDF model

Figure 2: Mutual exclusion of while-loops

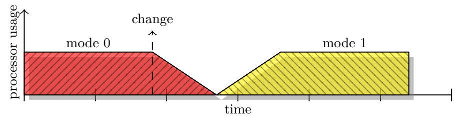
The tasks are derived by a multiprocessor compiler from a sequential program of which an example is shown in Figure 2a. Each while-loop in this program corresponds to a mode of the application. After parallelization a task graph is obtained. This task graph can be modeled with the SVPDF model in Figure 2b. Every function in the sequential program corresponds to one task in the task graph and every task corresponds to one actor in the SVPDF model. These actors are represented by nodes in the SVPDF model. Every variable in the sequential program is converted in a circular buffer and each circular buffer corresponds to at least one edge in the SVPDF model.

To understand the example it is sufficient to assume that the SVPDF model in Figure 2b has an HSDF-like behavior, i.e., actors can only fire if at least one token is present on all its inputs and a token is produced on all outputs when an actor finishes its firing. We can therefore conclude from this model that actor v_f and v_h cannot fire at the same time as a result of the cycle with one token in the model. This cycle is a result of that function h in the second while-loop cannot start before the value y of the first while-loop becomes available. Also, function f cannot execute for successive executions before the value z from the second while-loop becomes available. However, we can also conclude from the model that actor v_g and v_k can fire at the same point in time because they are not part of a cycle with a single token. They can fire at the same time if there is an input token present for v_g and v_k, which can occur after v_f has produced a token on each of its outputs and as a result v_h fires and produces a token for v_k before v_g finishes its execution. A similar situation can occur during the execution of the task graph after τ_f has produced a data item for τ_g and then a mode switch occurs.

Figure 3a illustrates intuitively that executions of tasks that belong to different modes can execute simultaneously during a mode transition. Locks can be used to prevent that tasks belonging to different modes execute simultaneously. This situation is shown in Figure 3b. This figure also shows the counterintuitive effect that the mode transition will in some case occur earlier despite that less tasks execute in parallel during a transition. The reason is that the WCRTs of the tasks can be reduced by making use of the knowledge that they execute mutual exclusively. This results in an improvement of the minimum throughput as computed with



(a) Without mutual exclusion



(b) With mutual exclusion

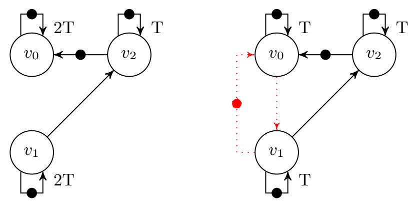
Figure 3: Processor usage during a mode switch

dataflow analysis.

Mutual exclusive execution is usually only enforced between tasks that execute on the same processor. Therefore even if locks are applied, a mode transition usually starts on different processors at different moments in time. As a result, tasks belonging to different modes will execute on different processors at the same moment in time. The lock presented in this paper can also be used to enforce mutually exclusive execution of tasks on different processors which might be beneficial because it can reduce contention at shared memory ports and thereby reduce the execution times of the tasks. However, this option will not be detailed in this paper.

The enforcement of mutual exclusivity with locks results in additional constraints on the order in which the tasks can execute. These constraints are modeled with the red edges in the SVPDF model in Figure 2b. These edges form a cycle with one token and as a result the firings of all actors in a block do not overlap with firings of actors in another block. However the actors in a block can still fire concurrently.

The locks are added in the tasks in such a way that deadlock does not occur. This is possible because the tasks and the task graph are derived from a sequential program which is deadlock-free by definition. Therefore we can insert the acquires and releases of the locks according to the order defined by the sequential program that is still a valid order. Other execution orders of the tasks do not result in a different functional behavior of the tasks because we can rely on the fact that the task graph that we create can be represented as a functionally deterministic dataflow model.



(a) Without lock (b) With lock

Figure 4: HSDF model before and after adding a lock

Adding locks can reduce the WCRTs of the tasks but does not necessarily improve the throughput of the application. The reason is that the lock enforces besides mutual exclusivity also the execution order as defined in the sequential program. This execution order might not be the order that results in the maximum throughput even in the case that the WCRTs are reduced. For ease of understanding we illustrate this with an HSDF example instead of an SVPDF model. In Figure 4a an HSDF model is shown in which actor v_0 and v_1 execute on one processor and v_2 on another processor. The WCETs of the actors is T and ψ_0 and ψ_1 is then $2T$ under the assumption that each actor has a budget T. In this case the throughput is determined by the cycle with the highest cycle mean which is $2T$. Figure 4b corresponds to the case that τ_0 and τ_1 execute mutually exclusive as a result of a lock and that in the sequential program first the function that corresponds to τ_0 is executed before the function corresponding with τ_1. As a result of the lock ψ_0 and ψ_1 are reduced to T. However, because the lock enforces an execution order, additional edges should be added in the HSDF model which are dotted and colored red in Figure 4b. These additional edges increase the maximum cycle mean to $3T$ and reduce the throughput to $\frac{1}{3T}$. This shows that the decision whether adding a lock is beneficial requires global analysis of the dataflow model.

4. TYPES OF MUTUAL EXCLUSIVITY

In this paper we distinguish between two types of mutual exclusivity: intra-iteration and inter-iteration mutual exclusivity.

Tasks are intra-iteration mutually exclusive if there is no overlap in their execution within one iteration of a while-loop. An example of such mutual exclusivity is an if-else-statement in which during one iteration of the loop either functions in the if-branch or the else-branch execute, but not both. Another example is if there is a data-dependency between two statements, preventing them from executing simultaneously.

Tasks can also be inter-iteration mutually exclusive. This means that tasks derived from functions located in the same while-loop do not execute simultaneously when considering different loop iterations. An example of this type would be two functions where the first function writes to a variable read by the second function and vice-versa. In an SVPDF model such a case can be found if there is a block, modeling a while-loop, having two actors on a cycle with one token on that cycle. Tasks that are intra-iteration mutually exclusive do not have to be inter-iteration mutually exclusive. For example the two branches of an if-else-statement are intra-iteration mutually exclusive but due to a pipelined execution they do not need to be inter-iteration mutually exclusive. In such a case tasks from both branches can execute in different iterations of the surrounding while-loop simultaneously.

If tasks in a task graph have intra-iteration and inter-iteration mutual exclusivity, this can be exploited by the method introduced in this paper. If tasks are both intra and inter-iteration mutually exclusive, they always execute mutually exclusive. The derived SVPDF model can be used to determine which tasks are mutually exclusive by searching cycles having one token on them. Exploiting this information does not require a change in the task graph. If such cycles do not exist, the lock introduced in Section 6 can be used to enforce mutual exclusivity and thus to create mutual exclusivity.

5. IMPROVED RESPONSE TIME

In this section we show that reduced replenishment intervals can be substituted in the WCRT equation given that some of the tasks scheduled by a budget scheduler are deactivated during a mode change and others are activated. We will make use of the fact that a budget scheduler allocates budgets to the tasks in a fixed cyclic order. We also make use of the property that a task returns control to the scheduler when not enabled.

As described in Section 3 it can be the case that during a mode transition all tasks scheduled on a processor use their time-slice immediately after each other. Therefore, the situation as depicted in Figure 1a can occur. This suggests that R_{abc} should be used in Equation 1 which is however not always the case. Instead the shorter replenishment R_{ac} and R_{bc} intervals can be used for τ_a and τ_b respectively while for τ_c the longer interval R_{abc} must be used.

The reason that a shorter replenishment interval can be used for τ_a and τ_b is that they are mutual exclusive and because the WCRT is defined as the maximum time between enabling and finish of task. We will make use of case-distinction to show that a valid bound on the WCRT is computed in case we use the reduced replenishment intervals in Figure 1b and Figure 1c in Equation 1 for the computation of ψ_a and ψ_b.

It is given that τ_a and τ_b execute mutually exclusive. Therefore we know that before a mode change only one of these tasks is active. As a consequence we can assume that R_{ac} and R_{bc} are valid replenishment intervals for τ_a and τ_b, respectively.

When a mode change occurs there can be a transition from the first mode in which τ_a is active to the second mode in which τ_b is active. After τ_a finishes its execution it will yield the processor. As a result τ_b will receive its budget in R_{bc} time. A similar situation occurs for the transition from the second mode in which τ_b is active to the first mode in which τ_a is active. Here τ_a will receive its budget in R_{ac} time after τ_b finishes its last execution in the previous mode.

Because in all the possible cases the budgets become available for τ_a and τ_b in respectively R_{ac} and R_{bc} we conclude that correct WCRTs are obtained when they are used in Equation 1.

For τ_c the situation is different than for τ_a and τ_b. For this task it can only be guaranteed that B_c is available in every R_{abc} because between two slices for τ_c there can be an execution of τ_a and τ_b while τ_c is enabled.

The same reasoning as applied in this section can be used to proof similar results for the case that an arbitrary number of mutual exclusive tasks are scheduled together with an arbitrary number of tasks that are not mutual exclusive.

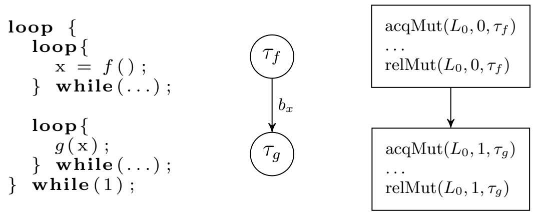
6. MUTUAL EXCLUSIVITY LOCK

This section describes our realization of the lock and the code generation done by our automatic parallelization tool. Furthermore, it presents the proof that the lock insertion method described in this section does not introduce deadlock.

6.1 Realization

We first describe the realization of a lock that is suitable for the most basic case which is when two tasks are made mutually exclusive. We then extend this realization for an arbitrary number of tasks. Finally, we explain an additional generalization to make the lock suitable for mutual exclusive execution of groups of tasks.

An OIL program with two modes in which each mode consists of one function is shown in Figure 5a. In this program the function f is executed for an unknown number of times



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

  loop{
    g(x);
  } while(...);
} while(1);
```

(a) OIL program (b) Task graph (c) Lock usage

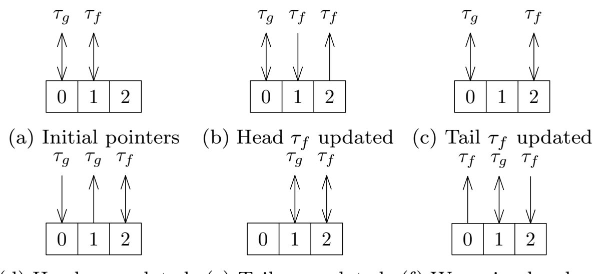
Figure 5: Example containing two modes

after which function g is executed for an unknown number of times. This is repeated forever. The loop conditions are left implicit in order to simplify the example. This program is converted into the parallel task-graph shown in Figure 5b, where task τ_f is extracted from function f. Buffer b_x is extracted from variable x.

The realization of the lock is inspired by the implementation of circular buffers [1, 2]. These buffers can be implemented with ordinary load and store operations instead of atomic read-modify-write operations which is needed to make them starvation-free. These atomic read-modify-write operations are not needed because each shared variable that is used in the buffer implementation is updated by only one task.

Similar to a circular buffer, the lock can be used by a task by calling two functions: *acqMut*, and *relMut*. This is illustrated in Figure 5c. The arguments to these functions are a lock identifier, the execution order and a reference to the task. For every lock it holds that tasks using the same execution order argument can execute simultaneously while tasks with a different execution order argument execute in the order indicated by this argument.

The implementation of the lock consists of a head and a tail pointer for each task using the lock. The head pointer of a task is incremented during an *acqMut* call and the tail pointer is incremented during a *relMut* call. When the pointers reach the end of the array they wrap around to the beginning of the array. Before a head pointer can be updated to a next location it must be verified that no tail pointer of an other task points to that location, i.e. the *acqMut* call blocks until this is the case. If the function *acqMut* blocks, a yield call is executed indicating to the scheduler that the next task can now use its budget by resuming its execution.



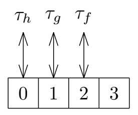
(a) Initial pointers (b) Head τ_f updated (c) Tail τ_f updated
(d) Head τ_g updated (e) Tail τ_g updated (f) Wrapping head τ_f

Figure 6: Head and tail pointer updates for a lock consisting of two tasks

For the two tasks τ_f and τ_g in the example from Figure 5 an array with three elements is allocated. The pointers are initialized as shown in Figure 6a. The head pointer is visualized as pointing upwards and the tail pointer as pointing downwards. Initially the head pointer of τ_f is the only pointer that can be updated without violating the rule that the head pointer may not point to the same array element as the tail pointer of an other task. Thus the *acqMut* call of τ_f updates the head pointer of τ_f, as shown in Figure 6b. After the *relMut* call of τ_f also the tail pointer is incremented as shown in Figure 6c. From this figure we can now conclude that only the head pointer of τ_g can be incremented, see Figure 6d. After function g is executed its *relMut* call will update the tail pointer of τ_g as shown in Figure 6e. At this point only the head pointer of τ_f can be updated after which this pointer will wrap around to the begin of the array as shown in Figure 6f. These steps can be repeated forever.

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

(a) OIL program

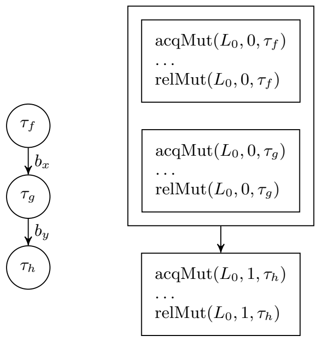


(b) Lock pointer initialization

Figure 7: Example containing three modes

The in the previous paragraph described lock can be made suitable for n tasks in a relatively straightforward way. This lock requires an array with $n+1$ locations. Figure 7 shows such a program containing three modes where all details about variables are left out for clarity. The corresponding task graph consists of three tasks and is omitted here due to its simplicity. The initialization of the pointers of the lock is depicted in Figure 7b. The conditions under which the pointers are allowed to be incremented remains as described in the previous paragraph.

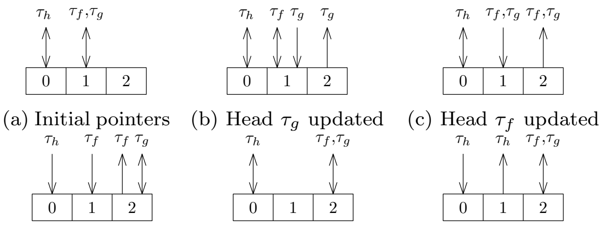
```
loop {
  loop{
    x = f();
    y = g();
  } while(...);
  loop{
    h(y);
  } while(...);
} while(1);
```



(a) OIL program (b) Task graph (c) Lock usage

Figure 8: Example usage of the lock having groups of tasks

This lock can be generalized to support groups of tasks which execute mutually exclusive with other groups of tasks. An example of an OIL program in which such mutual exclusion can be exploited is shown in Figure 8a. In the OIL program there are two modes but now with two functions f and g in the first mode and one function, h, in the second mode. We construct a task graph from it as shown in Figure 8b. That the tasks derived from the functions f and g can execute simultaneously can be seen in Figure 8c. Here,


τ_h τ_f,τ_g
0 1 2
τ_h τ_f τ_g τ_g
0 1 2
τ_h τ_f,τ_g τ_f,τ_g
0 1 2
τ_h τ_f τ_f τ_g
0 1 2
τ_h τ_f,τ_g
0 1 2
τ_h τ_h τ_f,τ_g
0 1 2

(a) Initial pointers (b) Head τ_g updated (c) Head τ_f updated
(d) Tail τ_g updated (e) Tail τ_f updated (f) Head τ_h updated

Figure 9: Head and tail pointer updates for a lock consisting of two groups of tasks

the execution order parameter of these tasks is both zero, indicating there is no constraint between the tasks. The main modification of the lock is that the pointers of the tasks that belong to the same group, point initially to the same location and these pointers can be incremented independently of the position of the other pointers that belong to the same group.

Since there are two groups in this example an array of three elements is created, one element for each group and a free one. The initial locations for the pointers are set according to the defined rules and are shown in Figure 9a. Tasks τ_f and τ_g both belong to the same group and can both move their head. Task τ_g performs this movement first as shown in Figure 9b. Now there are two options; the head of τ_f or tail of τ_g. The first option is shown in Figure 9c. Now both tasks can only update their tail in an arbitrary order for example first τ_g and then τ_h as shown in Figure 9d and Figure 9e. At this point the only possible movement is the head of τ_h as illustrated in Figure 9f. This sequence of head and tail updates can be repeated indefinitely.

6.2 Code Generation

We now show how such a lock can be used by an automatic parallelization tool such that tasks execute mutually exclusive. The code between the calls to the functions *acqMut* and *relMut* executes mutually exclusive as dictated by the rules outlined in the previous section.

For the simple example with two functions in two modes as shown in Figure 5, the implementation of the extracted tasks is shown in Figure 10. The *acqMut* function is the first statement in the outer while-loop and the *relMut* function is the last function such that the first inner loop is mutually exclusive with the second inner loop. The execution order argument is derived from the order of the statements in the sequential OIL program. Because f is before g, task τ_f has number zero and τ_g number one. Basically, an intra-iteration dependency is added between f and g. If the

```
do{
  acqMut(L0, 0, τf);
  acqProd(bx);
  do{
    write(bx, f());
  } while(...);
  relProd(bx);
  relMut(L0, 0, τf)
} while(1);
```

(a) Generated code of τ_f

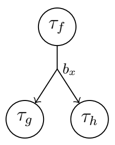
```
do{
  acqMut(L0, 1, τg);
  acqCons(bx);
  do{
    g(read(bx));
  } while(...);
  relCons(bx);
  relMut(L0, 1, τg);
} while(1);
```

(b) Generated code of τ_g

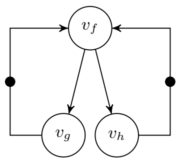
Figure 10: Tasks resulting from Figure 5

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

(a) OIL program



(b) Task graph



(c) Sequential program order

Figure 11: Mutual exclusion in a model application described by a conditional-statement

lock wraps back to execution order number zero, an inter-iteration dependency is added.

A similar generation of code can be done for if-statements. In Figure 11a a simple OIL program with an if-statement is shown and the obtained task graph after parallelization is shown in Figure 11b. Both tasks τ_g and τ_h read from a buffer b_x. The order from the sequential program is visualized in Figure 11c. Because the if-else-statement has two branches, the two possible orders can occur as shown.

Statements in different branches of an if-else-statement must be in one group if they use the same lock. This is because deadlock can occur otherwise because there is no sequential order defined between statements in two branches. However, statements in the if-branch are already intra-iteration mutually exclusive with statements in the else-branch. Statements in one branch can be made mutually exclusive using a second lock.

```
do{
  acqMut(L_0,
    0,τ_f);
  acqProd(b_x);
  write(b_x,
    f());
  relProd(b_x);
  relMut(L_0,
    0,τ_f);
} while(1);
```

(a) Task τ_f

```
do{
  acqMut(L_0,
    1,τ_g);
  acqCons(b_x);
  if(...) {
    g(read(b_x));
  }
  relCons(b_x);
  relMut(L_0,
    1,τ_g);
} while(1);
```

(b) Task τ_g

```
do{
  acqMut(L_0,1,τ_h);
  acqCons(b_x);
  if(...) {}
  else{
    h(read(b_x));
  }
  relCons(b_x);
  relMut(L_0,1,τ_h);
} while(1);
```

(c) Task τ_h

Figure 12: Generated tasks from Figure 11a including a lock

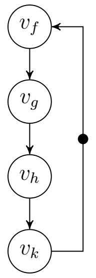
Assume that we indicate that functions g and h should execute mutually exclusive with function f. Note here that g and h are already intra-iteration mutually exclusive, but not inter-iteration and thus an overlap in execution can occur as a result of pipelining. After adding these functions as one group in a lock they also execute inter-iteration mutually exclusive. After adding the lock the implementation of the tasks becomes as shown in Figure 12. The *acqMut* call in these tasks is again the first statement in the while-loop and the *relMut* is the last statement. This ensures again that the entire loop body is mutually exclusive with the loop body of other tasks. Note here that also the acquire and release functions for the buffers are placed around the if-statement. This enables the derivation of an SVPDF model and guarantees a deadlock-free execution when no mutual exclusivity locks are used.

6.3 Deadlock-freedom

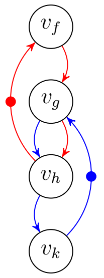
In this section we explain in more detail why the insertion of the acquire and release calls for the locks will not introduce deadlock.

```
loop {
  f();
  g();
  h();
  k();
} while(1);
```

(a) OIL program



(b) Constraints in the sequential program



(c) Constraints resulting from two locks

Figure 13: Sequential program consisting of four functions. Ordering is enforced in the model by partially preserving the sequential ordering

That the resulting task graph will always be deadlock-free is explained using the OIL program shown in Figure 13a. The ordering constraints resulting from the sequential program are shown in Figure 13b. Assume that we require that the tasks that result from the functions f, g, and h should execute mutually exclusive as well as that the tasks that result from the functions g, h and k should execute mutually exclusive. To achieve this we make use of two locks. The first lock enforces the execution order f, g, h, and then f again in the next iteration. This is modeled by the red edges in Figure 13c. The second lock enforces the execution order g, h, k and then g again in the next iteration. This is modeled by the blue edges in Figure 13c. Each time we refer to the next iteration an initial token is placed on the corresponding edge.

That the resulting task graph will remain deadlock-free after locks are added can be seen as follows. When a number of groups of statements in a loop is made mutual exclusive, intra-iteration dependencies are added between these groups. As presented in Section 6, these intra-iteration dependencies are added in such a way that they follow the order of the statements in the sequential specification. These added intra-iteration dependencies can thus never make the sequential schedule inadmissible. In Figure 13c dependencies are added for example from function f to function g and from function g to function k. The sequential schedule of Figure 13b shows that function g is scheduled after f and function k after g. Therefore, after adding the constraints of the lock, the sequential schedule is still admissible.

Second, when making groups of statements mutual exclusive, an inter-iteration dependency is added from the last group back to the first group. In the sequential schedule we have that the first statement of a loop executes after the last statement of the previous iteration of that loop. A group of statements can by definition never contain a statement that occurs earlier or later in the sequential specification than this first or last statement respectively. Adding an inter-iteration dependency between the last mutual exclusive group to the first group can thus never invalidate the sequential schedule. Consider for example the dependency in Figure 13c that prescribes that function f in iteration $i+1$ can only be executed after function h in iteration i is finished. This dependency does not invalidate the sequential schedule because in this schedule, function h in iteration i executes before function k in iteration i which on its turn executes before function f in iteration $i+1$.

7. SVPDF MODEL

In this section it will be shown that a corresponding SVPDF temporal analysis model can always be derived from a sequential OIL program in which the locks are specified. An SVPDF model is based on the VPDF model in which additional structure is added in the form of hierarchical blocks to enable efficient analysis. We first introduce the SVPDF model and its derivation from an OIL program without considering the locks. This is followed by the modeling of the constraints that result from the locks. In this section the modeling approach is shown for one lock, but the modeling of multiple locks is analogous to the modeling of one lock.

In this paper we only consider the derivation of the SVPDF model for scalar variables. However, actors in this model can be extended with phases indicating a sequence of how many tokens need to be consumed or produced every firing of an actor [7]. This number of tokens is based on the synchronization done by tasks. The modeling of mutual exclusivity can be done in a similar way as described in this paper for this more expressive model. For ease of understanding we therefore omit the phase information from the model.

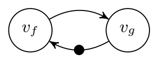
An SVPDF model is a directed graph $G = (V, E, P, \delta, \rho)$. Here V is a set of actors and $(v_i, v_j) \in E$ is the set of edges, with $v_i, v_j \in V$. Actors in an SVPDF model are not auto-concurrent, meaning that at most one firing of an actor can occur simultaneously. An actor can fire if one token is available on each of its input edges and after ρ time a token is produced on each of its output edges, with $\rho : V \rightarrow \mathbb{N}$. The number of initial tokens on an edge is given by $\delta : E \rightarrow \mathbb{N}$.

An SVPDF model is structured into blocks with port actors on the edges of a block. A block is characterized by a parameter $p \in P$. A parameter p defines the number of consecutive firings of actors in that block in respect to the actors surrounding that block. The value of p is unknown during analysis and can be infinite. A block only introduces structure and does not fire itself. Port actors are used to provide communication between actors in and outside of a block. A port actor either converts a token on an input edge directed from outside of a block inwards to p tokens on its output edges or it converts p tokens to one token if the direction of the edges is from inside a block to outside of that block. A more detailed explanation of the SVPDF model and port actors can be found in [6].

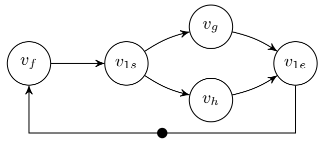
A task graph without mutual exclusive tasks can be modeled as an SVPDF model as follows. For every task an actor is included in the model. Every buffer is modeled by two oppositely directed edges where tokens on the edge from the producer to the consumer represent the full locations in the buffer and tokens on the other edge, empty locations. The initial tokens on this edge equals the buffer capacity. For every while-loop in the OIL program a block is included in the model. If a task contains this loop, the actor corresponding with this task is a (nested) child of the block corresponding with this loop. Blocks are nested analogously to the structure of while-loops in the OIL program. The value of the parameter characterizing a block corresponds to the number of iterations of the while-loop.

We now show that the lock as generated by the method introduced in this paper can be modeled in an SVPDF model. Two cases can be distinguished in the generation of the lock, either groups of tasks in one while-loop are made mutually exclusive, or these groups are distributed over multiple while-loops.

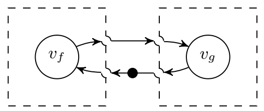
We first consider the most simple case of two groups of tasks in one while-loop, where each of the groups consists of only one task. When tasks are made mutually exclusive this can be represented by a cycle through the corresponding actors with one initial token on that cycle. This is illustrated by Figure 14a which contains two actors corresponding with two mutually exclusive tasks. Between these two actors a cycle is added consisting of two oppositely directed edges representing the ordering constraints enforced by the lock. The initial token on the bottom edge indicates the task that is enabled first by the lock, i.e. the task corresponding with the first function in the sequential ordering specified by the OIL program.



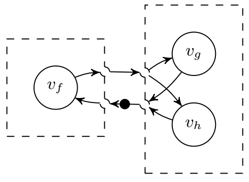
(a) Model of two tasks



(b) Model of three tasks divided into two groups



(c) Model of two tasks in two loops



(d) Model of two groups of tasks in two loops

Figure 14: SVPDF models corresponding with mutually exclusive tasks

When there are multiple tasks in a group no single cycle can be created anymore in the model because tasks in a group can execute simultaneously. Therefore, for every group consisting of more than one task two additional actors are added to the model. The first actor represents that all tasks in a group can start after the tasks in the previous group have finished. The second actor represents that the next group can only start after all actors in the current group have finished their firing. This is illustrated in Figure 14b in which there are two groups of tasks. The first group contains only one task, and thus no additional actors are required. The second group contains two tasks and thus two corresponding actors. Two additional actors are now added. Actor v_{1s} allows both actors v_g and v_h to start because separate edges (v_{1s}, v_g) and (v_{1s}, v_h) are added. Actor v_{1e} enforces that both actors are finished before actors in the next group can fire, which is again actor v_f in the figure. This is modeled by an edge from every actor in the group to actor v_{1e}.

When groups of tasks belonging to different while-loops, the corresponding actors are in different blocks. The approach described above must then be modified such that the correct rate conversion occurs by means of port actors. A port actor is added whenever a constraint following from the lock generation crosses a while-loop boundary and thus results in a rate-conversion. Figure 14c shows an example of two groups of which the corresponding tasks are in different while-loops. In the example a group consists of one task, and thus one actor. In the figure the cycle from actor v_f to v_g and back crosses four block boundaries and thus four port actors are added. These port actors model that a task

```
mutex (detectHeader) (decodeHeader);
mutex (detectHeader decodeHeader) (fft);

source ADC @ 250 kHz;

loop{
  loop{
    detectHeader(ADC, out vh, out h);
    if(vh){
      NSym' = decodeHeader(h);
    }
  } while(!vh);
  n = 0;
  loop{
    x = fft(ADC);
    y = demap(x);
    z = deint(y);
    w = convDecode(z);
    crc(w);
    n' = n + 1;
  } while(n < NSym);
} while(1);
```

Figure 15: Simplified WLAN application

in the corresponding while-loop can execute until the loop condition becomes false. The initial token that indicates which actor can initially start is placed before the port actor which is located on the outer-most block corresponding with the first while-loop in the ordering defined by the sequential specification. In the figure this token is placed before the port actor on the top in the left block, assuming that the left block corresponds to the first while-loop.

Finally, groups of tasks in multiple while-loops can also contain multiple tasks per group. This is illustrated in Figure 14d where the second group contains two tasks. Here the port actors are also used to indicate the simultaneous enabling of tasks in a group and the waiting until all tasks in a group are finished. Thus, these port actors are used instead of the two additional actors inserted for the case shown in Figure 14b.

8. CASE-STUDY

In this section we illustrate the approach presented in this paper by means of a simplified WLAN 802.11g receiver application of which the OIL program is shown in Figure 15. First a header must be detected by the *detectHeader* function in an input stream delivered by a source ADC executing time-triggered at 250 kHz. After a header is found it is decoded by the *decodeHeader* function and then *NSym* symbols in the received packet are decoded by the functions in the second inner while-loop. In this loop, first a symbol is transformed by an *fft* into the frequency domain. The resulting data is then demapped, deinterleaved and convolutional decoded. Finally, a CRC check is performed to verify the correctness of the resulting data.

From this WLAN application a task graph is extracted by the compiler with seven tasks and seven buffers. The periodic source imposes a throughput constraint of 250 kHz which corresponds to a period of 4 μs.

In this example we assume that the *detectHeader*, *decodeHeader* and *fft* task execute on the same processor and a WCET of 3 μs, 1 μs, and 3 μs, respectively. If we allocate a budget of 0.5 μs in a replenishment interval of 1.5 μs for these tasks then it follows from Equation 1 that the WCRT of the *detectHeader* task is equal to 9 μs. Because the period of the source is 4 μs we can immediately conclude that the throughput constraint cannot be met.

By inserting a lock we can reduce the WCRT of the *detectHeader*, *decodeHeader* and the *fft* task. We can indicate in an OIL program by means of the *mutex* keyword that tasks must execute mutually exclusive, as shown in Figure 15. The parameters behind this keyword are groups of tasks and each group of tasks is encapsulated by brackets. In the WLAN specification in Figure 15 there are two groups of tasks behind the first *mutex* keyword. The first group contains the *detectHeader* task and the second group the *decodeHeader* task. As a result of the lock only one of these tasks execute at any point in time on the processor and their execution order will be the order from the OIL program. Behind the second *mutex* keyword there are two groups of tasks with in the first group the *detectHeader* task and *decodeHeader* task and in the second group the *fft* task. As a result these three tasks obtain a WCRT equal to their WCET. These WCRTs are low enough to meet the throughput constraint given that the buffers are sized properly by making use of the SVPDF model of the application.

The SVPDF model of the WLAN application is shown in Figure 16. The source is not shown for clarity of the figure. In this figure the red dotted arrows denote the edges added to enforce mutual exclusivity between the *detectHeader*, *decodeHeader* and the *fft* tasks and between the *detectHeader* and *decodeHeader* tasks. The remaining tasks all run on a separate processor and therefore have a WCRT equal to their WCET which are, in application order starting with *demap*, 2.5 μs, 3 μs, 2.5 μs and 4 μs for *crc*. With this SVPDF model buffer sizes are computed for δ_h, δ_{hv}, δ_x, δ_y, δ_z, δ_w, δ_{NSym} of 1, 1, 2, 2, 2, 2, and 1, respectively.

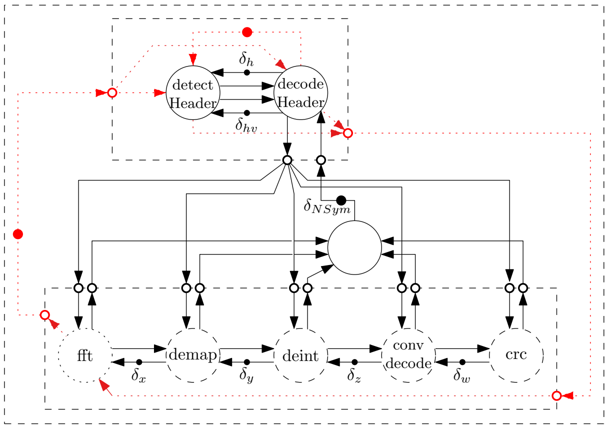


Figure 16: SVPDF model of the application in Figure 15

Figure 17 shows an execution trace derived with a dataflow simulator given the WCRTs when mutual exclusive execution is enforced. The numbers in the traces indicate the invocation number of the tasks. The trace is shown for a packet size of 3 symbols. As a result of the applied locks the *detectHeader*, *decodeHeader*, and *fft* task can execute on the same processor as is visualized in the trace for *Processor1*, instead of that 3 processors are required. This improves the utilization of one processor and frees two other processors. The execution of the application is pipelined and tasks belonging to different modes execute on different processors at the same point in time despite that locks are applied. The trace shows for example that the *detectHeader* and *deint* execute in parallel.

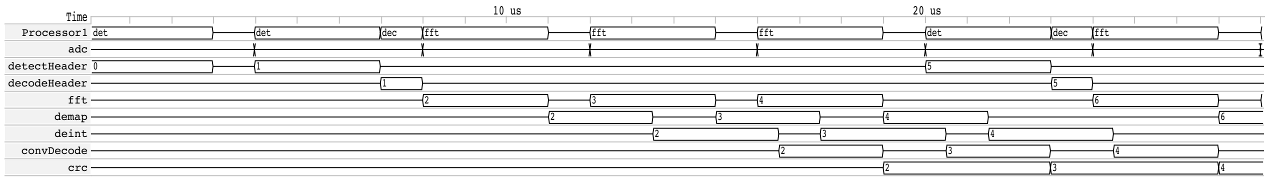


Figure 17: Execution trace of the WLAN application from Figure 15 with mutual exclusivity applied

9. CONCLUSION

In this paper we presented a dataflow analysis approach that takes into account that tasks execute mutually exclusive which results in reduced worst-case response times and an improved processor utilization. We furthermore introduced a starvation-free lock which allows us to enforce mutual exclusive execution of tasks. This lock allows parallel execution of tasks in a group of tasks but enforces sequential execution between groups of tasks. A key difference with existing locks is that groups of tasks can only acquire the lock in a predefined order.

We furthermore showed that mutual exclusive execution of tasks can be modeled in an SVPDF model which is used for checking whether after adding locks the throughput constraint is satisfied. This model is generated by a compiler from a sequential OIL program that describes the modal real-time stream processing application.

We also showed that the resulting parallel task graph is deadlock-free despite that additional constraints are introduced that enforce an execution order of groups of tasks as a result of the locks. The task graph is deadlock free because lock statements are added such that no constraints are introduced that prevent the execution order as defined by the sequential program.

That the introduction of our lock in an application can improve the processor utilization is demonstrated using a WLAN application. In this application 2 locks are introduced. Insertion of these locks reduced the worst-case response times such that 3 tasks can share the same processor which improves the utilization of this processor and frees 2 other processors.

Though we expect that the programmer has often a good idea about which tasks should be made mutually exclusive, an algorithm which determines which task should be made mutually exclusive is considered interesting future work.

10. REFERENCES

[1] T. Bijlsma, M. Bekooij, and G. Smit. Circular buffers with multiple overlapping windows for cyclic task graphs. *Int'l Conf. on High-Performance Embedded Architectures and Compilers (HiPEAC)*, 2011.

[2] T. Bijlsma et al. Communication between nested loop programs via circular buffers in an embedded multiprocessor system. In *Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*. ACM, 2008.

[3] G. Bilsen et al. Cyclo-static dataflow. *IEEE Trans. on Signal Processing*, 44(2):397–408, 1996.

[4] D. Culler, H. Singh, and A. Gupta. *Parallel Computer Architecture: a hardware/software approach.* Morgan Kaufmann, 1999.

[5] M. Geilen and S. Stuijk. Worst-case performance analysis of synchronous dataflow scenarios. In *IEEE/ACM/IFIP Int'l Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. ACM, 2010.

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

[7] S. Geuns, J. Hausmans, and M. Bekooij. Temporal analysis model extraction for optimizing modal multi-rate stream processing applications. In *Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2014.

[8] Q. Guangming. An earlier time for inserting and/or accelerating tasks. *Real-Time Systems*, 41(3):181–194, 2009.

[9] J. Hausmans et al. Two parameter workload characterization for improved dataflow analysis accuracy. In *Real-Time and Embedded Technology and Applications Symp. (RTAS)*, 2013.

[10] M. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proc. ACM Symp. on Principles of Distributed Computing (PODC)*, 1988.

[11] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8), 1974.

[12] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9), 1979.

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

[14] A. Nieuwland et al. C-HEAP: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems. *Design Automation for Embedded Systems (DAES)*, 7(3), 2002.

[15] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Syst.*, 26(2):161–197, 2004.

[16] L. Sha et al. Mode change protocols for priority-driven preemptive scheduling. *Real-Time Systems*, 1(3):243–264, 1989.

[17] F. Siyoum et al. Automated extraction of scenario sequences from disciplined dataflow networks. In *Int'l Conf. on Formal Methods and Models for Codesign (MEMOCODE)*, 2013.

[18] M. Steine, M. Bekooij, and M. Wiggers. A priority-based budget scheduler with conservative dataflow model. In *Euromicro Conf. on Digital System Design Architectures, Methods and Tools (DSD)*. IEEE, 2009.

[19] N. Stoimenov et al. Resource adaptations with servers for hard real-time systems. In *ACM Int'l Conf. on Embedded Software (EMSOFT)*. ACM, 2010.

[20] B. Szymanski. A simple solution to Lamport's concurrent programming problem with linear wait. In *Proc. Int'l Conference on Supercomputing*, 1988.

[21] B. Theelen et al. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *Int'l Conf. on Formal Methods and Models for Codesign (MEMOCODE)*. IEEE, 2006.

[22] K. Tindell, A. Burns, and A. Wellings. Mode changes in priority preemptively scheduled systems. In *IEEE Real-Time Systems Symp. (RTSS)*, 1992.

[23] J.-W. van den Brand and M. Bekooij. Streaming consistency: a model for efficient MPSoC design. In *Euromicro Conf. on Digital System Design Architectures, Methods and Tools (DSD)*, 2007.

[24] M. Wiggers, M. Bekooij, and G. Smit. Modelling run-time arbitration by latency-rate servers in dataflow graphs. In *Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2007.

[25] M. Wiggers, M. Bekooij, and G. Smit. Monotonicity and run-time scheduling. In *ACM Int'l Conf. on Embedded Software (EMSOFT)*, 2009.

[26] 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.