Resynchronization of Cyclo-Static Dataflow Graphs

Joost P.H.M. Hausmans
Eindhoven University of Technology
joost.hausmans@gmail.com

Marco J.G. Bekooij
NXP Semiconductors, Eindhoven
marco.bekooij@nxp.com

Henk Corporaal
Eindhoven University of Technology
h.corporaal@tue.nl

***Abstract*—Parallel stream processing applications are often executed on shared-memory multiprocessor systems. Synchronization between tasks is needed to guarantee correct functional behavior. An increase in the communication granularity of the tasks in the parallel application can decrease the synchronization overhead. However using coarser-grained synchronization can result in deadlock or violation of the throughput constraint for the application in case of cyclic data dependencies. Resynchronization tries to change the synchronization behavior in order to reduce the synchronization overhead. Determining the amount of resynchronization while preventing deadlock and satisfying the throughput constraint of the application, forms a global analysis problem. In this paper we present a Linear Programming (LP) algorithm for minimizing synchronization by means of resynchronization that is based on the properties of dataflow models. We demonstrate our approach with an extended Constant Modulus Algorithm (CMA) in a beam-forming application. For this application we reduce the number of synchronization statements with 30% while having a memory constraint of 200 tokens. The algorithm which calculates this reduction takes less than 20 milliseconds for this problem instance.**

I. Introduction

Parallelized applications typically require synchronization for correct behavior. Fine-grained communication and synchronization can help to prevent deadlock. However the overhead caused by fine-grained synchronization can become significant. By increasing the synchronization granularity, the synchronization overhead can be reduced but this often comes at the cost of an increase in the size of the communication buffer. Resynchronization determines the maximum synchronization granularity that still prevents deadlock and respects throughput and memory size constraints. However determining resynchronization is a global optimization problem with interesting trade-offs.

In the past it is shown that the amount of runtime overhead required for synchronization can be reduced significantly by removing redundant synchronization in statically scheduled systems [1], [2] that are modeled with Synchronous Dataflow (SDF) graphs. Methods are formulated to resynchronize these type of systems [3], [4]. These resynchronization methods add synchronization statements to make other synchronization statements redundant. However, these methods can only handle statically scheduled systems while this work also supports run-time scheduled systems.

In this work we model parallel stream processing applications with the Cyclo-Static Dataflow (CSDF) model [5]. This CSDF model is used to compute sufficient buffer capacities given a throughput constraint. Because of the one-to-one relation between an application and its CSDF synchronization model, changes in the CSDF model can be translated back to the synchronization of the application.

This paper presents a method for the resynchronization of CSDF graphs. We extend the method described in [6] with the calculation of a maximum resynchronization while having a throughput and memory size constraint where maximum resynchronization is defined as the maximum number of removed synchronization statements. We show that this method can be applied to applications which have function-parallelism and may have dynamic schedules (run-time scheduling). We show that the modification of the CSDF graph can be used to modify the synchronization behavior in the task-graph of the corresponding application.

978-3-9810801-7-9/DATE11/©2011 EDAA

The outline of this paper is as follows. Related work is discussed in Section II. In Section III we describe the basic idea behind the presented resynchronization approach. In Section IV we recapitulate the CSDF model and its properties on which our approach is based. Our resynchronization approach is described in Section V. Results of the evaluation of our approach can be found in Section VI. This case-study also shows that there exists a trade-off between memory size and synchronization overhead. In Section VII we conclude and indicate future work.

II. Related Work

Reduction of the synchronization overhead has been addressed in [2] in which it is shown that removing redundant synchronization in static order scheduled systems can reduce the synchronization overhead. In such a system, synchronization messages can be redundant because there exists a fixed order between the execution of tasks on the same processor. This observation of redundant synchronization messages can also be used for the resynchronization of applications on these systems. Adding synchronization statements can make other synchronization statements redundant. If the number of synchronization statements that become redundant is higher than the number of added statements, the resynchronization is effective and a reduction of the synchronization overhead is achieved. Algorithms that perform this type of resynchronization are presented in [3]. In [4] the same technique is exploited but with an additional latency constraint. These two techniques and their relation is developed further in [7].

For data parallel applications it is important to calculate a suitable partitioning such that the different parts of the data sets can be processed by different processors. In a suitable partitioning, communication often can be moved outside a loop and the number of the communicated messages is decreased. This is called message vectorization [8]. When synchronization is required for each message, the reduction of the number of messages also decreases the synchronization overhead. Loop coalescing methods are presented in [9] and [10]. Furthermore in data parallel applications the number of communication messages can be decreased by moving messages to different positions in the program such that other messages become redundant. This optimization is addressed in [11].

Compared to the above mentioned techniques, our technique is able to perform resynchronization on CSDF graphs in which it is possible to model applications with run-time scheduling [12] and functional parallelism on which iterations may overlap.

This work extends the work on dataflow analysis for buffer minimization in run-time scheduled systems as described in [6]. It relies on the same linearization techniques.

III. Basic Idea

The idea of increasing the synchronization granularity is illustrated with the task graph shown in Figure 1. This task graph is chosen as the simplest example to illustrate the basic concepts. Each task in the task graph consists of a code fragment. When each task is executed in parallel using run-time scheduling, synchronization must be added to guarantee functional correctness. Figures 2a and 2b show the code belonging to the tasks with added synchronization statements. Task t_1 first acquires some space to write in, then performs the actual computation, writes the result in the buffer and eventually releases the written place in the communication buffer c_x such that the result can be used by a consuming task. Task t_2 first acquires the required data, uses it and then releases the space again.

The synchronization behavior of this task graph is modeled as a CSDF graph which is shown in Figure 3. The edges in this CSDF graph with the same direction as the edges in the task graph, model the synchronization of full places in the buffer while the opposite edges model the space in the buffer. The required capacity of the buffer in the task graph is therefore equal to the sum of the number of initial tokens on these two edges. With the CSDF graph the capacity of buffer c_x is determined to prevent deadlock or to guarantee a certain throughput constraint. The phases in the CSDF graph correspond with executions of the acquire and release functions. In the CSDF graph, $\langle 4 \times 1 \rangle$ is a shorthand for $\langle 1, 1, 1, 1 \rangle$, 4 phases with a production, consumption or firing duration of one.

While the productions and consumptions in the CSDF graph correspond with the synchronization functions in the task graph, a modification of these productions or consumptions can be used to modify the task graph. For example a modification of a consumption from one tokens to two, can be used to modify the corresponding acquire function to acquire two places instead of one. The idea is to calculate a possible modification of the CSDF graph which can be used to modify the task graph. Figure 4 shows the result of a possible resynchronization of the CSDF graph. The second and the fourth consumption on edge (v_1, v_2) are performed earlier. This results in a larger number of required initial tokens. The result of the resynchronization can now be used to modify the task graph. A possible result of this modification is shown in the Figure 2c. While the number of required initial tokens corresponds the required capacity of the buffer in the task graph, this buffer capacity also needs to be increased to two buffer locations. This already shows that resynchronization comes at the cost of an increased buffer capacity.

The buffer capacity determination algorithm as presented in [6] is based on transforming the problem conservatively to a linear problem. Linear bounds on the cumulative token transfer are determined such that a schedule can be defined in which

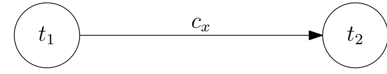


Figure 1. Example task graph

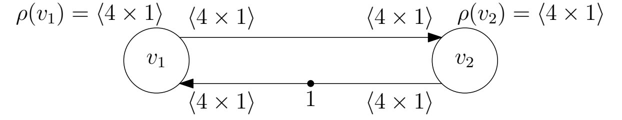


Figure 3. Synchronization behavior graph of the code in Figure 2

all tokens consumptions and productions are bounded by the linear bounds.

In the determined schedule for actor v_2 on edge (v_1, v_2), all token consumptions can be bounded by a linear upper bound such that all token consumptions are below this bound. A possible schedule for actor v_2 of Figure 3, which is bounded with a linear bound, $\hat{c}_{12}$, can be found in Figure 5a. In Figure 5b it is shown that an advancement of the second and fourth consumption to an earlier time in the schedule can be translated to a shift in the linear bound. If it was possible to shift the linear bound upwards with the amount needed for this advancement without violating the throughput and memory size constraints, the transformation of the CSDF graph as shown in Figure 4 is possible.

The algorithm presented in the next section, expresses the resynchronization as a linear shift of the linear bounds. In the LP algorithm, the shifts are maximized such that the possible resynchronization is maximized. Shifting these bounds results in larger buffer capacities. We illustrate this by means of Figure 6. This figure shows schedules for the actors of the CSDF graph in Figure 3. Tokens must be produced before they are consumed. For edge e_{12} this can be ensured by positioning the production bound ($\check{p}_{12}$) above the consumption bound ($\hat{c}_{12}$). This defines a constraint on the start times of the first firings of the two actors. The difference between the start times is annotated with β in Figure 6. For edge e_{21} the maximum vertical difference between the production bound ($\check{p}_{21}$) and consumption bound ($\hat{c}_{21}$) specifies the number of

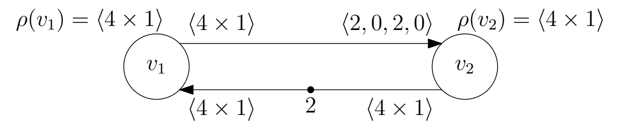
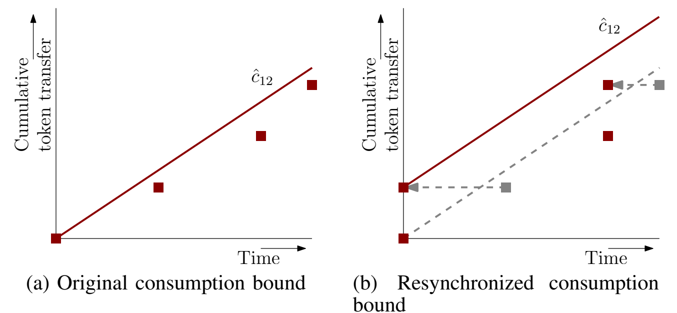


Figure 4. Resynchronized analysis graph of the code in Figure 2



(a) Original consumption bound (b) Resynchronized consumption bound

Figure 5. Bounds on the consumption on edge e_{12} in Figures 3 and 4

```
while(1) {
  for (int i=0; i<4; i++) {
    acquireSpace(cx, 1);
    x[i] = F(i);
    releaseData(cx, 1);
  }
}
```

(a) Task t_1

```
while(1) {
  for (int i=0; i<4; i++) {
    acquireData(cx, 1);
    printf("%d", x[i]);
    releaseSpace(cx, 1);
  }
}
```

(b) Task t_2

```
while(1) {
  for (int i=0; i<4; i++) {
    if (i % 2 == 0) acquireData(cx, 2);
    printf("%d", x[i]);
    releaseSpace(cx, 1);
  }
}
```

(c) Task t_2 resynchronized

Figure 2. Parallelized code for Figure 1

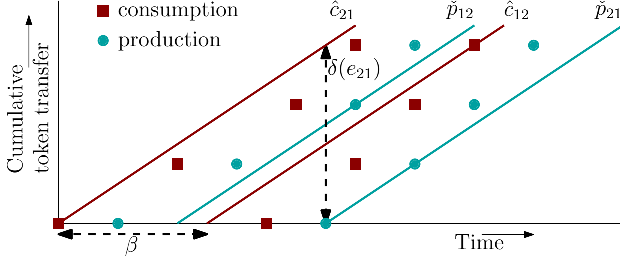


Figure 6. Example schedules for the actor firings of Figure 3, with from left to the right, the consumption bound on edge e_{21}, the production bound and the consumption bound on e_{12} and the production bound on e_{21}.

needed initial tokens, $\delta(e_{21})$. This number of tokens ensures that initially enough tokens can be consumed before tokens are produced on edge e_{21}. The number of needed initial tokens corresponds with the needed buffer capacity in the task graph. Shifting $\check{p}_{21}$ and $\hat{c}_{21}$ leads to an increase of the difference between $\check{p}_{21}$ and $\hat{c}_{21}$ and also shifting $\check{p}_{12}$ and $\hat{c}_{12}$ can lead, due to the start time constraint, indirectly to an increase of the vertical difference between $\check{p}_{21}$ and $\hat{c}_{21}$. Each shift of the bounds can thus lead, indirect or direct, to an increase of the needed buffer capacity. This shows the trade-off between the memory size and the resynchronization in the solution.

IV. Analysis Model

In this section we describe the analysis model we use for the calculation of maximum resynchronization. We first describe the model and its formalization and then we define properties of the model on which our analysis techniques rely.

The model we use to calculate the maximum amount of resynchronization is a CSDF [5] graph that models the synchronization behavior of an application. A CSDF graph is a directed graph $G = (V, E, \delta, \rho, \pi, \gamma, \theta)$ that consists of actors V and finite set of directed edges, $E = \{(v_i, v_j) \mid v_i, v_j \in V\}$ where (v_i, v_j) is an edge from v_i to v_j. We furthermore write e_{ij} for (v_i, v_j).

The number of initial tokens on edge $e \in E$ is given by $\delta(e)$ with $\delta : E \to \mathbb{N}$. The number of distinct phases of execution of an actor v_i is given by $\theta(v_i)$ and the actor transitions between phases in a cyclic fashion. We call these $\theta(v_i)$ phases the cyclo-static period.

A firing of an actor is enabled when on all input edges of the actor sufficient tokens are present. The token consumption quantum is the number of tokens that are required on an edge $e_{ij} \in E$ in a particular firing. This quantum is in firing f of actor v_j on edge e_{ij}, $\gamma(e_{ij}, ((f-1) \mod \theta(v_j)) + 1)$ tokens, with $\gamma : E \times \mathbb{N} \to \mathbb{N}$. The specified number of tokens is consumed atomically from all input edges when the actor fires. The number of produced tokens in firing f of actor v_i on edge e_{ij} is denoted by $\pi(e_{ij}, ((f-1) \mod \theta(v_i)) + 1)$, with $\pi : E \times \mathbb{N} \to \mathbb{N}$. All tokens of firing f of actor v_i are produced atomically on each output edge at the end of firing f. A firing ends $\rho(v_i, ((f-1) \mod \theta(v_i)) + 1)$ time units after it starts.

For edge e_{ij}, we define $\Pi(e_{ij}) = \sum_{f=1}^{\theta(v_i)} \pi(e_{ij}, f)$ as the number of tokens produced in one cyclo-static period and $\Gamma(e_{ij}) = \sum_{f=1}^{\theta(v_j)} \gamma(e_{ij}, f)$ as the number of tokens consumed in one cyclo-static period. Furthermore we assume that each actor v_i has an implicit self-edge e_{ii} with a single initial token such that subsequent firings of an actor can not overlap.

Furthermore we make shorthands for the cumulative token production and consumption of phase 0 up to (and without) phase p: $\Xi(e_{ij}, p) = \sum_{k=0}^{(p-1)} \pi(e_{ij}, p)$ for the production and $\Lambda(e_{ij}, p) = \sum_{k=0}^{(p-1)} \gamma(e_{ij}, p)$ for the consumption.

We use $\zeta(e_{ij})$ to specify the size that a token on edge e_{ij} takes in the memory.

Consistency of CSDF graphs is defined as in [5]. Similar to [5] we define a positive integer vector $\mathbf{z}$ of length $|V|$ which is called the repetition vector of the CSDF graph. For each actor v_i, element z_i of the repetition vector denotes the repetition rate of v_i and specifies the relative firing frequency between actors.

As we will see later on, resynchronization only modifies the times of consumptions and productions within the cyclo-static period. This means that $\Pi(e_{ij})$ and $\Gamma(e_{ij})$ do not change and thus consistency is preserved. The repetition vector of the CSDF graph does not change either.

For a strongly connected and consistent CSDF graph a period μ can be specified in which each actor should fire $z_i \cdot \theta(v_i)$ times. This μ can be used to steer the average token transfer rate which corresponds with the throughput.

If a CSDF graph is executed in a self-timed manner, actors start execution as soon as the required tokens are available. Self-timed execution of a CSDF graph is monotonic. This means that a decrease in response time or start time can only lead to earlier production time and can therefore never delay any later firing. As an addition to this, it can be shown that an increase of the number of initial tokens cannot lead to an increase in the enabling time of any firing [13]. This property allows us to calculate with a conservative number of initial tokens.

V. Resynchronization

In this section we first explain a method for transforming resynchronization into a shift of the linear bounds on the schedule as shown in Section III. We then extend the algorithm used for the calculation of start times of the actors in [6] such that a maximum resynchronization is calculated simultaneously with the start times while bounding the used memory. The result is a maximum resynchronization for each edge together with buffer capacities which satisfy the throughput and memory size constraint. We use the method presented in [6] to first determine a schedule for each actor. We refer to the

start time of firing f of actor v_i in this schedule as $s(v_i, f)$. We do not change these start times of firings but only assign different consumptions / productions to the firings.

We assume as an input for the calculation of resynchronization the set of edges for which resynchronization must be maximized. $E_c \subseteq E$ and $E_p \subseteq E$ are the sets of edges for which respectively consumption and production resynchronization must be determined. For each of these edges we assume that there is a specification of the phases on which resynchronization must be calculated. For simplicity we assume that the production / consumption quanta are equal for all of the phases that are specified for resynchronization. For the production resynchronization on edge e_{ij} we specify these phases as a list of consecutive phases of actor v_i starting at phase $\mathcal{P}_b^p(e_{ij})$ and ending at phase $\mathcal{P}_e^p(e_{ij})$: $\langle \mathcal{P}_b^p(e_{ij}), \ldots, \mathcal{P}_e^p(e_{ij}) \rangle$. For the consumption resynchronization the phases are specified similar: $\langle \mathcal{P}_b^c(e_{ij}), \ldots, \mathcal{P}_e^c(e_{ij}) \rangle$. We use $\theta_{ij}^p = \mathcal{P}_e^p(e_{ij}) - \mathcal{P}_b^p(e_{ij}) + 1$ for the number of phases on which the production resynchronization is applied. The number of phases on which consumption resynchronization is applied is defined analogues and is called: θ_{ij}^c .

The phases that are specified for resynchronization can be bounded with a linear bound. We call this bound the resynchronization bound. On each edge e_{ij}, $\Pi(e_{ij}) \cdot z_i$ tokens are produced in μ time. With this we know that a linear bound with a slope of $\frac{\Pi(e_{ij}) \cdot z_i}{\mu}$ is able to bound the schedule of the productions from below, we only have to determine the vertical offset of this bound. The slopes of the production and consumption bounds are equal because consistency tells us that $\Pi(e_{ij}) \cdot z_i = \Gamma(e_{ij}) \cdot z_j$ holds. For edge e_{ij} we write α_{ij} for its slope. We can now define a linear production and consumption bound with slope α_{ij} which only bounds productions / consumptions of the phases specified for resynchronization. In case of consumption resynchronization, this partial consumption bound is always lower or equal to the consumption bound on all the phases. The production bound on the resynchronization phases is also always higher or equal to the production bound on all the phases.

The partial consumption bound on the resynchronization phases of edge e_{ij} is called the consumption resynchronization bound and can be defined as $\alpha_{ij} \cdot t + \beta_{ij}^{rc}$ with β_{ij}^{rc} as:

$$\beta_{ij}^{rc} = \max_{p \in \langle \mathcal{P}_b^p(e_{ij}), \ldots, \mathcal{P}_e^p(e_{ij}) \rangle} (\Xi(e_{ij}, p) - \alpha_{ij} \cdot s(v_j, p)) \quad (1)$$

The production resynchronization bound of edge e_{ij} can be defined similar: $\alpha_{ij} \cdot t + \beta_{ij}^{rp}$ with β_{ij}^{rp} as:

$$\beta_{ij}^{rp} = \min_{p \in \langle \mathcal{P}_b^c(e_{ij}), \ldots, \mathcal{P}_e^c(e_{ij}) \rangle} (\Lambda(e_{ij}, p) - \alpha_{ij} \cdot (s(v_i, p) + \rho(v_i, p))) \quad (2)$$

Resynchronization can be seen as shifting consumptions and productions in the time. Productions can be postponed and consumptions can be advanced compared to the original schedule. These shifts are allowed because tokens can be produced later than was possible in the application and tokens are consumed earlier than was needed in the application. Logically productions and consumptions are shifted from one phase to an other such that they correspond with a certain firing. However this would lead to integer constraints in our optimization problem. To linearize the problem, we decouple the actual consumption and production from the phases. In our algorithm a consumption is an amount of tokens that must be consumed somewhere in the schedule as long as it is consumed earlier (or equal) than in the original schedule. For the production we do the same but now we demand that the tokens are produced at a later or equal time in the schedule than in the original schedule. After the calculation of the maximum shifts, the productions and consumptions can be assigned again to actual phases.

For each of the edges on which production resynchronization must be applied, we calculate a resynchronization factor, $\mathcal{R}_{ij}^p \in \mathbb{R}^+$. This factor denotes the maximum number of phases that a token production may be postponed. For the consumption resynchronization we have a similar factor: $\mathcal{R}_{ij}^c \in \mathbb{R}^+$ which denotes the maximum number of phases that a token consumption may be advanced. These factors are maximized in the algorithm given a memory size constraint. The maximum value for $\mathcal{R}_{ij}^p$ is $(\theta_{ij}^p - 1)$, because then all the productions can be shifted to the last phase of the cyclo-static period and thus higher values of $\mathcal{R}_{ij}^p$ will not make sense. The maximum value for $\mathcal{R}_{ij}^c$ is $(\theta_{ij}^c - 1)$.

The postponement of a production needs to be captured by the production resynchronization bound which was defined on the phases specified for resynchronization. The needed shift of this bound can be conservatively defined such that the productions are bounded for each possible value of the resynchronization factor, $\mathcal{R}_{ij}^p$. This can be done by using the maximum time a production postpones if it is shifted from one phase to a subsequent phase:

$$\mathcal{T}_{ij}^p = \max_{p \in \langle \mathcal{P}_b^p(e_{ij})+1, \ldots, \mathcal{P}_e^p(e_{ij}) \rangle} (s(v_i, p) - s(v_i, p-1))$$

With this we can redefine the production resynchronization bound of e_{ij} to $\alpha_{ij} \cdot t + \beta_{ij}^{rp} - \Delta_{ij}^p \cdot \mathcal{R}_{ij}^p$ with $\Delta_{ij}^p = \alpha_{ij} \cdot \mathcal{T}_{ij}^p$

The production bound on all the phases of e_{ij} can now be defined such that it also takes the resynchronization into account. The vertical offset of this bound can now be defined as:

$$\beta_{ij}^{p'} = \min(\beta_{ij}^p,\ \beta_{ij}^{rp} - \Delta_{ij}^p \cdot \mathcal{R}_{ij}^p) \quad (3)$$

From this definition we see that this production bound does not change if the production resynchronization bound is higher than the production bound on all the phases. We can therefore use the distance between these two bounds as free resynchronization space. In order to remove the min expression, we can define a lower limit on the resynchronization factor such that the production resynchronization bound is dominant for the production bound and thus the free resynchronization space is always used maximally: $\mathcal{R}_{ij}^p \geq \frac{\beta_{ij}^{rp} - \beta_{ij}^p}{\Delta_{ij}^p}$. If $\frac{\beta_{ij}^{rp} - \beta_{ij}^p}{\Delta_{ij}^p}$ is greater than $(\theta_{ij}^p - 1)$, the difference between the resynchronization bound and the original bound is greater than the maximum value of $\mathcal{R}_{ij}^p$. This means that the production on e_{ij} can be resynchronized fully without modifying the production bound. In the algorithm presented below we can use the original production bound as linear production bound for this edge and can remove the edge from E_p because resynchronization does not need to be determined in the algorithm for this edge. We now specify the vertical offset, $\beta_{ij}^{rp'}$, of bound $\check{p}$ as:

$$\beta_{ij}^{rp'} = \begin{cases} \beta_{ij}^{rp} & \text{if } e_{ij} \in E_p \\ \beta_{ij}^p & \text{otherwise} \end{cases} \quad (4)$$

With this we can specify a new production bound on e_{ij} in which $\mathcal{R}_{ij}^p$ is 0 for edges $e_{ij} \notin E_p$:

$$\check{p}_{ij}(t) = \alpha_{ij} \cdot (t - s(v_i)) + \beta_{ij}^{rp'} - \Delta_{ij}^p \cdot \mathcal{R}_{ij}^p + \delta(e_{ij}) \quad (5)$$

The bound on the consumptions shifted by the resynchronization can be defined similar with all the constraints and definitions analogues to the ones for the production bound but now moving the bound upwards:

$$\hat{c}_{ij}(t) = \alpha_{ij} \cdot (t - s(v_j)) + \beta_{ij}^{rc'} + \Delta_{ij}^c \cdot \mathcal{R}_{ij}^c \quad (6)$$

As in [6] we require that $\forall_{e_{ij} \in E} : \check{p}_{ij}(t) \geq \hat{c}_{ij}(t)$. We can rewrite this, similar to the procedure in [6], to a constraint in start times:

$$s(v_j) - s(v_i) \geq \frac{\beta_{ij}^{rc'} + \Delta_{ij}^c \cdot \mathcal{R}_{ij}^c - (\beta_{ij}^{rp'} - \Delta_{ij}^p \cdot \mathcal{R}_{ij}^p + \delta(e_{ij}))}{\alpha_{ij}} \quad (7)$$

In this equation, $\delta(e_{ij})$ represents the maximum allowed number of initial tokens on edge e_{ij} which controls the maximum difference between the start times of actors v_i and v_j.

Algorithm 1

Minimize

$$\sum_{e_{ij} \in E} a(e_{ij}) \zeta(e_{ij}) \delta'(e_{ij}) - \sum_{e_{ij} \in E_p} b_p(e_{ij}) \mathcal{R}_{ij}^p - \sum_{e_{ij} \in E_c} b_c(e_{ij}) \mathcal{R}_{ij}^c$$

Subject to

$$\forall e_{ij} \in E : s(v_j) - s(v_i) - \frac{\Delta_{ij}^c \cdot \mathcal{R}_{ij}^c + \Delta_{ij}^p \cdot \mathcal{R}_{ij}^p}{\alpha_{ij}} \geq \frac{\beta_{ij}^{rc'} - (\beta_{ij}^{rp'} + \delta(e_{ij}))}{\alpha_{ij}} \quad (8)$$

$$\forall v_{ij} \in V : s(v_i) \geq 0 \quad (9)$$

$$\forall e_{ij} \in E : \delta'(e_{ij}) \geq \alpha_{ij} \cdot (s(v_i) - s(v_j)) - \beta_{ij}^{rp'} + \Delta_{ij}^p \cdot \mathcal{R}_{ij}^p + \beta_{ij}^{rc'} + \Delta_{ij}^c \cdot \mathcal{R}_{ij}^c \quad (10)$$

$$\forall e_{ij} \in E : \delta'(e_{ij}) \geq 0 \quad (11)$$

$$\forall e_{ij} \in E_p : (\theta_{ij}^p - 1) \geq \mathcal{R}_{ij}^p \geq \frac{\beta_{ij}^{rp} - \beta_{ij}^p}{\Delta_{ij}^p} \quad (12)$$

$$\forall e_{ij} \in E_c : (\theta_{ij}^c - 1) \geq \mathcal{R}_{ij}^c \geq \frac{\beta_{ij}^c - \beta_{ij}^{rc}}{\Delta_{ij}^c} \quad (13)$$

$$\forall e_{ij} \in E \backslash E_p : \mathcal{R}_{ij}^p = 0 \quad (14)$$

$$\forall e_{ij} \in E \backslash E_c : \mathcal{R}_{ij}^c = 0 \quad (15)$$

$$\sum_{e_{ij} \in E} (\delta'(e_{ij}) + 1) \cdot \zeta(e_{ij}) \leq M \quad (16)$$

Algorithm 1 computes the maximum amount of resynchronization given a memory constraint M. The solution of the algorithm can be controlled by the weight factors $a(e_{ij})$, $b_p(e_{ij})$ and $b_c(e_{ij})$. Equation 8 constraints the start times of actors as defined in (7). It ensures that it is possible to find start times for the actors guaranteeing the throughput constraint and also filters the resynchronization solutions leading to deadlock. For edges that may contain initial tokens, (10) specifies the needed number of initial tokens such that given the start times and resynchronization, $\check{p}_{ij}(t) \geq \hat{c}_{ij}(t)$ holds for these edges.

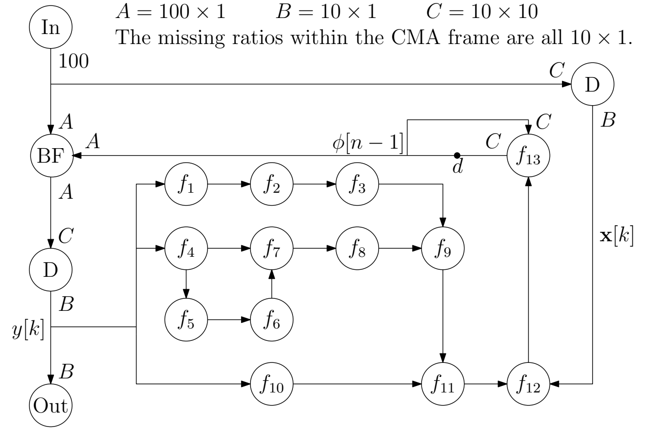


Figure 7. Task graph of the Extended Constant Modulus Algorithm

By using the token size, this number of initial tokens can be transformed to the needed buffer capacity. The sum of these buffer capacities needs to be bounded by M, which is enforced by (16). The number of initial tokens is increased by one to conservatively take the effect of ceiling this number after the algorithm into account. With (12) and (13) we bound the variables containing the resynchronization factor such that they do not exceed their maximum value and that for each edge e_{ij}, $\beta_{ij}^{p'}$ and $\beta_{ij}^{c'}$ are determined by the resynchronization bounds as explained for the derivation of (4).

The LP algorithm calculates with real numbers. We however need a integer value for each of the resynchronization factors such that they correspond with a number of phases that a production or consumption may be shifted and thus can be assigned again to an actual firing. This is done by flooring these values. The resulting resynchronization factors are now $\lfloor \mathcal{R}_{ij}^p \rfloor$ and $\lfloor \mathcal{R}_{ij}^c \rfloor$. Flooring the resynchronization factors means that productions / consumptions are shifted less phases than was determined as the maximum shift thus this is allowed.

The determined number of initial tokens $\delta(e_{ij})$ needs be a integer value too. Due to the monotonic behavior of a CSDF graph, the number of initial tokens may be increased without violating the throughput constraint and can therefore be ceiled. Using the floored resynchronization factors we can specify the number of sufficient initial tokens $\delta_s(e_{ij})$ as the smallest integer satisfying (10):

$$\delta_s(e_{ij}) = \lceil \alpha_{ij} \cdot (s(v_i) - s(v_j)) - \beta_{ij}^{rp'} + \Delta_{ij}^p \cdot \lfloor \mathcal{R}_{ij}^p \rfloor + \beta_{ij}^{rc'} + \Delta_{ij}^c \cdot \lfloor \mathcal{R}_{ij}^c \rfloor \rceil \quad (17)$$

The resynchronization factors can also be used to modify the CSDF graph and a corresponding application such that the synchronization granularity increases. This can be done in a similar way as shown in Section III.

VI. Case-Study

In this section the resynchronization technique is demonstrated on a CMA algorithm of which the task graph is shown in Figure 7. All functions (f_i) in the figure, are fine-grained parallel tasks. Together they form a task graph for which we can construct a CSDF model in a similar way as in Section III. We apply the presented resynchronization technique on this CSDF model in order to increase the synchronization granularity. The total number of synchronization statements considered

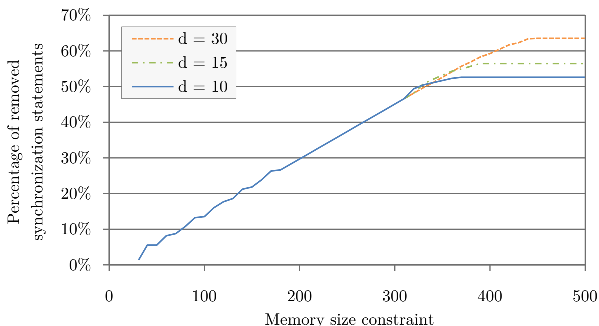


Figure 8. Trade-off between memory size and resynchronization

for resynchronization in this application equals 650. The goal is to decrease this number of synchronization statements. The possible reduction is limited because of the cycles containing edge (f_{13}, BF). The number of initial data items on this edge (d items in the task graph) specifies the maximum amount of tokens at any time on each edge in these cycles. The maximum number of tokens that can be consumed is thus equal to this value d. This limits the amount of phases that a consumption may be advanced and a production may be postponed. However resynchronizing the synchronization calls that regard space (not data) of the buffer (*acquireSpace* and *releaseSpace* in Section III) is allowed if the buffer sizes are large enough. Resynchronization of these calls does not add delay to the critical cycles while, as illustrated in Figure 3, these edges have an opposite direction and a variable number of initial tokens.

The algorithm needs to find the edges at which resynchronization is possible within the constraints of the cycles in the task graph. If the memory size constraint is small, the production on edge (f_{13}, BF) is resynchronized. Relaxation of this memory size constraint leads to the resynchronization of the consumption of f_1, f_4 and f_{10} instead of the resynchronization of edge (f_{13}, BF) because for this option, three instead of one buffer is increased in size. Further relaxation of this constraint changes the resynchronization in this cycle to the consumption resynchronization of f_2, f_6, f_7 and f_{10}. Because the algorithm maximizes the resynchronization, it is able to choose between above options of resynchronization in the critical cycles and resynchronization of the synchronization calls regarding space in the buffer depending on the cost in buffer sizes.

Figure 8 shows the trade-off between the memory size constraint and the amount of resynchronization that is possible given a throughput constraint of $\mu = 200$ and equal firing durations: $\forall_{v_i \in V} : \rho(v_i) = 1$. The weight factors and token sizes are also chosen to be equal. It is also shown that increasing the number of initial data items in the task graph (d) allows more resynchronization. The execution time of the resynchronization algorithm for this problem is below 20 milliseconds (using GLPK). The reader may wonder why we did not use a more accurate Integer Linear Programming (ILP) formulation in the algorithm. Tests with such a more accurate ILP formulation show that the results of the algorithm indeed slightly improve but the execution time of the algorithm rapidly increases for more relaxed memory constraints. For the same application the algorithm did not return an answer, given a memory constraint of 200 tokens, within one hour.

The method as used in this case-study is implemented in a parallelization tool for stream processing applications. The CSDF synchronization model generated by the tool is used to compute a possible resynchronization which is used to modify the original tasks.

VII. Conclusion

This paper presents a linear programming formulation for resynchronization of a CSDF graph given a throughput and a memory size constraint. It is shown that expressing advancing consumptions and postponing productions as a shift of linear bounds, enables the creation of an LP formulation that can be solved in polynomial time. This paper focuses on the resynchronization method self. Actual improvements depend on the application, granularity, platform, etc. and is considered as future work.

The case-study illustrates that especially cyclic dependencies can make resynchronization within a memory size constraint challenging. Furthermore it is shown that relaxation of the memory size constraint allows additional resynchronization.

The presented resynchronization approach can be useful for applications which are parallelized at a fine granularity. Our method can decrease the synchronization overhead of these applications by increasing the synchronization grain without violation of the throughput constraint. The presented method is suitable as an optimization step in a parallelization tool which makes the fine-grain parallelism explicit.

References

[1] P. Shaffer, “Minimization of Interprocessor Synchronization in Multiprocessors with Shared and Private Memory,” in *1989 International Conference on Parallel Processing, University Park, PA*, 1989.

[2] S. Bhattacharyya, S. Sriram, and E. Lee, “Minimizing Synchronization Overhead in Statically Scheduled Multiprocessor Systems,” in *Proceedings of the IEEE International Conference on Application Specific Array Processors*. IEEE Computer Society, 1995, p. 298.

[3] ——, “Self-Timed Resynchronization: A Post-Optimization for Static Multiprocessor Schedules,” in *Proceedings of the 10th International Parallel Processing Symposium*. IEEE Computer Society, 1996, pp. 199–205.

[4] S. Sundararajan, S. Sriram, and E. Lee, “Latency-Constrained Resynchronization for Multiprocessor DSP Implementation,” *Laboratory, University of Maryland at College Park*, 1996.

[5] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, “Cyclo-Static Dataflow,” *IEEE Transactions on Signal Processing*, vol. 44, no. 2, pp. 397–408, 1996.

[6] M. Wiggers, M. Bekooij, and G. Smit, “Efficient Computation of Buffer Capacities for Cyclo-Static Dataflow Graphs,” in *Proceedings of the 44th annual Design Automation Conference*. ACM, 2007, p. 663.

[7] S. Bhattacharyya, S. Sriram, and E. Lee, “Resynchronization for Multiprocessor DSP Systems,” *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 47, no. 11, pp. 1597–1609, 2000.

[8] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer, “An Interactive Environment for Data Partitioning and Distribution,” in *Distributed Memory Computing Conference, 1990., Proceedings of the Fifth*, 1990, pp. 1160–1170.

[9] C. Polychronopoulos, “Loop Coalescing: A Compiler Transformation for Parallel Machines,” in *Proceedings of the 1987 International Conference on Parallel Processing*, 1987, pp. 235–242.

[10] M. O’Keefe and H. Dietz, “Loop Coalescing and Scheduling for Barrier MIMD Architectures,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 9, pp. 1060–1064, 1993.

[11] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam, and N. Shenoy, “A Global Communication Optimization Technique Based on Data-Flow Analysis and Linear Algebra,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 21, no. 6, pp. 1251–1297, 1999.

[12] M. Wiggers, M. Bekooij, and G. Smit, “Monotonicity and Run-Time Scheduling,” in *Proceedings of the seventh ACM international conference on Embedded software*. ACM, 2009, pp. 177–186.

[13] M. H. Wiggers, “Aperiodic multiprocessor scheduling for real-time stream processing applications,” Ph.D. dissertation, University of Twente, Enschede, The Netherlands, June 2009.