Estimation and Optimization of the Performance of Polyhedral Process Networks

Haastregt, S. van

CHAPTER 5

APPLICATION TRANSFORMATION

In the previous chapter, we have presented different methods for fast performance evaluation of applications modeled as a PPN. In this chapter, we present how the results of such evaluations can be used to obtain alternative application instances. These alternative application instances are functionally equivalent, but differ in performance and resource cost characteristics. In Section 5.1, we discuss four transformations that we consider to automatically obtain alternative application instances from a given application specification. In Section 5.2, we present heuristics to select when and with which parameters the four transformations should be applied. In Section 5.3, we summarize this chapter.

5.1 Transformations

In the Daedalus design flow, the application is specified as a sequential program. By default, a single PPN process is constructed for each function call in the sequential program. The PPN obtained can easily be transformed in another PPN by transforming the sequential program such that a functionally equivalent PPN with different performance and resource cost characteristics is obtained. In this section, we consider the following four transformations: splitting (Section 5.1.1), merging (Section 5.1.2), stream multiplexing (Section 5.1.3), and scheduling (Section 5.1.4).

5.1.1 Splitting

To increase the amount of potential parallelism in an application modeled as a PPN, a designer can increase the number of processes. An established way to achieve this is by applying a process splitting transformation [SKD02, MNS09].

Definition 5.1 (Process Splitting Transformation)**.**
A *splitting transformation* with factor N on a PPN process p generates N copies of p that are identified as $p_0, p_1, \ldots, p_{N-1}$. These copies are referred to as the *partitions* of p. The original process iteration domain D_p is split into disjoint *subdomains* D_{p_0}, $D_{p_1}, \ldots, D_{p_{N-1}}$. The original process p is removed from the PPN.

Each partition executes the same function as the original process p, but for different iterations. The splitting transformation resembles a loop unrolling transformation in which a loop body is duplicated a number of times, or a loop splitting transformation in which a loop is split into multiple loops that each iterate over a subset of the iteration points of the original loop [Muc97, Chapter 17]. The original purpose of these loop transformations in a compiler is to increase instruction-level parallelism, whereas the purpose of process splitting in this dissertation is to increase the amount of coarser-grained task-level parallelism in an application. The process iteration domain of a process can be split using different systematic approaches to obtain different distributions of the points in the original process iteration domain [Ste04, SKD02]. In this chapter, we consider two different systematic approaches: modulo unfolding and plane cutting.

Definition 5.2 (Modulo Unfolding)**.**
A *modulo unfolding* splitting transformation, specified as $unfold(p, d, N)$, splits a process p into N partitions on dimension d. The process iteration domain of each instance p_i becomes

$$D_{p_i} = \{\mathbf{x} \mid \mathbf{x} \in D_p \ \wedge \ \mathbf{x}_d \bmod N = i\}.$$

Our $unfold$ transformation is defined for a single dimension d, whereas the UNFOLD procedure of Stefanov et al. is defined for multiple dimensions [Ste04, Chapter 3.3]. This merely serves to simplify our definition of $unfold$, motivated by our observation that unfolding transformations are often applied on a single dimension only. The behavior of Stefanov's UNFOLD procedure can always be obtained by applying $unfold$ on the partitions created by a previous $unfold$ transformation.

Definition 5.3 (Plane Cutting)**.**
A *plane cutting* splitting transformation, specified as $planecut(p, H)$, splits a process p using a set of hyperplanes $H = h_0, h_1, \ldots, h_{|H|-1}$. The hyperplanes divide process domain D_p into N subdomains, where N depends on the actual hyperplanes specified. For each subdomain x, a partition p_x with domain $D_{p_x} = x$ is created.

Process domains and dependence relations exhibit a regular structure when they are derived from static affine nested loop programs that have repetitive and regular behavior. As a result, the hyperplanes cutting such domains are closely related to

```
1  for (i = 0; i < 4; i++)
2    P1(&x[i]);
3
4  for (i = 0; i < 3; i++)
5    P2(&(y[i]) );
6
7  for (i = 0; i < 4; i++)
8    for (j = 0; j < 3; j++)
9      F(x[i], y[j], &x[i], &y[j]);
10
11 for (i = 0; i < 4; i++)
12   C(x[i]);
```

Figure 5.1: Sequential C code on which we demonstrate transformations.

each other. Therefore, we present an alternative way of specifying a plane cutting transformation, by specifying a single hyperplane h and a factor N instead of a set of hyperplanes H:

Definition 5.4 (Plane Cutting (alternative))**.**
Alternatively, a plane cutting transformation specified as $planecut(p, h, N)$ splits a process p into N instances using hyperplanes parallel to hyperplane h. A set of parallel hyperplanes H that divide D_p into N subdomains with comparable cardinalities are obtained by searching as explained by de Zwijger [Zwi12, Algorithm 1]. The process iteration domain of each instance p_i becomes

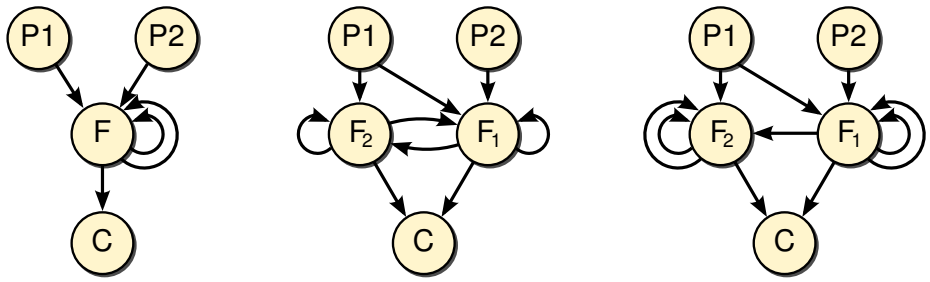
$$D_{p_i} = \{\mathbf{x} \mid \mathbf{x} \in D_p \ \wedge \ h_i \leq \mathbf{x} < h_{i+1}\}.$$

Examples

In Figure 5.2a, we show the PPN derived from the C program shown in Figure 5.1. In Figure 5.2b and 5.2c, we show the PPNs after applying modulo unfolding and plane cutting transformations on process F. We assume splitting factor $N = 2$ for both transformations, such that two partitions F_1 and F_2 are obtained.

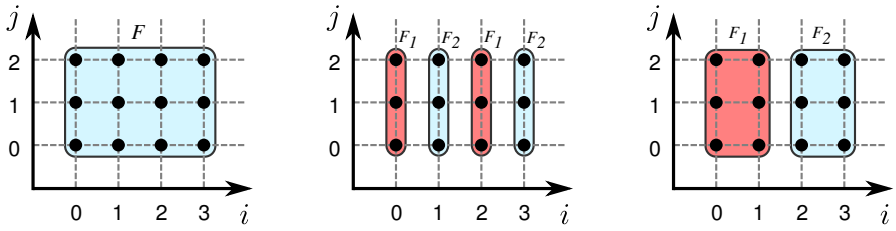
The original domain of process F is shown in Figure 5.3a. It consists of twelve points, corresponding to the twelve iterations of the for-loops at lines 7–8 in Figure 5.1. In Figure 5.3b, we show the two subdomains obtained after applying a modulo unfolding transformation $unfold(F, i, 2)$. The subdomain of F_1 consists of the six points in the original domain D_F for which $i \bmod 2 = 0$. The subdomain of F_2 consists of the remaining six points in the original domain D_F for which $i \bmod 2 = 1$.

In Figure 5.3c, we show the two subdomains obtained after applying a plane cutting transformation $planecut(F, \{i = 2\}, 2)$. The subdomain of F_1 consists of the six



a) Original. b) After modulo unfolding. c) After plane cutting.

Figure 5.2: A PPN and two transformed instances of the same PPN, obtained by splitting process F by a factor of two on its outermost dimension.



a) Original. b) Modulo unfolding. c) Plane cutting.

Figure 5.3: Process domains obtained after splitting by a factor of two.

points in the original domain D_F for which $i < 2$. The subdomain of F_2 consists of the remaining six points in the original domain D_F for which $i \geq 2$.

Position in Tool Flow

Splitting transformations not only affect the process being split, but also processes and channels adjacent to this process. For example, in the transformed PPNs shown in Figure 5.2b and 5.2c, process *P1* has two outgoing channels, whereas it has only one outgoing channel in the original PPN shown in Figure 5.2a. The precise implications for the adjacent processes and channels depend on the applied transformations. In the example shown in Figure 5.2, modulo unfolding results in one selfloop on process F_1, whereas plane cutting results in two selfloops on process F_1.

If we would apply the *unfold* and *planecut* operations on the PPN, then we should also update the adjacent processes and channels accordingly. Instead, we apply the *unfold* and *planecut* operations on the intermediate PDG that does not yet contain dependence information in the PNGEN tool flow [Zwi12]. Different from the ap-

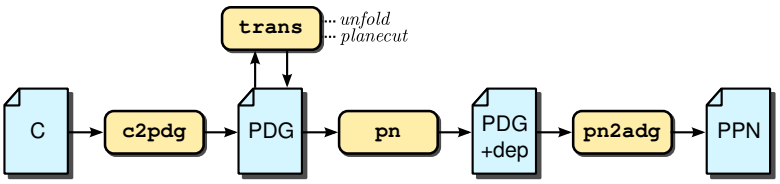


Figure 5.4: Application of splitting transformations in the PNGEN tool flow of Figure 2.9.

proach of Stefanov et al. [SKD02] that operates on the sequential code, we operate directly on polytopes. This is depicted by the `trans` block in Figure 5.4. Thus, we effectively apply transformations on an intermediate representation of the input program. The advantage of this approach is that the transformation only needs to operate on the process being split and its partitions. Adjacent processes and channels that result from the transformation are updated naturally by the PN tool, without any additional development effort needed for the `trans` block or the PN tool.

5.1.2 Merging

The splitting transformations discussed above increase the number of processes in a PPN. Assuming a separate processing resource is instantiated for each process, splitting transformations increase resource costs of PPN implementations. Complementary to splitting, the merging transformation combines processes into a single process, thereby decreasing resource costs.

Definition 5.5 (Process Merging Transformation)**.**
Application of a *merging transformation* on a set of PPN processes P results in a new *compound process* p_c which executes all firings originally executed by the processes in P. The original processes in P are removed from the PPN.

The domain cardinalities of the different processes in P are not necessarily equal. Thus, some processes in P should fire more often than others. Moreover, data dependence relations may exist between processes in P. The firings of these processes in the compound process should be scheduled such that these data dependence relations are not violated. We use the schedule computed by PNGEN to schedule the firings of the merged processes in the compound process, because this schedule includes all firings of all processes and respects data dependence relations. We refer to the work of Stefanov for further details on the merging transformation [Ste04, Chapter 3.6].

The merging transformation has been implemented in the ESPAM tool, where it can be applied by assigning multiple processes to the same processing resource in the

mapping specification. However, ESPAM only supports merging for programmable processing resources such as MicroBlaze processors. Merging LAURA processors is not supported in the current version of ESPAM. A workaround is possible if the compound process domain can be expressed as a convex polyhedral set. In such a case, the merged processes should be replaced by a compound process at the source code level. This form of merging is applied in Chapter 6.

5.1.3 Stream Multiplexing

For acyclic PPNs, the splitting transformations discussed above enable a designer to increase the throughput of a PPN. However, these splitting transformations may not yield any throughput increase for PPNs containing one or more cycles. This happens when the processes involved in a cycle depend on the output of a previous firing of its predecessor process, also known as a *recurrence* or *feedback*. As a result, the processes in a cycle may fire entirely sequentially, thereby preventing overlapped execution among the processes. Since the processes spend most of their time waiting for data in a blocking read state, their processing resources are idle for a considerable amount of time. A common solution to make use of these idle times is to process independent data streams. This can be done using a stream multiplexing transformation:

Definition 5.6 (Stream Multiplexing Transformation)**.**
Applying a *stream multiplexing transformation* with a factor N to a process p extends process domain D_p with an innermost dimension containing N points. For each value of N, process p operates on data that is not accessed for other values of N. This transformation is applied on all processes involved in a cycle of a PPN.

A stream multiplexing transformation neither increases nor decreases the latency or throughput of a single execution of the PPN. Only when multiple executions of the PPN are considered, the average period at which PPN executions finish is decreased, yielding an increase in throughput.

The stream multiplexing transformation resembles the software pipelining technique for programmable processors in which instructions from subsequent iterations of a loop are executed in an overlapping fashion [PD76, Lam88]. However, software pipelining works at the level of individual instructions, whereas our stream multiplexing transformation works at the level of coarser-grained tasks. Another difference is that software pipelining operates on the iterations of a given loop, whereas the stream multiplexing transformation introduces a new loop. Generation of a software pipelined loop for a programmable processor requires a sophisticated scheduling algorithm such as modulo scheduling. In contrast, applying the stream multiplexing

```
1 src(&v);
2 for (i = 0; i < 3; i++) {
3   P1(v, &v);
4   P2(v, &v);
5   P3(v, &v);
6 }
```

Figure 5.5: Sequential C code resulting in a PPN containing a feedback loop.

transformation on a PPN does not require any scheduling algorithm because of the self-scheduling semantics of the PPN model.

A technique closely related to stream multiplexing is C-slowing The C-slowing technique is often used in conjunction with register retiming to improve throughput of synchronous digital circuits [LRS93]. C-slowing replaces each register in a circuit with a sequence of C registers, such that C independent data streams can be in processed in an overlapped fashion. Retiming then tries to balance combinational path lengths by moving these registers through the combinational logic. As a result, the clock frequency and throughput may increase, at the expense of a higher latency caused by the additional registers. The C-slowing technique is closely related to the stream multiplexing transformation, as both add independent streams to overcome feedback in a design. However, the main purpose of C-slowing is to increase the clock frequency of a circuit, whereas the main purpose of stream multiplexing is to increase throughput of multiple executions of a PPN.

Zissules et al. conducted a case study on a QR decomposition algorithm for which they increased the number of independent streams [ZKD04]. This was done in an ad-hoc fashion, whereas our stream multiplexing provides a more systematic approach to accomplish the same goal.

Example

In Figure 5.5, we show a C program for which the corresponding PPN, shown in Figure 5.7a, contains a feedback loop involving *P1*, *P2*, and *P3*. In each execution of the PPN, processes *P1*, *P2*, and *P3* fire in sequence three times. Because each firing of these processes requires the output of the previous process through variable v, no overlapped execution occurs. This is depicted in Figure 5.8a.

In Figure 5.8b, we depict the firings of *P1*, *P2*, and *P3* after applying stream multiplexing with a second independent data set. That is, process *P1* starts operating on the first data "set" v_1 at time $t = 0$, and process *P1* starts operating on the second data set v_2 at time $t = 2$. As a result, two executions of the PPN complete in only

```
1  for (t = 0; t < F; t++)
2    src(&v[t]);
3  for (i = 0; i < 3; i++) {
4    for (t = 0; t < F; t++)
5      P1(v[t], &v[t]);
6    for (t = 0; t < F; t++)
7      P2(v[t], &v[t]);
8    for (t = 0; t < F; t++)
9      P3(v[t], &v[t]);
10 }
```

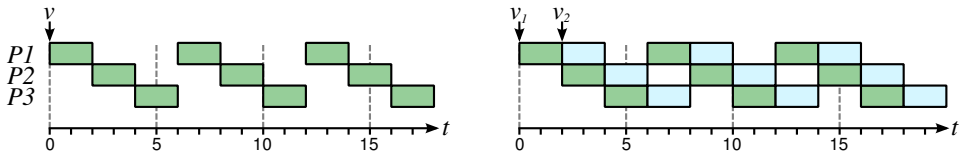
Figure 5.6: Applying stream multiplexing by a factor F to the program of Figure 5.5.

a) Original PPN. b) Corresponding PPN modeling graph.

Figure 5.7: PPN and PPN modeling graph derived from the C code in Figure 5.5.

slightly more time than needed for a single execution of the PPN. In Figure 5.6, we show the equivalent C program implementing a stream multiplexing transformation by a factor F. The transformation consists of applying a scalar expansion on all variables and adding a loop of F iterations. The scalar expanded variables are indexed using the iterator of the newly added loop.

After applying a stream multiplexing transformation of a factor two, each process is still idle for one third of the time, as shown by the gaps between the filled boxes in Figure 5.8b. This means applying a stream multiplexing transformation of a factor three would still not increase the latency of a single execution of the PPN but increase throughput of multiple executions. After stream multiplexing by a factor three, no processes are idle, which means three is the maximum stream multiplexing factor that does not increase latency for the given PPN. A stream multiplexing factor of four or higher would increase the latency of a single execution, because at time $t = 6$ process *P1* would start processing the fourth data set v_4, while output from *P3* belonging to the first data set is also available for processing by *P1*. In such a case, the splitting transformations can be considered to further increase throughput, because the cycle no longer limits throughput.



a) Without stream multiplexing. b) With stream multiplexing by a factor 2.

Figure 5.8: Flat execution profiles for the C code of Figure 5.5 and Figure 5.6.

5.1.4 Scheduling

In the previous chapters, we have assumed that each process in a PPN traverses its process iteration domain in the lexicographical order. Depending on the presence of data dependence relations between iterations, alternative execution orders of the points in the process iteration domain may exist that preserve all data dependence relations. Some of these alternative execution orders may yield a higher throughput when the iterations are executed in a pipeline fashion on for example a LAURA processor, which we show in an example below. We change the order in which the points of a process iteration domain are executed by applying a scheduling transformation.

Definition 5.7 (Process Scheduling)**.**
A *process scheduling* transformation on a process p, specified as $schedule(p)$, modifies the execution order of iterations such that independent iterations are grouped together and executed in sequence.

We distinguish between two types of schedules in a PPN: local schedules and global schedules. A *local schedule* defines the execution order of different iterations of an individual process. A *global schedule* defines the firing order of the different processes in a PPN. The scheduling transformation solely affects the local schedules of processes.

Motivating Example

We illustrate the process scheduling transformation using the PPN shown in Figure 5.2a, which was derived from the C code shown in Figure 5.1. Data dependences require that iteration $(0, 0)$ executes before iterations $(0, 1)$ and $(1, 0)$. Similarly, $(0, 1)$ should execute before $(0, 2)$.

When executing the iterations according to the original lexicographical order, we do not achieve the highest degree of overlapped execution. When implementing F using a P-stage pipeline and following the lexicographical order, execution of the first four iterations takes $3P + 1$ clock cycles, as depicted in Figure 5.9. However,

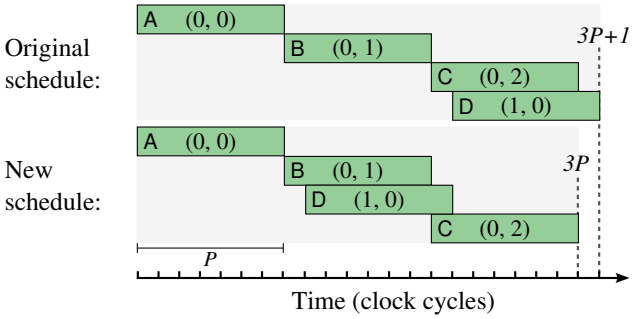


Figure 5.9: Pipeline behavior for two different schedules of the PPN shown in Figure 5.2a.

if we execute the first four iterations in the order $(0,0)$, $(0,1)$, $(1,0)$, $(0,2)$, we still respect data dependences but execution takes only $3P$ cycles. Although in this simplified scenario the gain is only one clock cycle, we have observed that changing the iteration execution order may increase throughput up to $2.7\times$ for applications such as QR decomposition [HK12].

Previous works have found that applying a skewing transformation to source code and then converting the transformed source code to a PPN may increase throughput of the PPN [SKD02, ZKD04, HK09]. A skewing transformation on the appropriate loop results in the same throughput increase for our motivating example. Thus, skewing is an effective way to increase overlapped execution, and consequently, improve pipeline utilization. However, identifying the skewing transformation parameters, such as the loop to skew, requires thorough studying of the application. Therefore, we present an automated approach to find an alternative execution order of process iterations that yields better pipeline utilization.

Scheduling PPN Processes

When applying a scheduling transformation, we use affine schedules to compactly define an execution order on the points of a process iteration domain:

Definition 5.8 (Affine Schedule)**.**
An *affine schedule* is a polyhedral map that assigns a positive time stamp to each point $\mathbf{i}$ of a process iteration domain. In this chapter, we denote an s-dimensional affine schedule as [1]

$$\theta = \left\{ \mathbf{i} \rightarrow \mathbf{i'} \mid \mathbf{i'} = H \cdot \mathbf{i} + \mathbf{h} \right\},$$

where $\mathbf{i}$ is an n-dimensional iteration vector, $\mathbf{i'}$ is an s-dimensional time stamp vector, H is an $n \times s$ matrix, and $\mathbf{h}$ is a vector of size s.

For a given iteration $\mathbf{i} \in D_p$, computing $\theta(\mathbf{i})$ yields a time stamp at which iteration $\mathbf{i}$ can execute. These time stamps should not be interpreted as absolute time, but rather as a partial order on the iterations in D_p. We assume that execution of an iteration takes one time unit and that sufficient processing resources are available to execute all iterations with the same time stamp simultaneously. Two affine schedules θ_p and θ_q for two dependent processes p and q are *valid* if for all pairs of dependent write and read operations $(\mathbf{w}, \mathbf{r})$, the schedules enforce that the write operation is executed before the read operation. That is, when a write operation $\mathbf{w}$ of p produces data for a read operation $\mathbf{r}$ of q, then $\theta_p(\mathbf{w}) \prec \theta_q(\mathbf{r})$ should hold. In the remainder of this chapter, we only consider valid schedules.

As an example, consider the affine schedule

$$\theta = \{(i, j) \rightarrow i + j\}. \tag{5.1}$$

For iteration $(1, 2)$, the schedule yields 3 which means the iteration can execute at time 3. For iteration $(2, 1)$, the schedule also yields 3. This means that both iterations can execute at the same time and, assuming that the schedule is valid, that both iterations can execute in parallel.

If a schedule is multidimensional, that is, $s > 1$, then execution times are ordered according to the lexicographical order. For example, a schedule

$$\theta = \{(i, j) \rightarrow (i + j, j)\} \tag{5.2}$$

yields $\theta(1, 2) = (3, 2)$ and $\theta(2, 1) = (3, 1)$. Because $(3, 1) \prec (3, 2)$, iteration $(2, 1)$ should execute before iteration $(1, 2)$.

A PPN process traverses its process domain in a sequential fashion according to the lexicographical order, which is a total order. That is, for any two iterations $\mathbf{i_1}$ and $\mathbf{i_2}$, the lexicographical order defines which iteration is executed first. To comply with the

[1] In literature, e.g., [Fea92a], an affine schedule is often denoted alternatively as

$$\theta(\mathbf{i}) = H \cdot \mathbf{i} + \mathbf{h},$$

where $\mathbf{i}$, H, and h follow those of Definition 5.8.

PPN process semantics, we should consider only those affine schedules that define a total order on the iterations of a process domain. The one-dimensional schedule of Equation (5.1) yields the same time stamp for different iterations, which implies it does not define a total order. Extending this schedule to the two-dimensional schedule of Equation (5.2) results in a schedule that yields a unique time stamp for each possible pair of positive values (i, j). This property allows us to use the two-dimensional schedule of Equation (5.2) in a process scheduling transformation. A schedule is said to be *bijective* if it assigns a unique time stamp to each distinct iteration vector.

To *apply* a scheduling transformation on a process p, we modify the process domain D_p to reflect the order given by an affine schedule θ_p. That is, we transform the process domain D_p into a new domain D'_p. For bijective schedules, each point in D_p has exactly one corresponding point in D'_p. The new domain D'_p is obtained by polyhedral map application of the schedule to the process domain:

$$D'_p = \theta_p(D_p). \tag{5.3}$$

The resulting domain D'_p is again traversed according to the lexicographical order.

Example Application of a Schedule

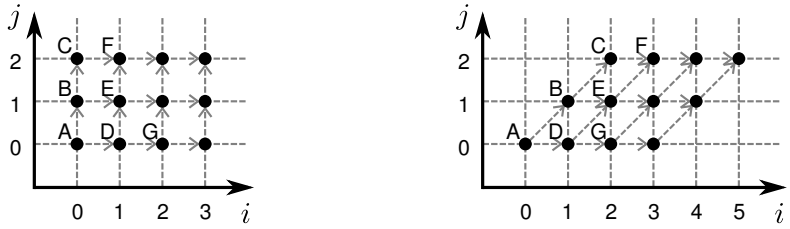
We illustrate application of a schedule using the PPN shown in Figure 5.2a. We apply a new schedule on process *F* of this PPN. The domain of this process is extracted from the for-loops in the sequential code of Figure 5.1 as

$$D_F = \left\{ (i,j) \in \mathbb{Z}^2 \;\middle|\; \begin{array}{l} 0 \leq i \leq 3 \wedge \\ 0 \leq j \leq 2 \end{array} \right\}.$$

By applying the two-dimensional schedule of Equation (5.2), we obtain a new domain

$$D'_F = \left\{ (i',j') \in \mathbb{Z}^2 \;\middle|\; \begin{array}{l} j' \leq i' \leq j' + 3 \wedge \\ 0 \leq j' \leq 2 \end{array} \right\}.$$

In Figure 5.10, we show the original and the transformed process domains. Both domains have the same cardinality because each point shown in Figure 5.10a has a counterpart in Figure 5.10b that can be obtained by applying the schedule to the point. To indicate the correspondence between points in the original and transformed domains, we have labeled seven points with a letter. For example, the counterpart of point $(1, 1)$ labeled 'E' is $(2, 1)$. The same labels are used in Figure 5.9. Traversal of the original domain according to the lexicographical order results in the execution order A, B, C, D, ... Traversal of the transformed domain according to the lexicographical order results in the execution order A, D, B, ..., C, ... This corresponds



a) Original. b) After scheduling.

Figure 5.10: Process domains of F of Figure 5.2a before and after application of the schedule in Equation (5.2). A data dependence from one iteration point of F to another is indicated by an arrow.

to the new schedule depicted in the bottom part of Figure 5.9 in which 'D' is moved forwards to execute in a pipeline fashion with 'B'. Thus, by applying the schedule of Equation (5.2), we achieve the desired overlapped execution. Below, we discuss how to obtain a schedule for a given PPN.

Determining a Schedule

The chosen schedule affects the degree of overlapped execution between process iterations that is achievable by a scheduling transformation. Finding a schedule that maximizes overlapped execution is a non-trivial optimization problem. A natural way to overlap execution of process iterations is to perform loop parallelization. This is a well-studied field in compiler technology, in which various loop parallelization algorithms have been proposed. Existing algorithms differ in the way they represent the data dependence relations of nested loop programs. For example, Allen and Kennedy's algorithm [AK87], Wolf and Lam's algorithm [WL91], and Darte and Vivien's algorithm [DV97] take as input an approximation of the dependence graph. Such an approximation restricts the ability of the algorithms to reveal all available parallelism [DRV01]. On the other hand, Feautrier's algorithm [Fea92a, Fea92b] takes the exact dependence graph as input and is therefore more powerful than the other algorithms. Also, Feautrier's algorithm will find the optimal schedule if it can be expressed as an affine mapping of the iteration space. Lim and Lam's algorithm takes a similar input representation as Feautrier's algorithm, but maximizes parallelism while minimizing the number of synchronizations [LL98].

Feautrier's algorithm is employed by e.g. the MMAlpha tool [GQR03] to generate hardware from algorithms specified in the Alpha language. Feautrier's algorithm has a high computational complexity, which motivated Feautrier to apply the algo-

rithm to sets of communicating regular processes [Fea06]. Unfortunately, Feautrier does not elaborate on the implications of the new schedule for the communication channels between processes. Later in this section, we show that these implications cannot always be ignored. Another way to address the computational complexity of Feautrier's algorithm and control flow overhead of the resulting schedules is by limiting the possible schedule coefficients [PBCC08]. This results in simpler schedules, at the expense of more scheduling dimensions, which may counteract the benefits of simpler schedules.

Applying Feautrier's Scheduling Algorithm to PPNs

Feautrier's multidimensional scheduling algorithm takes as input a *Generalized Dependence Graph* (*GDG*) represented as $G = (V, E, \mathcal{D}, \mathcal{R})$, where

- V is a set of vertexes representing statements,
- E is a set of edges representing data dependences,
- $\mathcal{D}$ is a set containing a polyhedral set for each vertex, and
- $\mathcal{R}$ is a set containing a polyhedral map for each edge.

Given a GDG, the algorithm constructs a multidimensional schedule for each statement in a greedy fashion. In each step, the algorithm constructs a linear program to find an affine function with minimum latency that satisfies as many dependence relations as possible. The dependences that are not satisfied are considered in a subsequent recursive step. Each recursive step leads to a new dimension in the schedule being constructed. The algorithm terminates when all dependences are satisfied, or when no affine schedule can be found.

We are interested in Feautrier's algorithm for two reasons. First, Feautrier's algorithm finds the *optimal* schedule if it can be expressed in the affine form of Definition 5.8. That is, no other affine schedule exists that yields a lower execution latency. This implies that Feautrier's algorithm includes all transformations that can be expressed using an affine mapping of an iteration domain, such as loop interchange or loop skewing [Fea92b, Viv02]. Second, we do not have to perform any additional analysis to run Feautrier's algorithm on a PPN because all input needed for Feautrier's algorithm is already made available by the exact data dependence analysis step of PNGEN.

To apply Feautrier's scheduling algorithm to a PPN, we relate statements to processes and dependences to channels. That is, for each process p, we add a vertex representing p to V and we add the process domain D_p to $\mathcal{D}$. For each channel c, we add an edge representing e to E and we add the channel relation M_c to $\mathcal{R}$. Feautrier's algorithm computes an affine schedule for each vertex. We apply the schedule of

each vertex to the corresponding process domain according to Equation (5.3). As a result, all processes of a PPN execute their iterations in the optimal order found by Feautrier's algorithm, potentially increasing overlapped execution.

Extension to a Bijective Schedule

The schedule returned by Feautrier's algorithm is not necessarily a bijective schedule. In fact, the schedule is only bijective if no overlap between any pair of iterations is possible, which occurs only if an application is entirely sequential. When two iterations may execute in parallel, then the schedule yields the same time stamp for both iterations. To comply with the PPN process semantics, we should extend the schedule with one or more dimensions such that for each iteration the schedule yields a unique time stamp.

We use the default algorithm of `isl` [Ver08] to extend the schedule found by Feautrier's algorithm to a bijective schedule. This default algorithm minimizes the dependence distance over the dependences, using an approach similar to Pluto's [BBK+08]. For our running example schedule of Equation (5.1), extending the schedule using `isl` gives the schedule of Equation (5.2) in which a second dimension containing j has been added.

Impact of Scheduling

The schedule computed by Feautrier's algorithm does not necessarily enforce in-order communication of data between processes. Thus, after applying the schedule, the order in which tokens are produced by the producer process may be different from the order in which tokens are consumed by the consumer process, and vice versa. This requires us to perform a reordering test [TKD07] on each channel after applying a scheduling transformation. Some channels may be classified as out-of-order after scheduling, and thus these should be implemented using a reordering buffer to preserve the functional behavior of the original application.

Existing reordering buffer designs were shown to have a considerable negative impact on both performance and resource usage [TKD03]. To avoid counteracting the performance benefits of a better schedule because of possible reordering communication, we use the reordering buffer that was presented in Section 3.6. Read and write operations on this reordering buffer take only one clock cycle. This means that replacing a FIFO buffer with a reordering buffer increases resource usage, but does not introduce additional delay cycles. As a result, we avoid counteracting the benefits of a better schedule.

```
1 for (i=0; i<=5; i++) {
2   for (j=max(0,i-3); j<=min(2,i); j++) {
3     F(x[i], y[j], &x[i], &y[j]);
4   }
5 }
```

Figure 5.11: Code to traverse the transformed iteration space of Figure 5.10b in the lexicographical order.

Another consequence of the scheduling transformation is the complexity of the evaluation logic blocks of a LAURA processor. The complexity of iterating through a rescheduled domain typically increases. To illustrate this using our running example, we show the code which iterates over the rescheduled process domain in the right part of Figure 5.11. This code is more complicated than the code iterating over the original process domain, because loop bounds of the `j`-loop are now `max` and `min` expressions involving `i`. This increases the combinational path lengths in the RTL for the evaluation logic blocks of the read and write units shown in Figure 2.12, affecting both resource usage and the maximum achievable clock frequency. From experiments, we found that control overhead induced by a scheduling transformation may reduce the clock frequency by 50%, potentially negating the benefits of increased overlapped execution. To avoid that control overhead counteracts the benefits of a better schedule, a designer may for example choose to consider the evaluation logic optimization techniques described in Section 3.5.2.

5.2 Transformation Efficacy Analysis

In the previous section, we have discussed four different transformations that can be applied on a PPN. Many combinations of these transformations are possible to obtain design points that provide different tradeoffs between circuit area and performance. Deciding which transformations to apply to obtain a particular design point is not trivial. In this section, we present how the results of PPN throughput analysis can be applied to assess the efficacy of transformations. That is, for the transformations that we consider, we discuss the conditions when a particular transformation should be applied to obtain a particular change on PPN throughput.

5.2.1 Splitting

Throughput of a PPN can often be increased by applying one of the splitting transformations discussed in Section 5.1.1 on a process. To apply a splitting transforma-

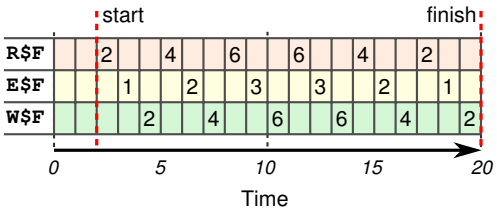


Figure 5.12: Statement execution profile for function F of the program shown in Figure 5.1. Empty cells represent zero.

tion, the designer should select a splitting method such as modulo unfolding or plane cutting, and the splitting factor. Selection of the splitting method was discussed extensively by Meijer et al. [MNS09]. However, Meijer's algorithm still requires the designer to specify a splitting factor. We therefore present heuristics to find a splitting factor in this section.

An obvious upper bound on the splitting factor of a process is the cardinality of the process domain. If the splitting factor for a process is chosen higher than the domain cardinality, then some partitions resulting from the splitting transformation contain zero iterations, meaning that a lower splitting factor would suffice as well.

Maximum Iteration Overlap

Another upper bound on the splitting factor of a process is the maximum *iteration overlap*. We define iteration overlap as the number of process iterations that can execute simultaneously at a given time, assuming a sufficient number of processing resources is available. The maximum iteration overlap thus represents the maximum number of process iterations that can execute simultaneously during the entire execution of the PPN. We propose two different methods to obtain this upper bound: by profiling or by analytical means.

Profiling-based Determination of Maximum Iteration Overlap

For the profiling-based method we employ cprof to obtain the maximum iteration overlap. We profile the sequential application code on a hypothetical ideal machine with an infinite number of processing resources as described in Section 4.6.3. We can extract the iteration overlap at a given time t from the statement execution profile of a process using Equation (4.6). By ranging t between the process start and finish times we obtain the maximum iteration overlap.

In Figure 5.12, we show an execution profile obtained using cprof for function F of the program shown in Figure 5.1. The first iteration $(0, 0)$ of the process derived from

the function call to F starts at time 2. Since F takes two input arguments, two read operations are executed which execute in parallel on the ideal machine. At time 3, the function executes and at time 4, the two output arguments are written. Considering the entire execution profile E$F in the range $[2, 20)$, at most three iterations of F execute in parallel. Thus, the maximum iteration overlap for the program of Figure 5.1 equals three.

Analytical Determination of Maximum Iteration Overlap

For the analytical method we employ Feautrier's multi-dimensional scheduling algorithm. Recall from Section 5.1.4 that for each process iteration $\mathbf{i}$, we can use Feautrier's algorithm to compute the earliest timestamp $\mathbf{t} = \theta_p(\mathbf{i})$ at which $\mathbf{i}$ can execute. This earliest timestamp is solely determined by the data dependences of the application. Feautrier's algorithm assumes no processing resource contention occurs, resembling an ideal machine. For iterations that execute in parallel, the schedule yields the same timestamp. To find out the maximum iteration overlap for a given schedule, we compute the maximum number of iterations executing at the same timestamp.

The iterations executing in parallel at a given timestamp $\mathbf{t}$ are given by the inverse of the schedule

$$\theta_p^{-1}(\mathbf{t}), \quad \text{where } \mathbf{t} \text{ is in the range of } \theta_p(\mathbf{i}), \forall \mathbf{i} \in D_p.$$

That is, we only consider timestamps $\mathbf{t}$ at which one or more points in the domain execute. A piecewise quasipolynomial that gives the number of iterations executing in parallel at a time $\mathbf{t}$ can be found by computing the cardinality using the barvinok library. The upper bound on this piecewise quasipolynomial represents the maximum number of iterations executing in parallel, and is given by

$$\max \left| \theta_p^{-1}(\mathbf{t}) \right|. \tag{5.4}$$

This upper bound can be found using the barvinok library.

We illustrate the analytical method using the schedule of Equation (5.1) for the function call to F of the program shown in Figure 5.1. The iterations executing at a timestamp t are given by the inverse polyhedral map

$$\theta^{-1}(t) = \{t \rightarrow (i, t - i) \mid 0 \leq i \leq 3 \wedge i \leq t \leq i + 2\},$$

which can be obtained using isl. For example, computing $\theta^{-1}(1)$ tells us that iterations $(0, 1)$ and $(1, 0)$ can execute in parallel at $t = 1$. This can be verified by looking at Figure 5.10a: iterations B and D only depend on A, since B and D only have an incoming arrow from A. Thus, once A has been executed, both B and D can execute.

The number of iterations that can execute at a given time t is given by the piecewise quasipolynomial

$$|\theta^{-1}(t)| = \begin{cases} t+1 & \text{if } 0 \leq t \leq 2, \\ 6-t & \text{if } 3 \leq t \leq 5, \\ 0 & \text{otherwise.} \end{cases}$$

The upper bound of this piecewise quasipolynomial equals 3, implying that at most three iterations can execute in parallel in the program of Figure 5.1. This is in agreement with the value found by means of profiling-based determination of maximum iteration overlap: at most three iterations execute simultaneously, as shown by profile `E$F` in Figure 5.12.

Average Iteration Overlap

Using both the profiling-based and analytical approaches discussed above, we found that at most three iterations execute in parallel in the program of Figure 5.1. Thus, an upper bound on the splitting factor is three. However, only during two out of six occasions, three iterations actually execute in parallel, and in four out of six occasions, a third processor would be idle.

Using the maximum iteration overlap as a splitting factor then results in a system in which some processors are used only during these few points in time. This may result in a high area overhead while a slightly lower throughput could be achieved with significantly less processors. Therefore, the average number of process iterations executing simultaneously may provide a better tradeoff between throughput and resource cost, as proposed by Eager et al. [EZL89]. We propose two different methods to obtain the average iteration overlap: by profiling or by analytical means.

Profiling-based Determination of Average Iteration Overlap

To determine the average iteration overlap by profiling, we again leverage cprof's application analysis method presented in Section 4.6.3. We extract the average iteration overlap from the statement execution profile of a process by dividing the process domain cardinality by the number of non-zero entries in `E$`.

Using Figure 5.12, we find the average iteration overlap for function `F` in the program of Figure 5.1. The process domain of `F` consists of twelve points. The execution profile `E$F` consists of six non-zero entries. Thus, the average iteration overlap is $\frac{12}{6} = 2$.

Analytical Determination of Average Iteration Overlap

To determine the average iteration overlap analytically, we again leverage Feautrier's algorithm. Instead of computing the maximum number of iterations using Equation (5.4), we compute

$$\frac{\sum_{\mathbf{t}\in\Theta_p} \left|\theta_p^{-1}(\mathbf{t})\right|}{|\Theta_p|}, \quad \text{where} \quad \Theta_p = \{\theta_p(\mathbf{i}) \mid \mathbf{i} \in D_p\}\,. \tag{5.5}$$

That is, we evaluate the piecewise quasipolynomial at every timestamp and sum these evaluations, which can be done using the `barvinok` library [Ver03a]. We then divide by the total number of timestamps to obtain the number of iterations executing in parallel on average.

For function F in the program of Figure 5.1, Equation (5.5) evaluates to

$$\frac{1+2+3+3+2+1}{6} = 2.$$

Thus, the average iteration overlap is two.

Depending on design constraints, different upper bounds on the process splitting factor may be considered. If maximum performance is required regardless of resource cost, then the maximum iteration overlap should be used as an upper bound. If a less expensive solution is required, then the average iteration overlap provides an upper bound that provides a good balance between resource cost and performance, as shown by Eager et al. [EZL89].

5.2.2 Merging

Meijer et al. investigated applying the merging transformation on programmable processors such as the MicroBlaze [MNS10]. In this section, we investigate application of the merging transformation on LAURA processors. In the general case of LAURA processor merging, resource cost savings are limited, because the IP cores implementing each process' functionality should still be provided. These IP cores often account for the greater part of the LAURA processor cost. However, when LAURA processors execute the same function, then a merging transformation can reduce resource cost by *resource-sharing* the IP core among the processes in the compound process.

The processes merged onto the same LAURA processor compete for the same IP core of the LAURA processor. This may cause a decrease in throughput if at least one of these processes is in the critical path. Therefore, two LAURA processors should only be merged if they do not execute at the same time. This can be determined by

inspecting statement execution profiles obtained from cprof. For example, assume the arrays `E$1_0`, `E$1_1`, and `E$1_2` in Figure 4.14 represent the execution profiles of three separate LAURA processors. That is, these arrays indicate when the IP core of the LAURA processor is active during the execute stage of a process iteration. At most one of the three `E$`-arrays contains a one at any time, meaning that at most one of the three LAURA processors is active at any time. Therefore, we conclude that merging these three LAURA processors into a single LAURA processor would not affect throughput.

5.2.3 Stream Multiplexing

The stream multiplexing transformation aims to increase throughput of multiple executions of a PPN containing a feedback loop. A stream multiplexing transformation can still be beneficial when the cycle mean of the feedback loop cannot be decreased by other transformations of processes in the feedback loop, such as for example a splitting transformation, or by replacement of a programmable processor with a dedicated hardware IP core. We first identify two conditions when a stream multiplexing transformation can be beneficial. We then discuss how to determine the maximum stream multiplexing factor such that the latency of a single PPN execution is not increased.

Efficacy Conditions

A first condition is that a complete execution of the PPN is independent of the previous execution of the PPN, to enable interleaving of multiple executions. This is often the case for the streaming applications that we consider, as the PPN often works on discrete and independent units of the incoming data stream such as video frames.

A second condition is that the PPN should have a feedback loop that limits throughput of a single execution of a PPN. Such a feedback loop can be detected using the MCM analysis technique presented in Section 4.5. Computing the cycle means of a PPN reveals which parts of a PPN prevent meeting a target throughput τ. The cycle means that are greater than $T = \frac{1}{\tau}$ represent parts of the PPN that prevent meeting throughput τ. The cycle means are the result from three different classes of cycles that occur in the PPN modeling graph:

1. cycles involving only one process;
2. cycles resulting from feedforward edges; and
3. cycles resulting from feedback edges.

These three cycle classes stem from the three channel classes identified for the construction of edges in the PPN modeling graph that is discussed in Section 4.5.3.

Cycle	Cycle mean	Class	Feedback loop?
src → *src*	3	first	no
P1 → *P1*	9	first	no
P2 → *P2*	18	first	no
P3 → *P3*	24	first	no
src → *P1* → *src*	6	second	no
P1 → *P2* → *P3* → *P1*	51	third	yes

Table 5.1: Cycles in the PPN modeling graph for a PPN derived from Figure 5.5.

Cycles of the first class always originate from the selfloop added to the PPN modeling graph to eliminate auto-concurrency of a process. If the corresponding cycle mean is greater than T, then a period of T time units cannot be achieved due to sequential execution of all process iterations on a single processing resource. A limitation of the MCM analysis technique is that pipelined execution of multiple process iterations cannot be captured, because the MCM analysis technique does not incorporate the II value of a process. The actual execution time of a process may be lower than reported by the MCM technique if $II < \Lambda$ such that pipelined execution of process iterations is possible. The actual execution time of a pipelined process depends on the presence of selfloops in the original PPN. Such selfloops represent a feedback loop in which an iteration depends on the output of a previous iteration. We therefore consider a cycle of the first class as a feedback loop if the original PPN contains a selfloop for the process in the cycle.

Cycles of the second class originate from the backedges added to model finite buffer sizes. If the corresponding cycle mean is greater than T then the cycle represents a buffer whose size is too small to sustain period T. These cycle means can be prevented from being the maximum cycle mean by enlarging buffer sizes such that they do not affect performance. In Section 4.5.3, we have described how initial tokens on backedges can be chosen such that cycles of this second class never have the maximum cycle mean. We therefore ignore cycles of the second class when analyzing the PPN for feedback loops.

Cycles of the third class originate from cycles present in the original PPN. If the corresponding cycle mean is greater than T then the cycle represents a bottleneck inherent in the application. We therefore always consider a cycle of the third class as a feedback loop.

As an example, we consider the PPN shown in Figure 5.7a. The PPN modeling graph constructed from this PPN is shown in Figure 5.7b. The cycles in the PPN

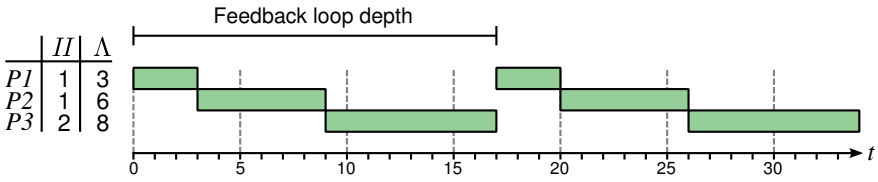


Figure 5.13: Flat execution profiles for the code of Figure 5.5.

modeling graph are listed in Table 5.1. All four cycles of the first class are not considered as a feedback loop, because the original PPN does not contain a selfloop for any of the processes in these cycles. As discussed above, cycles of the second class are never considered as a feedback loop. The PPN modeling graph contains one cycle of the third class, which is considered a feedback loop. We therefore conclude that a stream multiplexing might be beneficial and proceed to determine the maximum stream multiplexing factor.

Maximum Stream Multiplexing Factor

The maximum stream multiplexing factor is the maximum number of PPN executions that can be interleaved without increasing the latency of a single PPN execution. We illustrate how the maximum factor can be found using the flat execution profiles shown in Figure 5.13. These flat execution profiles are obtained by profiling the code of Figure 5.5 using cprof with the II and Λ values for each process shown in the left part of Figure 5.13. The factor is determined by the depth of the feedback loop and the II of the process functions involved in the feedback loop. We represent a feedback loop as a set of PPN channels $C \subseteq \mathcal{E}$.

The depth of a feedback loop C is the number of clock cycles since the start of the first process in the feedback loop until the next firing of the first process in the feedback loop. The feedback loop depth can be determined from the flat execution profiles obtained using cprof. For the flat execution profiles shown in Figure 5.13, we find that the feedback loop depth is 17 clock cycles. Alternatively, the feedback loop depth can be determined by analysis of the PPN. The dependence distance of a channel $(a \rightarrow b)$ represents the distance between process a and b as an iteration count. We use the channel size as a scalar approximation of the dependence distance, as motivated in Section 4.5.3. The sum of the dependence distances of the channels in the feedback loop gives the feedback loop depth expressed as an iteration count. To obtain the feedback loop depth $\mathit{depth}(C)$ expressed in terms of clock cycles, we multiply the size of each channel $c \in C$ with the latency of the process that writes to

c:

$$depth(C) = \sum_{c \in C} S_c \cdot \Lambda_{\sigma(c)},$$

with S_c being the size of channel c.

After determining the feedback loop depth, we compute the number of PPN executions that can be interleaved by dividing the feedback loop depth by the maximum II of all processes in the feedback loop. For the example of Figure 5.13, the maximum II of all processes is two because of process *P3*. Dividing the maximum feedback loop depth by the maximum II gives the number of independent executions of the feedback loop that can be interleaved. For the example of Figure 5.13, dividing 17 by 2 gives 8.5, which we round down to eight complete executions. Thus, a stream multiplexing transformation with a factor of eight can be applied to increase the throughput of multiple executions of the PPN, without increasing the latency of a single execution of the PPN.

5.2.4 Scheduling

Processes containing deeply pipelined IP cores may suffer from pipeline underutilization which limits throughput. Such underutilization is caused by a data dependency of the current iteration on a previous iteration that is still in the pipeline. Using the scheduling transformation presented in Section 5.1.4, the distance between dependent iterations can be altered, such that a higher throughput may be obtained. However, a scheduling transformation only increases throughput under certain circumstances, while it increases the control overhead of a LAURA processor in many cases. We therefore identify the following four criteria to assess the efficacy of a scheduling transformation on a process.

1. The purpose of a scheduling transformation is to increase pipeline utilization. Thus, the processor onto which a process is mapped should allow pipelined execution of process iterations. In terms of our implementation model of Definition 3.1, this means that $II < \Lambda$.
2. The process should have sufficient "room" for overlapped execution. Applying a scheduling transformation to a process which inherently executes its iterations in a fully sequential fashion will not improve performance.
3. The process should exhibit significant idling because of data dependences, causing the pipeline to be underutilized. Applying a scheduling transformation to a process that already yields full pipeline utilization will not improve performance.
4. The control overhead resulting from the new schedule should not cancel out

the performance gain of the new schedule.

Criterium 1 implies that the scheduling transformation is only effective for processes mapped onto LAURA processors. Overlapped, pipelined execution of process iterations is not possible on the programmable processors supported by ESPAM, such as the MicroBlaze, because their single-threaded instruction pipeline is too short to allow overlapped execution of process iterations.

Criterium 2 requires analysis of the application. The maximum iteration overlap that was introduced in Section 5.2.1 gives an upper bound on the number of iterations that can execute in an overlapped fashion. A maximum iteration overlap of one means that none of the iterations may execute in a partially overlapped fashion because the application is inherently sequential. In such a case, a scheduling transformation cannot improve overlapped execution, and thus should not be applied.

Criterium 3 can be evaluated in two ways: by analyzing the application code using cprof (cf. Section 4.6), or by analyzing a scheduled version of the application code using cprof. The first method is less accurate than the second, but is easier to perform because no changes to the application code have to be made.

To get a rough assessment of whether a scheduling transformation improves overlapped execution using the first method, we evaluate the original application code using cprof on both a real machine and an ideal machine. We assume a pipeline depth of four, that is, $\Lambda_{\mathrm{F}} = 4$ and $II_{\mathrm{F}} = 1$, meaning that up to four iterations can be active simultaneously. In Figure 5.14, we show the flat execution profile for the program of Figure 5.1 on the real and ideal machine. We observe that on the real machine, only one iteration is active for most of the time. On the ideal machine, on average two iterations are active. In both cases, the pipeline is underutilized, because a maximum iteration overlap of four dictated by the pipeline depth is not achieved. A scheduling transformation increases the average utilization from one to two simultaneously active iterations. We have verified using RTL simulation that a scheduling transformation on the program of Figure 5.1 indeed increases overlapped execution. As another example, consider the flat execution profiles of a 1D Jacobi kernel [BBK+08] in Figure 5.15. On the real machine, on average 7 iterations are active simultaneously. On the ideal machine, 29 iterations are active simultaneously. Although more overlapped execution occurs on the ideal machine, the average iteration overlap of seven on the real machine is already sufficient to keep the five-stage pipelined IP core of the application fully utilized. We have verified using RTL simulation that a scheduling transformation does not increase overlapped execution of the Jacobi application.

Alternatively, to get a more accurate assessment of the impact of scheduling on throughput using the second method, we evaluate a scheduled version of the application code using cprof. The scheduled application code can be obtained in a semi-

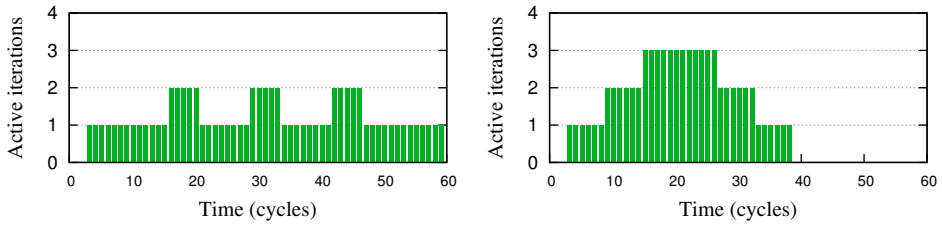


Figure 5.14: Flat execution profile for function `F` of the program shown in Figure 5.1.

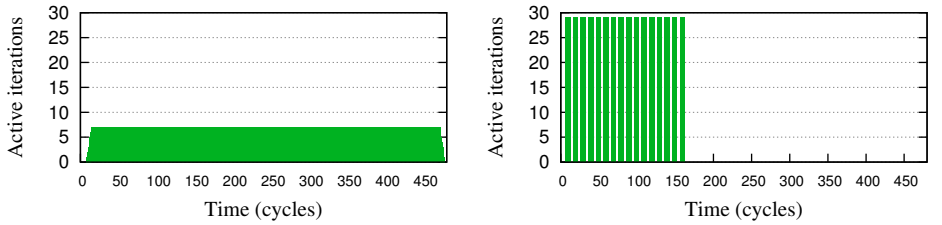


Figure 5.15: Flat execution profile for the Jacobi application.

automated way using for example CLooG [Bas04] or `isl` [Ver08]. By comparing the execution time of the original application code with the execution time of the scheduled application code, we quantify the effect of a scheduling transformation. For the example of Figure 5.14, we measure a decrease in execution time of 29%. For the example of Figure 5.15, we measure an increase in execution time of 56%, which means the scheduling transformation degrades performance. As a result of the analysis, we chose to apply the scheduling transformation for the example of Figure 5.14, but not for the example of Figure 5.15.

Criterium 4 is difficult to address at compile time, because the effects of the new schedule on control overhead are not known until time-consuming low-level synthesis and place-and-route steps have been performed. To avoid time-consuming synthesis steps, we use a heuristic to quickly determine if a particular schedule is likely to result in significant control overhead. A non-unit coefficient in a schedule leads to "gaps" in the transformed domain. For example, consider the polyhedral map of Equation (2.1) which has a coefficient of two for j_1. By applying this polyhedral map to the polyhedral set of Figure 2.2b, we obtain the transformed polyhedral set shown in Figure 2.3. Because of the non-unit coefficient, the transformed polyhedral set contains gaps in dimension j. To handle such gaps in the LAURA processor,

a division by the coefficient is required in the evaluation logic blocks. This is not a problem for coefficients that are a power of two, since division by such values can easily be implemented in RTL using bit shifts. For coefficients that are not a power of two, the resulting division may severely limit the maximum achievable clock frequency. Therefore, when Feautrier's algorithm computes a schedule with coefficients that are not a power of two, a scheduling transformation is not likely to yield higher throughput.

5.3 Conclusion and Summary

We have discussed four PPN transformations in this chapter: process splitting, process merging, stream multiplexing, and scheduling. We have presented how each of these transformations can be applied to a PPN in an automated fashion in the Daedalus tool flow. This enables a designer to quickly obtain functionally equivalent implementations of the same application that differ in performance and resource cost aspects.

Deciding when to apply any subset of the discussed transformations to obtain an implementation meeting a particular performance requirement is a nontrivial task for a designer. We leverage two techniques introduced in Chapter 4 to guide the designer in selecting the appropriate transformations and transformation parameters: the analytical MCM analysis technique and the profiling-based cprof technique. This enables a designer to systematically obtain an implementation that best matches a performance constraint.