Unified Dataflow Model for the Analysis of Data and Pipeline Parallelism, and Buffer Sizing

Joost P.H.M. Hausmans[§] joost.hausmans@utwente.nl
Stefan J. Geuns[§] stefan.geuns@utwente.nl
Maarten H. Wiggers maarten.wiggers@gmail.com
Marco J.G. Bekooij[§ ¶] marco.bekooij@nxp.com

[§] University of Twente, Enschede, The Netherlands
[¶] NXP Semiconductors, Eindhoven, The Netherlands

***Abstract*—Real-time stream processing applications such as software defined radios are usually executed concurrently on multiprocessor systems. Exploiting coarse-grained data parallelism by duplicating tasks is often required, besides pipeline parallelism, to meet the temporal constraints of the applications. However, no unified model and analysis method exists that can be used to determine the required amount of data and pipeline parallelism, and buffer sizes simultaneously.**

This paper presents an analysis method which can determine the required amount of data parallelism by describing data parallelism in a dataflow model without replicating dataflow actors. This allows to make a trade-off between the amount of data and pipeline parallelism that is required to meet the temporal constraints of the application. It is also shown how large the buffers need to be such that the determined amount of data and pipeline parallelism required for the satisfaction of the throughput constraint, can be realized. Furthermore, it is shown that the use of the applied circular buffers enables the proposed dataflow modeling.

The presented analysis method is demonstrated using a WLAN 802.11p transceiver application. This application contains multi-rate behavior and has a cyclic data dependency because of a re-encoding loop. Given the real-time constraints of the application, sufficient buffer sizes and sufficient data parallelism are derived.

I. Introduction

Real-time stream processing applications, such as software defined radios, often have strict temporal constraints imposed by periodic sources or sinks in the applications. Such applications are often executed on Multiprocessor Systems-on-Chip (MPSoCs) to meet these periodic constraints. Stream processing applications typically contain a fair amount of pipeline parallelism and this pipeline parallelism is used to utilize the different processors of the MPSoC [1].

One of the biggest challenges when developing such real-time systems is to guarantee the temporal constraints of the application. Temporal analysis methods are required to give such guarantees. The temporal behavior of stream processing applications can be accurately modeled by using timed dataflow models [2], [3]. Dataflow models exist which support static [4] as well as dynamic behavior [5], [6] and can be extracted automatically from stream processing applications [7]. Next to that, methods exist to verify the temporal constraints of applications [8], [9] and dataflow models can also be used for the optimization of applications given such temporal constraints [10].

Streaming applications exploit the inherent pipeline parallelism to increase the achieved throughput. However, exploiting only pipeline parallelism might not be sufficient to meet the temporal constraints of applications. Exploiting coarse-grained data parallelism is often required as well and can also increase the throughput significantly [1]. Data parallelism uses the fact that stateless tasks can be replicated such that different data samples can be processed in parallel by these replicated tasks. Replicating tasks can result in smaller buffers than is possible when only pipeline parallelism is used to meet the throughput requirement. On the other hand, pipeline parallelism often requires less additional processing resources. Efficient implementations of applications therefore trade off data parallelism with pipeline parallelism to achieve their throughput requirements.

Timed dataflow models are often used to analyze the throughput of streaming applications. In such dataflow models, actors model tasks. Traditionally, task replication is modeled by duplicating actors in split-join constructs [1], [11]. However, such split-join constructs are difficult to parameterize, which hinders the analysis of the trade-off between data and pipeline parallelism.

In this paper, we present an analysis method based on Synchronous Dataflow (SDF) models [4] in which data parallelism is modeled simultaneously with the pipeline parallelism of the application. Data parallelism is modeled in the SDF model without replicating actors which allows us to make the trade-off between data and pipeline parallelism.

Dataflow analysis methods exist which can determine minimal buffer sizes by using the trade-off between the amount of pipeline parallelism in different buffers [12], [13], [14], [15]. These methods model the size of buffers as initial tokens which are parameterized in the dataflow model. In this paper we extend these approaches and also use initial tokens that are parameterized to model the amount of replication of a task. Because buffers as well as data parallelism are modeled with initial tokens that are parameterized, existing buffer sizing techniques can be used to determine the required amount of data and pipeline parallelism simultaneously. Furthermore, the analysis methods are applicable for cyclic task graphs.

Previous dataflow analysis methods use self-edges to limit the auto-concurrency of actors. These self-edges contain one initial token to obtain the property that firings of the actors do not overlap. We use the number of self-edge tokens to model the replication of the corresponding task. We show in this paper that this model can be accurate and that it is conservative. To show this, we introduce a new expansion method from an SDF graph to an equivalent Homogeneous Synchronous Dataflow (HSDF) graph. For the proposed dataflow model it is required that the worst-case response time of a task is independent of the amount of replication of the task. This is enabled by the applied circular buffers by which tasks communicate. The used circular buffer has support for multiple producers and consumers. Implementing the communication with the

This work is supported by the Sensor Technology Applied in Reconfigurable systems for sustainable Security (STARS) project.

replicated tasks by separate FIFO buffers is in general not possible for our approach because it does not correspond with the proposed dataflow modeling.

The outline of the paper is as follows. Related work is discussed in Section II. The basic idea of the analysis method is presented in Section III. One of the enablers of the proposed analysis method is the use of a specific type of circular buffer. The properties and the implementation of this circular buffer is discussed in Section IV. Section V presents the modeling of data parallelism by means of an SDF model and contains proofs regarding the accuracy and conservativeness of the modeling. In Section VI the combination of pipeline parallelism and data parallelism is described in more detail and the trade-off between data and pipeline parallelism is discussed. The approach is evaluated in a case-study in Section VII and we conclude this paper in Section VIII.

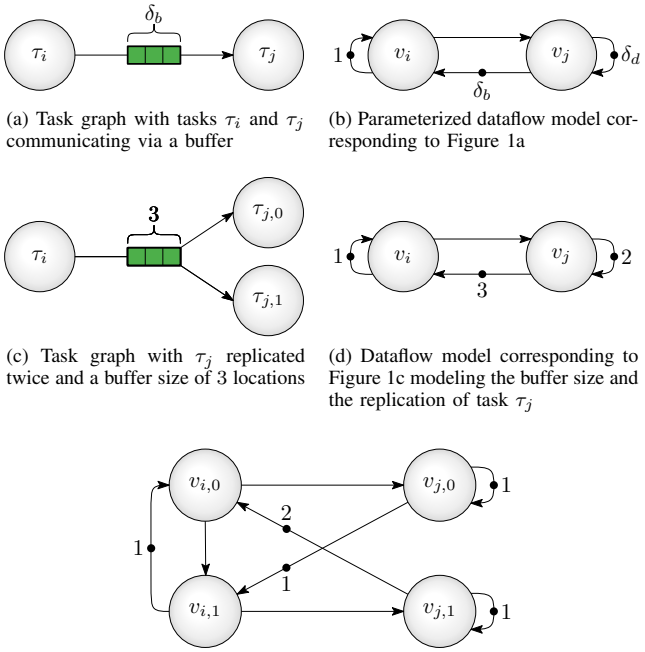
II. Related Work

Data parallelism is often used to increase the amount of parallelism of applications. For example polyhedral models are used in [16] to exploit the available data parallelism in loops with a finite number of iterations. However, the temporal behavior of the applications is not modeled and also the combination with pipeline parallelism is not considered. In [17] the throughput of applications is determined for cyclic polyhedral models with data parallelism. However, the combination with pipeline parallelism is not considered.

It is shown in the context of the StreamIt language [18] that the combination of pipeline parallelism and coarse-grained data parallelism is important [1]. Other methods acknowledge the benefits of the combination of data and pipeline parallelism [19], [20] and use dataflow models to analyze the temporal behavior of data parallel tasks. However, none of these methods determines the amount of data parallelism that is required to meet the throughput constraint of an application.

In [11] a method is presented that also does compute the required amount of data parallelism. It uses an Integer Linear Programming (ILP) to determine which of the predefined set of replications ensures that the temporal constraints can be met. However, just as the other methods mentioned above, it only considers acyclic task graphs. Next to that, it does not determine the required amount of data parallelism and pipeline parallelism simultaneously. The above mentioned papers model data parallelism in a dataflow model by means of duplicating the dataflow actor corresponding to a task. Next to that, they use explicit split and join tasks to communicate with the data parallel tasks. Such split-join constructs are difficult to parameterize because the graph topology depends on the amount of data parallelism. This hinders the trade-off between data and pipeline parallelism. In [11], pipeline parallelism is determined separately from data parallelism, which means that the trade-off between data and pipeline parallelism is not considered.

We model data parallelism by using only one dataflow actor corresponding to a task and allow this actor to occur multiple times in parallel in the schedule. This is modeled using a self-edge with multiple tokens which is parameterized in the dataflow model. This allows to use existing buffer sizing algorithms to determine the required amount of data parallelism simultaneously with the required amount of pipeline parallelism. Furthermore, the analysis method of this paper is applicable for cyclic task graphs and also explicitly makes the trade-off between pipeline parallelism and data parallelism. In our approach communication between tasks is done via



(a) Task graph with tasks τ_i and τ_j communicating via a buffer

(b) Parameterized dataflow model corresponding to Figure 1a

(c) Task graph with τ_j replicated twice and a buffer size of 3 locations

(d) Dataflow model corresponding to Figure 1c modeling the buffer size and the replication of task τ_j

(e) Expanded dataflow model corresponding to the dataflow model of Figure 1d

Figure 1. Illustration of the basic idea of the proposed modeling of data parallelism using dataflow models.

buffers with support for multiple readers and writers [21], [22]. Therefore, we do not need split and join tasks.

Buffer sizes as well as data parallelism are modeled with initial tokens that are parameterized in the dataflow model. Buffer sizing methods exist which determine the minimal amount of initial tokens that are required to meet a throughput constraint. From our analysis follows that such buffer sizing methods can also be used to determine the required amount of data parallelism simultaneous with the minimal buffer sizes. As we will show in Section VI, a trade-off exists between increasing the size of buffers and increasing the amount of data parallelism. This trade-off can be resolved by using cost functions for initial tokens in combination with existing buffer sizing methods. Buffer sizing methods which give exact solutions exist [12], [14]. The main drawback of these methods is that they have a Non-Polynomial (NP) time-complexity. In fact, finding the minimal amount of buffering that is required to meet a throughput constraint is shown to be an NP-complete problem [14]. Buffer sizing algorithms with a polynomial time-complexity are also developed [23], [13], [15]. They are based on linearizations of dataflow schedules and find conservative (not exact) buffer sizes which guarantee the throughput constraint. Such linearization methods are even more relevant because they also enable the compositional analysis of dataflow models [24]. We can also use such approximation algorithms. However, due to the linearization of schedules, they do not accurately capture the trade-off between pipeline and data parallelism, which is detailed in Section VI.

III. Basic Idea

This section illustrates the basic idea behind the proposed analysis method for data and pipeline parallelism.

Traditionally, self-edges with one token are used to limit the auto-concurrency of dataflow actors. Such self-edges pre-

vent overlap of firings of actors which corresponds with the fact that the current execution of a task needs to be finished before the next execution can start. We propose to use more than one token on such a self-edge to model data parallelism. On the model side this looks straightforward. The maximum number of overlapping firings of the dataflow actor is limited by the number of self-edge tokens. However, two important issues arise. First, we need to prove that these overlapping firings indeed correspond with the executions of the data parallel tasks. And second, we have to show that the temporal behavior is modeled conservatively.

For the second problem it is important that the worst-case response time of a task is independent of the amount of data parallelism. This response time is the maximum time a task takes to finish an execution after it is enabled. Furthermore, we have to show that buffers are modeled conservatively. In the proposed modeling, the space of a buffer is shared between the data parallel tasks. This is not possible when a separate buffer is used for each replication. We therefore use a circular buffer with support for multiple producers and multiple consumers [22]. In Section IV we show that also separate buffers for each replication can be used as long as the total buffer size is a multiple of the amount of replication.

Figure 1 illustrates the idea of our dataflow modeling approach and the problems that arise when dataflow actors are replicated to model data parallelism. Figure 1a shows a task graph of two tasks, τ_i and τ_j, connected by a buffer with δ_b locations. Figure 1b shows the dataflow model corresponding to this task graph. The buffer is modeled with two opposite directed edges and the two tasks are modeled with two actors. Dataflow actor v_i, corresponding to τ_i, cannot be replicated because τ_i has state. This is modeled with a self-edge containing one token which is equal to how self-edges were previously used to limit the auto-concurrency of dataflow actors. The self-edge of actor v_j, corresponding to τ_j, illustrates the new modeling technique and can contain more than one token to denote the replication of task τ_j. Both the parameterized number of tokens modeling buffer size and data parallelism can be determined simultaneously by existing buffer sizing methods such that the required throughput is guaranteed.

In Figure 1c, the buffer size δ_b is chosen to be equal to three and task τ_j is replicated twice into tasks $\tau_{j,0}$ and $\tau_{j,1}$. Figure 1d shows the corresponding dataflow model. As can be seen in Figure 1c, both task τ_i as well as tasks $\tau_{j,0}$ and $\tau_{j,1}$ share the same buffer. This is implemented by using a circular buffer with support for multiple consumers and multiple producers. Task τ_i writes data to the buffer in a consecutive order. Furthermore, each location is read by one consuming task. The locations that are read by a task are fixed. In this example, execution n of task $\tau_{j,0}$ uses the data of location $(2 \cdot n) \mod 3$ in the buffer and execution n of task $\tau_{j,1}$ uses the data from location $(2 \cdot n + 1) \mod 3$. Task τ_i can reuse a location when both tasks $\tau_{j,0}$ and $\tau_{j,1}$ no longer need that location.

Figure 1e shows the expanded dataflow model which is used to show the correctness of the modeling of data parallelism. This expanded dataflow model is only used to show the relation between the task graph and the dataflow model and is not used to determine the right values for the parameters. The expanded dataflow model is equivalent to the dataflow model of Figure 1d. As can be seen, actor v_i is split into two actors, $v_{i,0}$ and $v_{i,1}$, which are tightly coupled by a cycle with one token. This means that these two actors fire consecutively. On the other hand, actor v_j is split into actors $v_{j,0}$ and $v_{j,1}$ which both have their own self-edge. This means that they can fire independently of each other, which corresponds to a data parallel execution of the corresponding tasks. Actor $v_{j,0}$ corresponds to task $\tau_{j,0}$ of Figure 1c and $v_{j,1}$ corresponds to $\tau_{j,1}$. The firings of both actors $v_{i,0}$ and $v_{i,1}$ correspond to executions of task τ_i.

The edges between actors $v_{i,0}$ and $v_{i,1}$ and actors $v_{j,0}$ and $v_{j,1}$ model the buffer between the corresponding tasks. The topology of these edges depends on the size of the buffer. As can be seen in Figure 1e, for a buffer of size three, they form the cycle $((v_{i,0}, v_{j,0}), (v_{j,0}, v_{i,1}), (v_{i,1}, v_{j,1}), (v_{j,1}, v_{i,0}))$ through all four actors which illustrates that the corresponding tasks reuse each others buffer locations.

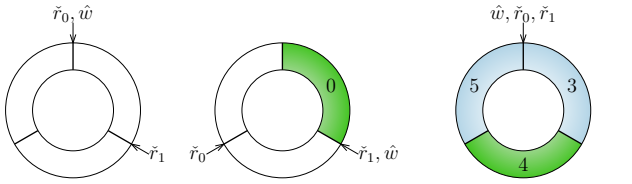
When data parallelism is directly modeled using replication of the corresponding dataflow actors, one would directly end up with the expanded dataflow model of Figure 1e. However, in such a model it would not be possible to model a parameterized size of a buffer because the topology of the edges cannot be changed dynamically. Therefore, it is not possible to make a trade-off between data and pipeline parallelism.

IV. Implementation

The previous section illustrated how applications with data parallel tasks can be analyzed using dataflow models. In this section we present how such applications can be implemented. Specifically, we discuss an implementation of the buffers between tasks for which the synchronization costs are not dependent on the amount of data parallelism that is used.

Often tasks with data parallel behavior are implemented by adding split and join tasks and by using a separate buffer for each replication of a task [1], [11]. This approach has a number of drawbacks. First of all, because each replication has a separate buffer, the space of the buffers cannot be used efficiently. Secondly, the split and join tasks need to be mapped to the platform which means that less resources remain for the other tasks. When the communication with a data parallel task is implemented with a buffer with support for multiple producers and multiple consumers, all the tasks can use buffer space that is at that moment not used by other tasks. The same buffer space can thus be used by different tasks. Therefore, the required size of such a shared buffer is often smaller than the sum of the sizes of the separate buffers.

The type of buffer we use in this paper is a circular buffer that uses overlapping windows [22]. Each producer and each consumer has its own window in which it is allowed to write and read respectively. The buffer also allows for multiple producer windows and multiple consumer windows at once. Synchronization is decoupled from the actual read/write operations. Consumer tasks can immediately acquire a data location for reading as soon as the producer has written it. Consumer tasks thus synchronize by checking if the required location is already written. Furthermore, each consumer task moves the tail of their window such that locations that are not required anymore fall out of their window. Producer tasks synchronize by acquiring new space locations by moving the head of their write window and by signaling that a location is written. Because the buffer is circular, the movement of the windows is implemented in a cyclic manner. When the window reaches the end of the buffer, it wraps around and continues again at the beginning of the buffer. The administration of these wraparounds can be done by storing the wrap count for each task. The head of a write window is never allowed to overtake the tail of any of the read windows because that would mean that data can



(a) Initial state (b) After one execution of both τ_i as well as $\tau_{j,1}$ (c) After six executions of τ_i, three executions of $\tau_{j,1}$ and one execution of $\tau_{j,2}$

Figure 2. The state of the buffer of Figure 1c at different points in time. $\hat{w}$ is the head of the write window of task τ_i and $\check{r}_0$ and $\check{r}_1$ are the tails of the read windows of tasks $\tau_{j,0}$ and $\tau_{j,1}$ respectively.

be overwritten before it is read. Overtaking happens when the head of the write window moves to a further location in the buffer than the tail of a read window. Initially, the read window can acquire an amount of space equal to the buffer size. The head pointer thus starts one wrap around earlier than the read windows and overtakes the tail of a read window when it is more than buffer size locations ahead.

We use this buffer in such a way that it fits the data parallel implementation we propose. We make use of the fact that consumer tasks can skip locations, i.e., remove them from their read window without have to acquire them first. Consuming tasks thus never block on data that is not required by that task. The order in which the different replications of a task read/write locations in the buffer is fixed. The first replication always reads/writes the first locations in the buffer and then skips the locations that are read/written by the other replications. We therefore know that each location in the buffer is written by exactly one replication of a task and is also read by exactly one of the replications of a task.

We introduce an initial offset of the read/write windows to correspond with this fixed order of reads and writes. Each replication of a task τ_d is identified by a unique number n, $0 \leq n < \delta_d$, with δ_d the amount of replication of the task. Each of the replications of τ_d initially skip $n \cdot \gamma$ locations where γ is the number of read/written locations per execution of τ_d from/to a certain buffer. Each replication then tries to acquire the next γ locations in its window to read/write them. After this succeeds and the execution of the task is finished, the tail pointer corresponding to the replication is moved with $\delta_d \cdot \gamma$ locations. This thus skips $(\delta_d - 1) \cdot \gamma$ locations that are read/written by the other replications. The implementation of the buffer is such that all the $\delta_d \cdot \gamma$ locations are moved at once. The time it takes to move the window with n locations does not depend on how large n is. Therefore, the worst-case response time of the task does not depend on the amount of data parallelism of the task.

We illustrate the global idea of this implementation of the buffers with Example 1.

Example 1:
Consider the task graph of Figure 1c. The circular buffer in this task graph contains three locations and has one producing and two consuming tasks. The initial state of this buffer is illustrated in Figure 2a. The head pointer of task τ_i is denoted by $\hat{w}$. The tail pointers of tasks $\tau_{j,0}$ and $\tau_{j,1}$ are denoted by $\check{r}_0$ and $\check{r}_1$ respectively. The three cells in the circle correspond with the locations in the circular buffer. Initially all locations are empty. The head pointer of τ_i and the tail pointer of $\tau_{j,0}$ point to location 0. Task $\tau_{j,1}$ has initially skipped one location and its tail pointer therefore points to location 1.

Figure 2b shows the state of the buffer after one execution of both tasks τ_i and $\tau_{j,0}$ and no execution of $\tau_{j,1}$. Task τ_i has acquired and written location 0 which is then read by task $\tau_{j,0}$. After $\tau_{j,0}$ finishes its first execution, it moves its tail pointer with two locations to location 2. Location 0 is therefore not needed anymore by the consuming tasks and can thus be reused by task τ_i. This is illustrated in the figure by the dark (colored green) background of the cell. Figure 2c illustrates the buffer state after six executions of τ_i, three executions of $\tau_{j,0}$ and one execution of $\tau_{j,1}$. In this situation the buffer is completely full with data though location 1 (denoted by 4) is already read by task $\tau_{j,0}$ which is illustrated by the dark (colored green) background. The other full locations are illustrated by the light (colored blue) background of the cells. Task τ_i cannot make progress in this situation. Next to that, task $\tau_{j,0}$ has already used all of the data that belongs to it and can thus also not make progress. The only task that is enabled is task $\tau_{j,1}$ which can still execute twice in this situation.

This example also illustrates two additional properties of the proposed implementation. In Figure 2b, task $\tau_{j,0}$ has executed even when task $\tau_{j,1}$ is not yet enabled. Thus our approach does not have a blocking split behavior. Next to that, in Figure 2c task $\tau_{j,0}$ has already executed three times while task $\tau_{j,1}$ only executed once. This shows that task $\tau_{j,0}$ can progress before task $\tau_{j,1}$ has finished the same execution. Thus the proposed implementation does also not have blocking join behavior.

In the example it is also shown that the buffer location are reused by the different tasks. The third execution of task $\tau_{j,0}$ reads from location 1 in the buffer just like the first execution of $\tau_{j,1}$ does. Such a reuse of buffer location cannot be achieved when separate FIFO buffers are used to communicate with the replicated tasks. However, there is a situation in which separate FIFO buffers can be used. When the total buffer size is equal to an integer multiple of both the producer replication factor as well as the consumer replication factor, the total buffer size can be split fair into separate FIFO buffers. This can be achieved by increasing the buffer size to the smallest integer multiple of the least common multiple of the replication factors that is larger than the computed buffer size.

The idea of separate FIFO buffers is illustrated in Figure 3 in which the size of the buffer of Figure 1c is increased to four location. The expanded HSDF model of Figure 3 illustrates that in that case, two separate cycles can be distinguished. One cycle through actors $v_{i,0}$ and $v_{j,0}$ and a cycle through the other two actors. These separate cycle can be translated to separate first-in first-out (FIFO) buffers connecting tasks $\tau_{j,0}$ and $\tau_{j,1}$ to task τ_i of Figure 1c. Which buffer a task needs to write to or read from during an execution can be computed with a modulo operation which only adds a constant cost, i.e., not dependent on the amount of data parallelism.

V. Modeling Data Parallelism

In this section we describe the modeling of data parallel tasks. The first subsection discusses the SDF analysis model and its properties that are used in this paper. The second subsection presents the dataflow model of a data parallel task and the correspondence between buffers and the dataflow model. The third subsection shows that the dataflow model allows the same scheduling freedom as a data parallel task and in the fourth subsection we prove that such a data parallel task is modeled conservatively. These first four subsections assume

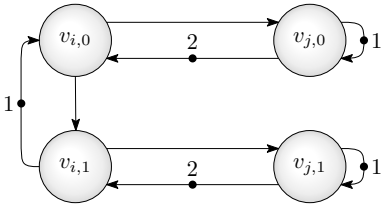


Figure 3. Situation in which the buffer of Figure 1c has a size of four location and can be split into separate FIFO buffers.

that the processor on which a task is executed is not shared. The last subsection combines data parallel tasks with run-time budget schedulers to illustrate that the method is more generally applicable.

A. Dataflow Model

The model we use to determine the required amount of data parallelism is the SDF [4] model. An SDF graph $G = (V, E, \delta, \rho, \pi, \gamma)$, is a directed graph that consists of a set of actors V and a set of directed edges E. Communication takes place by producing and consuming tokens over these edges. An edge $e_{ij} \in E$, directed from actor v_i to actor v_j represents an unbounded token queue and contains initially δ_{ij} tokens. An actor $v_j \in V$ is enabled to fire when on each input queue e_{ij} at least $\gamma(e_{ij})$ tokens are present, with $\gamma : E \to \mathbb{N}$. At the start of a firing of actor v_j, $\gamma(e_{ij})$ tokens are consumed from all input queues e_{ij} atomically. The firing finishes ρ_j time units after its start. At this finish, v_j atomically produces $\pi(e_{jk})$ tokens on each output queue e_{jk}.

Functionally deterministic dataflow graphs, such as SDF graphs, have a monotonic temporal behavior [25]. This has a number of important implications. For example, decreasing the firing duration of one actor in the graph can never lead to a later enabling of any of the actors in the graph during self-timed execution. During self-timed execution, actors fire as soon as they are enabled. Similarly, increasing the amount of initial tokens in the graph can never lead to a later enabling of any of the actors in the graph. See [25], [26] for more details on monotonicity of SDF graphs.

Another important property of SDF models is consistency. For an inconsistent SDF model, either any finite number of initial tokens will result in deadlock or an unbounded accumulation of tokens on an edge. Algorithms exist to verify consistency of connected SDF models [4]. In consistent SDF models, the average rate at which tokens are produced on an edge is equal to the average consumption rate on that edge. Therefore, a repetition vector $\mathbf{q}$ can be determined which contains the relative firing frequencies between the actors. We use q_i for the repetition factor of actor v_i where q_i is the ith component of $\mathbf{q}$. When every actor v_i in an SDF model fires exactly q_i times, it holds for each edge in the SDF model that the number of tokens produced on an edge is equal to the number of consumed tokens from that edge:

$$\forall_{e_{ij} \in E} \ : \ q_i \cdot \pi(e_{ij}) = q_j \cdot \gamma(e_{ij}) \tag{1}$$

B. Correspondence Between Task Graph and Dataflow Model

The task graph specifies the behavior of an application consisting of parallel tasks connected by buffers. The corresponding SDF model is used to determine temporal properties and suitable parameter values for the application. Figure 4 shows the correspondence relation between the SDF model

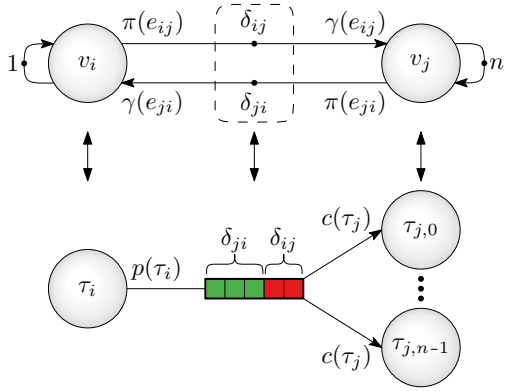


Figure 4. Relation between task graph including data parallelism (bottom) and the corresponding dataflow model (top).

and the tasks in such a task graph. Each task is modeled using one dataflow actor. The amount of data parallelism of a task is modeled using an edge from the corresponding actor back to itself. The number of initial tokens on this self-edge corresponds with the amount of data parallelism of that task. In Figure 4, task τ_j is replicated n times and is modeled using one SDF actor, v_j, with a self-edge containing n initial tokens.

Self-edges are also used to model stateful tasks. Such self-edge then contains a fixed number of initial tokens to denote that the task communicates with itself. The number of tokens on this self-edge reflects the execution to which the state is written. For example a task which writes state to the next execution of the task corresponds with an actor which has a self-edge with one token. The self-edge which models the state of a task forms an upperbound on the possible data parallelism of a task.

To illustrate the idea, we assume in this section that tasks are executed on a processor which is not shared between different tasks. The response time of a task, i.e., the maximum time a task takes to finish an execution after it is enabled, is thus less or equal to the Worst-Case Execution Time (WCET) of the task. When the firing duration of each actor is chosen to be equal to the WCET of the corresponding task, the dataflow model is conservative [25]. Tasks communicate via buffers and the synchronization with these buffers also takes time. We assume that this synchronization time is included in the WCET. In Section IV we have shown that the buffers can be implemented in such a way that the synchronization time is independent of the amount of data parallelism that is used. Therefore, the synchronization time can easily be included in the WCET of a task.

As discussed in Section IV tasks communicate via circular buffers that have support for multiple producers and multiple consumers [21], [22]. This enables data parallel tasks to share a buffer.

Tasks communicate as follows. A producing task first acquires empty locations from the buffer and after the task has filled them with data, the full locations are signaled as written. Consuming tasks use these full locations and release them as empty locations after the locations are not needed anymore. This behavior of a buffer is modeled in the dataflow model using two oppositely directed edges, e_{ij} and e_{ji} as shown in Figure 4. The number of initial tokens on edge e_{ij}, δ_{ij}, corresponds to the number of initially filled locations in the corresponding buffer. The buffer contains in total $\delta_{ij} + \delta_{ji}$

locations. Edge e_{ij} models the flow of full locations and edge e_{ji} the flow of empty locations. The skipped locations are not modeled because they do not add additional synchronization.

Tasks read/write during every execution the same amount of locations in a buffer. The number of locations that are read/write each execution corresponds with the number of tokens that are consumed from/produced to the corresponding queue in the dataflow model. Consider a buffer from task τ_i to task τ_j. Then we have $\gamma(e_{ji}) = \pi(e_{ij}) = p(\tau_i)$ with $p(\tau_i)$ the number of locations written by task τ_i in the considered buffer. And similarly, we have $\gamma(e_{ij}) = \pi(e_{ji}) = c(\tau_j)$ with $c(\tau_j)$ the number of locations that are read by task τ_j. This is also illustrated in Figure 4.

The order in which the different data parallel replications of a task read/write locations in the buffer is fixed. This allows us to define a mapping between the executions of the data parallel tasks and the corresponding actor firings such that the read/write locations correspond with the consumed/produced tokens. We use this mapping in the next sections to prove that the correspondence relation between task graph and dataflow model on the one hand preserves scheduling freedom and on the other hand is conservative.

Executions of tasks are mapped to firings of dataflow actors as follows. Consider a data parallel task τ_d with δ_d replications and actor v_d is the actor which corresponds to this task. We identify replication k of τ_d where $0 \leq k < \delta_d$. Execution n of replication k of τ_d then corresponds with firing $k + n \cdot \delta_d$ of actor v_d.

C. Accurate Modeling

In this section we show that the introduced modeling of data parallelism by a dataflow actor allows the same scheduling freedom as the corresponding data parallel task.

Without loss of generality we assume for the proof that there is only one task which is data parallelized. We call the actor corresponding to this task, actor v_d. The amount of data parallelism of this task is equal to δ_d which means that the self-edge dependencies of the corresponding dataflow actor should allow δ_d simultaneous firings.

To show this we use the expansion of an SDF graph to an equivalent HSDF graph [4], [3]. An HSDF graph is an SDF graph in which all the numbers of consumed and produced tokens per firing, are equal to 1. Note that we use this expansion to an equivalent HSDF graph merely to prove the scheduling freedom. We do not use this HSDF graph itself to model the behavior of the task graph. We use a modified expansion method to make the data parallelism visible.

The expansion of an SDF graph to an equivalent HSDF graph makes use of the balance equation from (1). Each SDF actor v_i is modeled using q_i HSDF actors. Each firing of an HSDF actor corresponds to a firing of the SDF actor. After every HSDF actor has fired exactly once, the tokens are again on the same edge as in the initial situation. This corresponds to q_i firings of SDF actor v_i.

The expansion to HSDF translates the edges of the SDF model to multiple edges in the HSDF model based on the communication between firings of the actors in the SDF model. Also self-edges are translated to multiple edges in the HSDF model. A self-edge for actor v_d with δ_d tokens specifies in an SDF model that firing n of v_d can only start when firing $n-\delta_d$ of v_d is finished. This is modeled in the HSDF model with an edge between the actors modeling firing n and firing $n - \delta_d$.

To show the scheduling freedom of the dataflow model we use an expansion of an SDF model to an equivalent HSDF model which differs from the expansion method in [4], [3]. Equation (1) can be rewritten to include the amount of data parallelism of actor v_d: $\forall_{e_{ij} \in E} \ : \ q_i \cdot \delta_d \cdot \pi(e_{ij}) = q_j \cdot \delta_d \cdot \gamma(e_{ij})$. Instead of expanding each SDF actor v_i to q_i HSDF actors, we can also model an SDF actor v_i with $q_i \cdot \delta_d$ HSDF actors. Actor v_d is then modeled using $q_d \cdot \delta_d$ HSDF actors. We call this new HSDF model the expanded dataflow model.

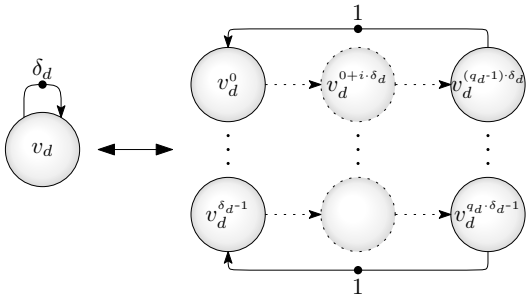


Figure 5. Expansion of an SDF actor modeling data parallelism (left) to HSDF actors (right).

Figure 5 shows how an SDF actor v_d with a self-edge containing δ_d initial tokens is modeled using $q_d \cdot \delta_d$ HSDF actors. Because we model v_d with $q_d \cdot \delta_d$ HSDF actors, the self-edge dependencies in the HSDF graph have the illustrated structure. As is shown in Figure 5 we can distinguish δ_d groups of q_d HSDF actors. The different groups in Figure 5 start with firing the actors denoted by v_d^0 to $v_d^{\delta_d-1}$. No self-edge dependencies exist between these δ_d groups, only inside the group itself.

Each of these groups models exactly one of the data parallel replications of the task. Group k, with $0 \leq k < \delta_d$ models the firings $i = k + n \cdot \delta_d, \ n \in \mathbb{N}$ of the SDF actor. These firings correspond, as defined in Section V-B, to execution n of replication k of the corresponding data parallel task.

Because there are no self-edge dependencies between the different groups of HSDF actors, there is also no dependency in the schedule between the groups when the self-edge dependencies are considered. Each of the δ_d groups of HSDF actors can thus be scheduled individually which means that the dataflow model allows the same scheduling freedom as the data parallel task.

D. Proof of Conservativeness

This section contains the proof that the dataflow model for data parallel tasks as proposed in this paper is conservative. We prove that the production times of tokens in the dataflow model form an upperbound on the times on which the tasks write to the corresponding buffer locations.

In [25] sufficient conditions are derived under which dataflow models are temporally conservative to task graphs. One condition is that the number of consumed and produced tokens on each edge in the dataflow model needs to be equal to the number of read and written locations of the corresponding connection in the task graph. This is ensured by the correspondence between dataflow model and task graph presented in Section V-B. Given this correspondence it is shown in [25] that when the following equation holds for every task and corresponding actor, the whole dataflow model is conservative to the task graph.

$$\forall_i : \ e(i) \leq \hat{e}(i) \implies \forall_i : \ f(i) \leq \hat{f}(i) \qquad (2)$$

We use $e(i)$ for the external enabling time of execution i of a task, i.e., the time at which the task can read sufficient locations from the adjacent buffers. The external enabling time of the corresponding actor firing is denoted by $\hat{e}(i)$. This is the earliest time at which the required tokens are present on the incoming queues. Furthermore, $f(i)$ is the finish time of execution i of the task and $\hat{f}(i)$ is the finish time of firing i of the corresponding actor.

The intuition behind Equation (2) is that the arrival times of tokens form an upperbound on the times at which buffer locations are filled with data. Monotonicity of the temporal behavior of functional deterministic dataflow graphs [25] guarantees that then also the enabling times of subsequent firings of actors are never earlier than the corresponding external enabling times of the tasks. Therefore, when the local condition of Equation (2) holds for every task and corresponding actor, all the production times of tokens form upperbounds on the times at which buffer locations are filled. This means that the complete dataflow model is temporally conservative to the task graph.

Equation (2) is only applicable when one actor corresponds with one task whereas we address the case in which an actor corresponds to multiple replications of the same task. Therefore, we extend Equation (2) to data parallel tasks and show that it holds for the dataflow analysis method as proposed in this paper.

We use the mapping between task executions and dataflow firings presented in Section V-B to derive Equation (3). Execution n of replication k of a task τ_d corresponds with firing $k+n\cdot\delta_d$ of the corresponding dataflow actor v_d. We use $e^k(n)$ for the external enabling time of execution n of replication k of task τ_d and $f^k(n)$ for the finish time of execution n of replication k.

$$\begin{aligned}&\forall_{0\leq k<\delta_d}:\\ &\forall_n:\ e^k(n)\leq\hat{e}(k+n\cdot\delta_d)\implies\forall_n:\ f^k(n)\leq\hat{f}(k+n\cdot\delta_d)\end{aligned}\tag{3}$$

We now prove this relation for the proposed dataflow model of data parallel tasks. By using max-plus algebra semantics [27] of dataflow actors on actor v_d we obtain:

$$\hat{f}(i)\geq\max\left(\hat{e}(i),\hat{f}(i-\delta_d)\right)+\rho_d\tag{4}$$

The firing duration, ρ_d, of actor v_d is equal to the WCET of the corresponding task τ_d. Next to that, different executions of replication k of task τ_d cannot overlap. It therefore holds that:

$$f^k(n)\leq\max\left(e^k(n),f^k(n-1)\right)+\rho_d\tag{5}$$

We now prove Equation (3) by using induction on execution number n. We make use of the left part of Equation (3), i.e., the enabling times of the dataflow actor are conservative to the enabling times of the corresponding task execution, to prove the right part of Equation (3). We can safely assume that the finish times of executions numbers smaller than 0 are equal to $-\infty$ to denote that they do not exist. With this assumption the base case of the induction proof, $n=0$, can be proven:

$$\begin{aligned}f^k(0)&\leq\max\left(e^k(0),f^k(-1)\right)+\rho_d\\&\leq\max\left(e^k(0),-\infty\right)+\rho_d\\&\leq e^k(0)+\rho_d\\&\leq\hat{e}(k)+\rho_d\\&\leq\hat{f}(k)\end{aligned}\tag{6}$$

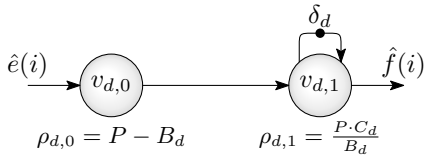


Figure 6. Dataflow component modeling the temporal behavior of a task scheduled with a run-time budget scheduler.

For the induction step we prove that when Equation (3) holds for execution $n-1$ of replication k, it also holds for execution n of replication k. When Equation (3) holds for execution $n-1$ we have: $f^k(n-1)\leq\hat{f}(k+(n-1)\cdot\delta_d)$. With $e^k(n)\leq\hat{e}(k+n\cdot\delta_d)$ we have:

$$\begin{aligned}f^k(n)&\leq\max\left(e^k(n),f^k(n-1)\right)+\rho_d\\&\leq\max\left(\hat{e}(k+n\cdot\delta_d),\hat{f}(k+n\cdot\delta_d-\delta_d)\right)+\rho_d\\&\leq\hat{f}(k+n\cdot\delta_d)\end{aligned}\tag{7}$$

This proves that Equation (3) holds for every execution n which means that data parallel tasks are conservatively modeled by the proposed dataflow model.

E. Combination with Run-Time Schedulers

The previous sections assumed that processors are not shared between tasks. In this section we show that the approach is also applicable when tasks do share processors. We do this by showing that the analysis method introduced in this paper is also applicable for latency-rate actor components which are used to model the temporal behavior of tasks scheduled with run-time budget schedulers [25]. Such latency-rate actor components are more widely applicable and can for example also be used to model variations in execution times [28].

Run-time budget schedulers guarantee every task a minimum time budget B in every time interval of length P. The temporal behavior of a task can be conservatively analyzed using a latency-rate actor component. Such an actor component is shown in Figure 6 for a data parallel task τ_d. Each of the replications of τ_d have the same time budget B_d. In this Figure, C_d is used for the WCET of task τ_d and δ_d corresponds to the amount of data parallelism of τ_d.

It can be shown by using the same method as in Subsection V-C that the dataflow component from Figure 6 models the amount of data parallelism accurately. We therefore only provide the proof that the actor component is temporally conservative to the corresponding task.

The finish time of a task τ_d, scheduled with a run-time budget scheduler can be bounded as follows [25]:

$$f(i)\leq\max\left(e(i)+P-B_d,f(i-1)\right)+\frac{P\cdot C}{B_d}\tag{8}$$

Each replication of a task has its own budget. Therefore, this finish time can also be used for a replication k of a task:

$$f^k(n)\leq\max\left(e^k(n)+P-B_d,f^k(n-1)\right)+\frac{P\cdot C}{B_d}\tag{9}$$

Using the max-plus semantics of the dataflow component of Figure 6 we obtain:

$$\hat{f}(i)\geq\max\left(\hat{e}(i)+P-B_d,\hat{f}(i-\delta_d)\right)+\frac{P\cdot C}{B_d}\tag{10}$$

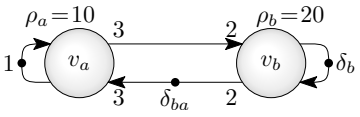


Figure 7. Example of a trade-off between data and pipeline parallelism.

δ_{ba}	δ_b	μ
4	1	80
5	1	70
6	1	60
4	2	60
6	2	40
7	2	35
8	2	30
7	3	30
10	3	20

Table I. Results for Figure 7 illustrating the average period for different token assignment.

By using Equations (9) and (10) we can prove Equation (3) by induction on execution number n. This proof is omitted because it is analogous to the induction proof of Section V-D.

VI. Determining Data and Pipeline Parallelism

With the introduced modeling of data parallelism by means of one actor with a self-edge containing multiple tokens, we can also derive the required amount of data parallelism. In this section we present a method for this derivation.

Data parallelism of a task is modeled as a cyclic dependency on the corresponding SDF actor itself. Because buffers are also modeled as cyclic dependencies, determining data parallelism is strongly related to analysis methods which determine the required buffer sizes given a throughput constraint. When these buffer sizing methods are instrumented correctly, they can also be used to determine data parallelism.

Buffer sizing methods such as [12], [23], [15] use graph algorithms to determine the minimum required amount of tokens that allow the required throughput. Also the costs of tokens can be taken into account when determining this minimum amount of tokens. The methods make use of the fact that increasing the amount of tokens on edges can make tokens on other edges redundant. The costs of tokens can be used to resolve the trade-off between the amounts of tokens on the different edges. These costs are thus used to express a preference for which edges should contain the least amount of tokens.

The dataflow schedules that are used in these buffer sizing methods inherently exploit pipeline parallelism. Given the data dependencies as specified in the dataflow model it is explicit which firings of actors may overlap. The amount of tokens in the cycles form an upperbound on the amount of pipeline parallelism that can be used.

The buffer sizing methods can also be used to determine the required amount of data parallelism. The number of tokens can be determined for the self-edge cycle modeling the data parallelism such that the temporal constraints are met. This allows to determine the required amount of data parallelism simultaneously with the amount of pipeline parallelism and the corresponding buffer sizes. This makes the trade-off between data and pipeline parallelism explicit. By using suitable token costs, the choice can be made between increasing buffer sizes or increasing the amount of data parallelism. Also it can be determined how large the buffers need to be to exploit a certain degree of data parallelism.

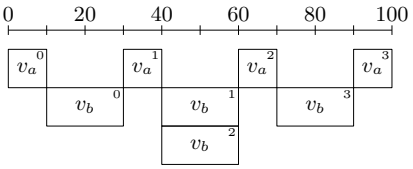
Example 2:
Figure 7 shows an example demonstrating the trade-off between data and pipeline parallelism. It presents the dataflow model corresponding to two tasks, one producer and one consumer. The producer, modeled with actor v_a, has a WCET of 10 time units and is a stateful task which cannot be data parallelized. Each execution, three locations are filled with data by this task. The consumer task, modeled with actor v_b, reads two locations per execution. This task has a WCET of 20 time units and can be data parallelized. A choice can be made between increasing the buffer connecting the two tasks (increasing δ_{ba}) and allowing more pipeline parallelism or increasing the amount of data parallelism of the consuming task (increasing δ_b). This illustrates the trade-off between data and pipeline parallelism.

The results of exploiting this trade-off are shown in Table I. The first two columns show the different assignments of initial tokens to edge e_{ba} and the self-edge of v_b. The third column, denoted by μ, shows the average period that can be achieved given the token assignment. It corresponds to the average period in which each SDF actor v_i fires q_i times. For example, for the first row μ is equal to 80 time units which means that on average, v_a can fire twice per 80 time units and v_b on average can fire three times per 80 time units. This period is the inverse of the throughput of the application and is calculated using a maximum cycle ratio algorithm [29]. Only the minimum numbers tokens are illustrated per average period. The maximum cycle ratio algorithm is exact and also the exhaustive search is complete which implies that we find the exact numbers of required tokens for each average period.

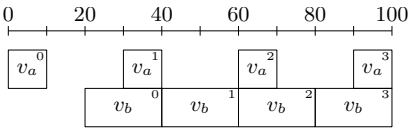
As is shown in Table I, the average period decreases when the amount of tokens increases. The lowest average period that can be reached by the task graph is 20 time units. For this period, actor v_a limits the throughput due to its self-edge. Table I also shows the trade-off between increasing the buffer size (increasing δ_{ba}) and increasing the amount of data parallelism (increasing δ_b). For example, the third and the fourth row in the table result in an equal μ. This means that the throughput that is achieved when the size of the buffer is equal to six locations and no data parallelism is used, is equal to the throughput that is achieved when the buffer size is equal to four locations and the consuming tasks is replicated twice.

As discussed, buffer sizing methods can be used to determine the required amount of data parallelism next to the required buffer sizes. Exact buffer sizing methods such as [12], [14] have a non-polynomial time-complexity which can be problematic for larger dataflow graphs. Buffer sizing methods exist with a polynomial time-complexity [23], [15] and are based on linearizations of schedules. The schedules of the dataflow actors are approximated with linear equations which actually model a strictly periodic token production and consumption. Such linear equations can be used to solve the buffer sizing problem efficiently, but can result in non-optimal token placements. Because of the strictly periodic token production, the trade-off between data and pipeline parallelism cannot be exploited as good as with the exact buffer sizing methods.

This is illustrated in Figure 8 which shows two different schedules with an average period of 60 time units for the dataflow model of Figure 7. Figure 8a shows a schedule that exploits data parallelism. Edge e_{ba} contains four tokens and δ_b is equal to two. The horizontal axis denotes time and the rectangles in the figure represent the different firings of actors v_a and v_b. The number in the top right corner of each rectangle indicates the number of each firing. The schedule of Figure 8a exploits data parallelism and is not strictly periodic. Figure 8b shows the only possible strictly periodic schedule with an average period of 60 time units. The required number of tokens



(a) Schedule exploiting data parallelism with $\delta_{ba} = 4$ and $\delta_b = 2$



(b) Strict periodic schedule with $\delta_{ba} = 6$ and $\delta_b = 1$

Figure 8. Two different schedules for Figure 7 with an average period of 60 time units.

δ_{SRC}	1
δ_{FILTER}	1
δ_{FFT}	3
δ_{EQ}	1
δ_{DEMAP}	1
δ_{DEINT}	1
δ_{VIT}	1
δ_{RE-ENC}	1
δ_{CH-EST}	2

$\delta_{FILTER,SRC}$	1
$\delta_{FFT,FILTER}$	3
$\delta_{EQ,FFT}$	4
$\delta_{CH-EST,FFT}$	6
$\delta_{DEMAP,EQ}$	1
$\delta_{DEINT,DEMAP}$	1
$\delta_{VIT,DEINT}$	2
$\delta_{RE-ENC,VIT}$	1
$\delta_{CH-EST,RE-ENC}$	2
$\delta_{EQ,CH-EST}$	0

Table II. Result of the analysis method applied on the dataflow model of Figure 9.

for this schedule is six for edge e_{ba} and δ_b is equal to one. The trade-off as shown in Table I for an average period of 60 time units can thus not be made when periodic schedules are considered.

VII. Case-Study

In this section we show the applicability of the analysis method by applying it to a WLAN 802.11p transceiver [30]. The application is executed on an MPSoC and data parallelism as well as pipeline parallelism is required to meet the throughput constraint of the application. We will consider the part of the WLAN 802.11p transceiver which is active during packet decoding mode.

Figure 9 shows the SDF model corresponding to this packet decoding mode. The numbers of tokens that are produced and consumed are only drawn when they are unequal to one. The throughput constraint is imposed by the periodic source (SRC) which produces one sample every $8\mu s$ ($= \frac{1}{125kHz}$). The samples that are produced by the source are first processed by a filter and then by an FFT. The packets are then subsequently decoded using the channel equalizer (EQ), demapper (DEMAP), deinterleaver (DEINT) and the viterbi decoding (VIT) tasks. The packets are encoded for error correction with a rate of $\frac{1}{2}$ which means that each transmitted m bits are decoded to $\frac{m}{2}$ error corrected bits. The viterbi task decodes these bits and therefore has an input of 2 samples and outputs only 1 sample.

The result of the channel equalizer is improved by using a channel estimator (CH-EST). The output of the viterbi task contains the error corrected bits. These corrected bits are re-encoded (RE-ENC) and compared with the output of the FFT. Based on the difference between the bits, an estimation of the channel is computed by the channel estimation task. The estimation that is computed using sample i is used as the setting of the channel equalizer during the reception of sample $i+2$. This is modeled using two initial tokens on the channel estimation cycle.

Figure 9 only shows the edges on which the tokens cannot be chosen freely. The other edges are omitted for clarity. The behavior of the viterbi and re-encoding tasks depends on the result of the previous executions plus the current sample. They thus do have state which is modeled using self-edges with one token. On the other edges, the number of tokens that are required to meet the throughput, are computed using the analysis method described in Section VI. The result of this analysis is shown in Table II. It contains the required number of tokens for the omitted edges. Two tasks are replicated, the FFT task (replicated three times) and the channel estimation task (replicated two times). The results are determined with the same exhaustive search approach as discussed in Example 2.

The FFT task has a high execution time and therefore needs to be replicated three times to meet the throughput constraint. To exploit this introduced data parallelism, the sizes of the buffers connecting the FFT task need to be sized correctly. The sizes are 3, 4 and 6 locations for the buffer with the FFT, the equalizer and the channel estimator, respectively.

Due to the fixed number of two tokens in the channel estimation loop, the amount of pipeline parallelism that can be used in this loop does not suffice to meet the throughput constraint. However, the analysis method has determined that when the channel estimation task is replicated twice, the throughput can be met.

VIII. Conclusion

The analysis method introduced in this paper allows to model data parallelism accurately without having to replicate dataflow actors. Instead, the data parallelism is modeled using a cyclic self dependency with the number of tokens on this cycle corresponding to the amount of data parallelism. We have shown that the parameterized dataflow model is conservative to the corresponding task graph by making use of a new SDF to HSDF conversion.

It is shown that with the introduced dataflow model, the required buffer sizes and the amount of data and pipeline parallelism can be determined simultaneously, which makes the trade-off between them explicit. Furthermore, it has been shown that existing buffer sizing methods can be used to choose between increasing buffer sizes or increasing data parallelism in order to meet the throughput constraint.

It is also shown that task graphs containing data parallelism can be implemented with circular buffers with support for multiple producers and multiple consumers. These buffers can be implemented such that the synchronization costs are not dependent on the amount of data parallelism that is used. It is shown that this property enables the proposed analysis model combining data and pipeline parallelism. Furthermore, we have shown that if FIFO buffers are applied instead of circular buffers, the parameterized dataflow model is only conservative if the computed number of tokens is an integer multiple of the amount of replication of the tasks.

The applicability of the analysis method is demonstrated with a WLAN 802.11p application. It is shown that despite the cyclic re-encoding loop of this application, the amount of data parallelism can be derived simultaneously with the amount of

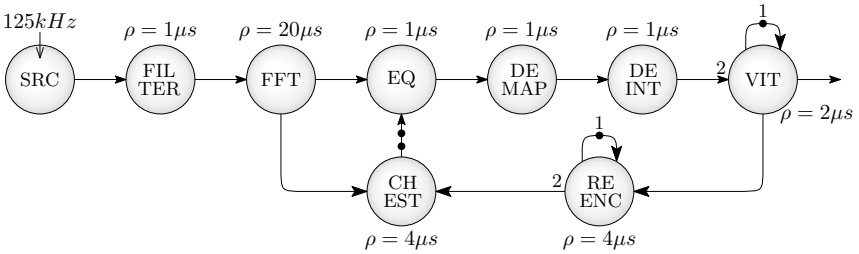


Figure 9. Dataflow model of the packet decoding mode of a WLAN 802.11p transceiver.

pipeline parallelism such that the throughput requirements of the application can be met. Also the required buffer sizes have been determined by using the proposed analysis method. The proposed analysis method can thus be used to make the trade-off between data and pipeline parallelism.

REFERENCES

[1] M. Gordon, W. Thies, and S. Amarasinghe, "Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs," in *Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2006, pp. 151–162.

[2] E. Lee and T. Parks, "Dataflow Process Networks," *Proc. of the IEEE*, vol. 83, no. 5, pp. 773–801, 1995.

[3] S. Sriram and S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., 2000.

[4] E. Lee and D. Messerschmitt, "Synchronous Data Flow," *Proc. of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[5] B. Theelen *et al.*, "A Scenario-Aware Data Flow Model for Combined Long-Run Average and Worst-Case Performance Analysis," in *Proc. of the Int'l Conf. on Formal Methods and Models for Co-Design (MEMOCODE)*. IEEE, 2006, pp. 185–194.

[6] M. Wiggers, M. Bekooij, and G. Smit, "Buffer Capacity Computation for Throughput-Constrained Modal Task Graphs," *ACM Trans. on Embedded Computing Systems (TECS)*, vol. 10, no. 2, p. 17, 2010.

[7] S. Geuns, J. Hausmans, and M. Bekooij, "Automatic Dataflow Model Extraction from Modal Real-Time Stream Processing Applications," in *Proc. of the Conf. on Languages, Compilers and Tools for Embedded Systems (LCTES)*. ACM, 2013, pp. 143–152.

[8] A. Ghamarian *et al.*, "Throughput Analysis of Synchronous Data Flow Graphs," in *Proc of the Int'l Conf. on Application of Concurrency to System Design (ACSD)*. IEEE, 2006, pp. 25–36.

[9] B. Bodin, A. Munier-Kordon, and B. De Dinechin, "K-Periodic Schedules for Evaluating the Maximum Throughput of a Synchronous Dataflow Graph," in *Proc. of the Int'l Conf. on Systems, Architectures, Modeling and Simulation (SAMOS)*. IEEE, 2012, pp. 152–159.

[10] J. Hausmans, M. Bekooij, and H. Corporaal, "Resynchronization of Cyclo-Static Dataflow Graphs," in *Design, Automation and Test in Europe (DATE)*, 2011.

[11] M. Kudlur and S. Mahlke, "Orchestrating the Execution of Stream Programs on Multicore Platforms," in *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*. ACM, 2008, pp. 114–124.

[12] S. Stuijk, M. Geilen, and T. Basten, "Exploring Trade-offs in Buffer Requirements and Throughput Constraints for Synchronous Dataflow Graphs," in *Proc. of the Design Automation Conf. (DAC)*. ACM, 2006, pp. 899–904.

[13] M. Wiggers, M. Bekooij, and G. Smit, "Computation of Buffer Capacities for Throughput Constrained and Data Dependent Inter-Task Communication," in *Design, Automation and Test in Europe (DATE)*. IEEE, 2008, pp. 640–645.

[14] O. Moreira *et al.*, "Buffer Sizing for Rate-Optimal Single-Rate Data-Flow Scheduling Revisited," *IEEE Transactions on Computers*, vol. 59, no. 2, pp. 188–201, 2010.

[15] R. de Groote *et al.*, "Back to Basics: Homogeneous Representations of Multi-Rate Synchronous Dataflow Graphs," in *Proc. of the Int'l Conf. on Formal Methods and Models for Co-Design (MEMOCODE)*. IEEE, 2013, pp. 35–46.

[16] T. Stefanov, B. Kienhuis, and E. Deprettere, "Algorithmic Transformation Techniques for Efficient Exploration of Alternative Application Instances," in *Proc. of the Int'l Conf. on Hardware/software Codesign and System Synthesis (CODES+ISSS)*. ACM, 2002, pp. 7–12.

[17] S. Meijer, H. Nikolov, and T. Stefanov, "Combining Process Splitting and Merging Transformations for Polyhedral Process Networks," in *IEEE Symp. on Embedded Systems for Real-Time Multimedia (ESTIMedia)*. IEEE, 2010, pp. 97–106.

[18] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A Language for Streaming Applications," in *Proc. of the Int'l Conf. on Compiler Construction (CC)*. Springer, 2002, pp. 179–196.

[19] L. Schor *et al.*, "Expandable Process Networks to Efficiently Specify and Explore Task, Data, and Pipeline Parallelism," in *Proc. of the Int'l Conf. on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. IEEE, 2013.

[20] D. Bui and E. Lee, "StreaMorph: A Case for Synthesizing Energy-efficient Adaptive Programs Using High-level Abstractions," in *Proc. of the Int'l Conf. on Embedded Software (EMSOFT)*. IEEE, 2013.

[21] K. Denolf *et al.*, "Exploiting the Expressiveness of Cyclo-Static Dataflow to Model Multimedia Implementations," *EURASIP Journal on Advances in Signal Processing*, 2007.

[22] T. Bijlsma, M. Bekooij, and G. Smit, "Circular Buffers with Multiple Overlapping Windows for Cyclic Task Graphs," in *Trans. on High-Performance Embedded Architectures and Compilers (HiPEAC): Volume 5, Issue 3*, 2011.

[23] M. Wiggers *et al.*, "Efficient Computation of Buffer Capacities for Multi-Rate Real-Time Systems with Back-Pressure," in *Proc. of the Int'l Conf. on Hardware/software Codesign and System Synthesis (CODES+ISSS)*. ACM, 2006, pp. 10–15.

[24] J. Hausmans *et al.*, "Compositional Temporal Analysis Model for Incremental Hard Real-Time System Design," in *Proc. of the Int'l Conf. on Embedded Software (EMSOFT)*. ACM, 2012, pp. 185–194.

[25] M. Wiggers, M. Bekooij, and G. Smit, "Monotonicity and Run-Time Scheduling," in *Proc. of the Int'l Conf. on Embedded Software (EMSOFT)*. ACM, 2009, pp. 177–186.

[26] M. Geilen, S. Tripakis, and M. Wiggers, "The Earlier the Better: A Theory of Timed Actor Interfaces," in *Proc. of the Int'l Conf. on Hybrid Systems: Computation and Control (HSCC)*, April 2011.

[27] F. Baccelli *et al.*, *Synchronization and Linearity: an Algebra for Discrete Event Systems*. Wiley, 1992.

[28] J. Hausmans *et al.*, "Two Parameter Workload Characterization for Improved Dataflow Analysis Accuracy," in *Real-Time and Embedded Technology and Applications Symp. (RTAS)*. IEEE, 2013, pp. 117–126.

[29] A. Dasdan, "Experimental Analysis of the Fastest Optimum Cycle Ratio and Mean Algorithms," *Trans. on Design Automation of Electronic Systems (TODAES)*, vol. 9, no. 4, pp. 385–418, 2004.

[30] P. Alexander, D. Haley, and A. Grant, "Outdoor Mobile Broadband Access with 802.11," *Communications Magazine*, vol. 45, no. 11, pp. 108–114, 2007.