Project no. 248828

ADVANCE

Strategic Research Partnership (STREP)
ASYNCHRONOUS AND DYNAMIC VIRTUALISATION THROUGH PERFORMANCE
ANALYSIS TO SUPPORT CONCURRENCY ENGINEERING

Design of protocol and interface for feedback mechanism to upstream levels

D21

Due date of deliverable: March 1^{st}, 2013
Actual submission date: February 27^{th}, 2013

Start date of project: February 1^{st}, 2010

Type: Deliverable
WP number: WP6
Task number: WP6d

Responsible institution: TWENTE
Editor & and editor's address:

Version 1.0 / Last edited by Robert de Groote / February 27, 2013

Project co-funded by the European Commission within the Seventh Framework Programme		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Revision history:

Version	Date	Authors	Institution	Section affected, comments
0.1	01/04/2013	Robert de Groote	TWENTE	Initial version
1.0	02/27/2013	Robert de Groote	TWENTE	Final version

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
WP6d	Feedback to higher levels in the tool chain	TWENTE*, UvA, USTAN, HERTS

°This task list may not be equivalent to the list of partners contributing as authors to the deliverable

*Task leader

Contents

Chapter 1

Introduction

This deliverable describes the details of the workflow of the placement and scheduling of an S-Net application. The workflow consists of a translation from S-Net into synchronous dataflow, and an exploration of the possible mappings onto a hardware platform for a near-optimal schedule.

1.1 Goal of this task

The goal of this task is to design the protocol for providing feedback to higher levels in the toolchain. With higher levels we mean the runtime system of S-Net (LPEL). Feedback to other levels such as the compiler or the designer is not addressed in this deliverable.

Feedback consists of proposed changes in the current schedule and placement employed by the running application, and is derived from observed behaviour regarding the resource usage by components and tasks.

Results delivered by this task will directly be used by LPEL, the runtime system of S-Net. Adopting the proposed schedule and placement changes will lead to a change in observed behaviour, which on its turn will result in updated feedback. In principle, this process is continued indefinitely and ideally converges to an optimal placement and schedule for the given hardware architecture.

1.2 Relation with other work packages

Work carried out in work package 6 deals with run-time resource management and the development of the *resource management layer*. Interaction between the resource management layer and other layers is illustrated in figure 1.1, which is described in the original project description. The resource management layer is responsible for analysing and evaluating the application's performance by observing the hardware usage. Statistical performance information that is

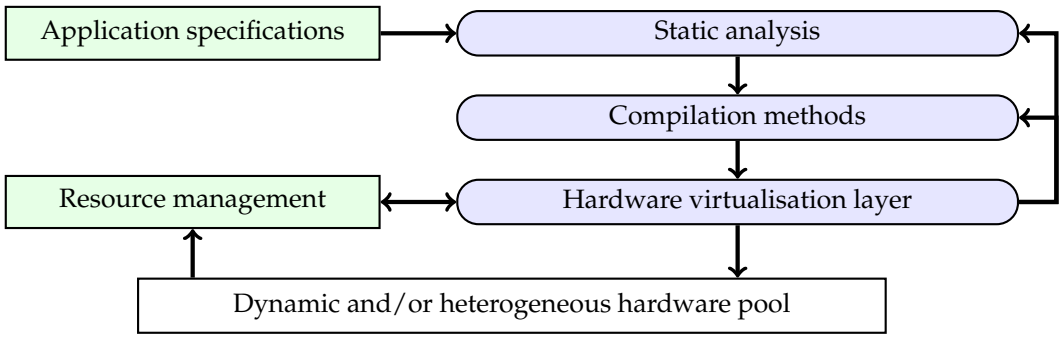


Figure 1.1: The Advance vision

compiled from the observed performance is fed back to higher levels through the hardware virtualisation layer, which is realised by LPEL.

Smart resource management is a key issue in the X-ray image processing use case that is developed in work package 7 and involves an application having highly dynamic timing behaviour. The other use cases are expected to benefit equally from the experience gained in this work.

Chapter 2

Application Modelling

The behaviour of an application modelled in S-Net is in principle non-deterministic[6]. Box specifications are single-in-single-out (SISO) with respect to *streams*, and have a *type* that specifies the records that are produced in response to a specific input record. Furthermore, the number of records that may be produced by a single box invocation is unknown beforehand (this number may even be zero).

This makes *timing analysis* of such applications rather complex. Especially when an application being modelled essentially has a fixed control flow, a stricter model that allows for better timing analysis is better suitable. In this chapter, we explore how an S-Net, in which the multiplicities of output records *are known a priori*, may be modelled as a synchronous dataflow graph. The purpose of the synchronous dataflow model is meant to allow for performance analysis. The synchronous dataflow model therefore only captures the timing behaviour of the original S-Net specification. Because, under the assumption of fixed record multiplicities, there is a one-to-one mapping from the S-Net specification to an abstract dataflow graph, a (near-)optimal schedule for a dataflow graph can be translated back into a (near-)optimal schedule for the S-Net application.

Note that the assumption of fixed record multiplicities is valid for all the use cases that are available in the project. We furthermore remark that non-determinism that cannot be captured as such in the synchronous dataflow paradigm may be modelled by the concept of *scenarios* (see below).

Choosing the dataflow model has as a primary advantage that long-term timing behaviour may be analysed efficiently in case task execution times are fixed, whereas in case of stochastic execution times, techniques are available to obtain bounds on the statistical moments of the performance. Bounds on measures such as minimum and maximum throughput, latency and jitter may be iteratively calculated from a linear time-invariant system of equations in max-plus algebra. Furthermore, as will be further explained in the Chapter 3, *task scheduling* and *placement* may be modelled in a straightforward way by expanding the application's max-plus system specification.

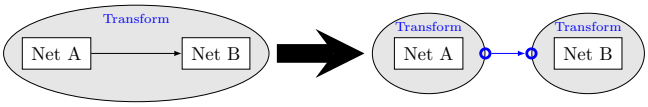


Figure 2.1: Transformation of serial composition into synchronous dataflow

This chapter details the recursive translation of an S-Net network into a synchronous dataflow graph. This translation is the first step in the workflow that provides feedback based on observed behaviour.

2.1 Building a dataflow model

In order to represent an S-Net network as a dataflow graph, we assume that multiplicities (the number of records that are produced by one box invocation) of output records is fixed and known beforehand. This has as a result that combinators such as serial and parallel replication can be analysed statically, as well.

Rather than constructing a dataflow model for the application, a model is constructed for the intermediate network representation (INR) of the S-Net [9]. As a result, all tasks that are executed at runtime are taken into account when exploring possible schedules. The following sections further describe the transformation of specific S-Net combinators and compositions into their dataflow counterparts.

2.1.1 Network Combinators

The S-Net network description language consists of a number of *network combinators*. These combinators are the essential construction principles in S-Net and combine networks into more complex networks. There are four combinators: *serial* and *parallel* composition of two (different) networks, and *serial* and *parallel* replication of a single network. For each of these combinators we define how they are treated in terms of transformations on the synchronous dataflow graph representations of their operands.

Serial composition

Serial composition is the simplest type of combinator found in S-Net. A serial composition of two networks (each of which may be as simple as a single box or synchrocell) connects the output stream of the left operand to the input stream of the right operand. In the runtime system, running a serial composition of two networks involves no extra housekeeping tasks. Serial composition can therefore be represented in a dataflow graph without much effort. See Figure 2.1 for a graphical illustration of the translation of serial composition into synchronous dataflow.

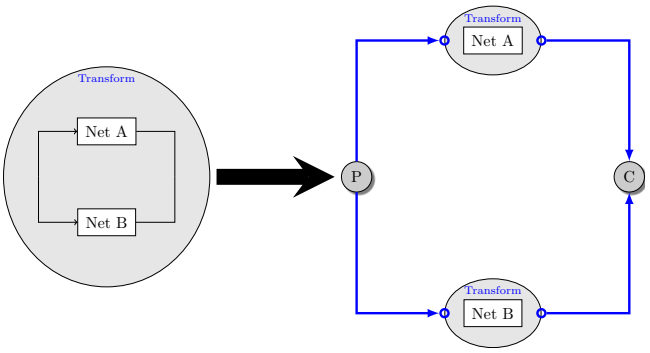


Figure 2.2: Transformation of parallel composition into synchronous dataflow

Parallel composition

Parallel composition of two networks combines the two networks (or boxes) in parallel. Any record that is sent into the parallel composed networks is sent to exactly one of the operand networks. The recipient of the record is determined by the type of the record and the types accecpted by the operand networks.

Such a parallel composition is impossible to precisely model in a synchronous dataflow graph. A synchronous dataflow graph specifies the number of records that are produced per invocation of a task such that deadlock and boundedness may be decided upon statically, whereas in an S-Net these multiplicities are unknown. We therefore slightly adapt the assumption made in S-Net and require that *scenarios* (see Section 2.2) may be identified such that in each scenario the record multiplicities are fixed. The non-determinism intrinsic to S-Net is then captured by the possibly non-deterministic transition between scenarios.

When scheduling the parallel composition of two networks in S-Net, two extra tasks (*Parallel split* (P) and *Parallel collect* (C)) are created by the S-Net runtime system. These two tasks take care of dispatching the records to the proper sub-networks, and of combining the output streams of each of the sub-networks into a single stream.

The dataflow representation of the parallel composition may thus be specified as shown in Figure 2.2.

Serial replication

The serial replication combinator is a unary operator, i.e. it is applied to a single network. The result of the operation is a network that is replicated infinitely many times, where the replicas are connected by serial composition. Records travel through the serially combined replicas of the network until a *termination pattern* is matched. Because it is not known a priori when this termination record is submitted, we can not precisely model this in a synchronous dataflow graph. We therefore, similar to the case of parallel composition, choose to move

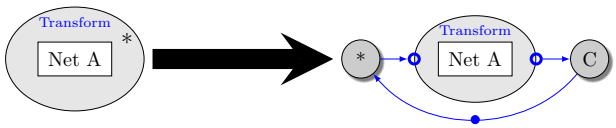


Figure 2.3: Transformation of serial replication into synchronous dataflow

the non-determinism to the transition between scenarios, and model a single replication statically as a synchronous dataflow graph that represents a single scenario.

Within a single scenario, the semantics of a star combinator in an S-Net are thus interpreted as those of a *feedback* combinator. In other words, it is assumed that star combinators are used to feed a *state* forward into the future invocation of the network.

Scheduling a star combinator involves a number of housekeeping tasks: the *star dispatcher* (denoted by *) and *star collector* (denoted by C) take care of the routing of records. The dataflow transformation may thus be stated as shown in Figure 2.3.

We remark that, where in the S-Net records produced by "Net A" are fed forward into the next replication of the net, in the dataflow graph the token represents that data is sent to the *next invocation* of the actor that represents the star dispatcher.

Parallel replication

We choose to not include this combinator in the transformation, as data-parallelism is captured by the box language. In Single-assignment-C (SaC), data-parallelism is exploited [13].

2.1.2 Boxes and System tasks

In the design of S-Net, boxes, filter boxes and synchrocells are the atomic units in the construction of complex networks. User-defined boxes that perform the actual computations may simply be represented by a single actor (task) in a synchronous dataflow graph. The same holds for filter boxes. Synchrocells are slighly different because they are stateful [5], and therefore they are modelled as a dataflow actor with a self-loop.

The runtime system creates a number of extra tasks to perform housekeeping, i.e., routing of records. These housekeeping tasks are collectively known as system tasks. Each of these systems tasks introduces extra latency in the network and must therefore be taken into account when scheduling the application. We therefore choose to include these system tasks in the dataflow model, by representing them with a single actor.

2.2 Modelling scenarios and timing behaviour

A dataflow graph merely describes the data dependencies that exist between tasks. In order to analyse the timing behaviour of the application modelled by the dataflow graph, it is necessary to include timing information, i.e., task latencies.

Traditionally, in the realm of realtime safety-critical systems, the purpose of timing analysis is to verify whether safety-critical constraints are met. In order to do so, worst-case estimates of task execution times are used, with worst-case guarantees on throughput and latency as a result.

However, the timing constraints in the ADVANCE project are typically less safety-critical and therefore demand a less pessimistic approach. The execution times of the actors in the dataflow graph are the probabilistic resource usage models derived from the monitoring information provided by the S-Net runtime system.

2.2.1 Scenarios

Scenarios capture the different processing steps that are taken depending on properties of the data. For example, in an MPEG decoder scenarios allow a designer to model the processing of progressive frames and initialisation frames separately. In terms of S-Net, these scenarios capture the properties that are not known at compile time, such as the number of records and their types that are produced by a single invocation of a box, or the number of replicas that are actually processed in a serially replicated network. Both functional and extra-functional properties may be captured in such a scenario. For example, the computation time required for a box to process a certain record may depend on (features of) the values carried by the record.

Scenarios may be derived or learnt from the monitoring data that is recorded from a dry-run of the application on the targeted hardware architecture. Statistical resource usage information is gathered from the monitoring subsystem of LPEL. The formal specification of the resource usage *models* lies outside the scope of this deliverable.

2.2.2 Timing behaviour

Timing behaviour includes both the specification of the (stochastic) execution time of a single task (i.e., S-Net box) in response to receiving a specific message, and the composition of these individual behaviours into the behaviour of the entire system.

The timing behaviour of individual tasks in the dataflow graph is specified on a per-scenario basis. This information is distilled form the statistical resource usage models provided by work package 4, which are built from the monitoring data provided by the runtime system of S-Net. Section 3.1 gives a detailed explanation on how the system's timing behaviour is analysed from the behaviours of the individual tasks.

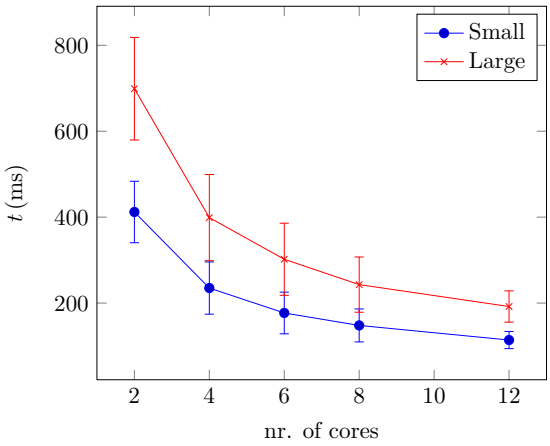


Figure 2.4: Latency of the feature tracking task in the Philips use case for two different scenarios (small versus large search area) with different number of cores used to exploit data-parallelism. Measurements were obtained for a stream of 500 images with fixed parameters.

2.2.3 Data-parallel boxes

Boxes that are imlemented in Single-assignment C (SaC) have a specific timing behaviour due to their dependency on the number of worker threads assigned to these boxes. In order to include these boxes in the dataflow model, these boxes are assigned probability distributions per execution scenario. Each execution scenario defines the number of assigned cores and the computational load. Using the statistical resource usage model we can then compute the expected latency from the number of assigned cores and computational load. Figure 2.4 illustrates the statistical resource usage of a data-parallel SaC box under different scenarios. The graph denoted "Small" was obtained by measuring the latency of the data-parallel feature tracker SaC box, applied to a search area of 700×700 pixels, whereas for the graph denoted "Large" the search area was 900×900 pixels.

2.3 Use case: X-Ray Image Processing

As an example case to illustrate how a scheduled S-Net is modelled by a dataflow graph, we will translate the S-Net of the use case from Philips into a dataflow graph. In the S-Net, record multiplicities are precisely 1. This means that the dataflow graph is in fact a homogeneous SDF (HSDF) graph.

The scenarios for this S-Net are the *initialisation phase* where all the init boxes (which produce a record that contains an initial state) produce one token, and the *streaming phase* where an (in principle) infinite data stream is processed. During the streaming phase, different scenarios capture the timing variations

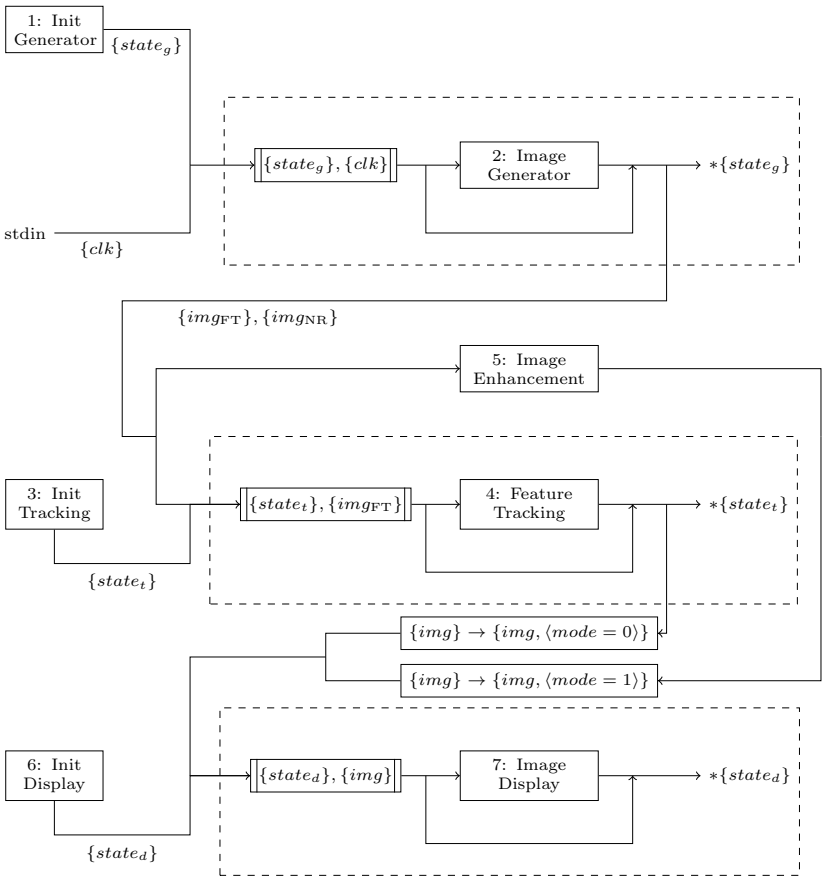


Figure 2.5: The S-Net specification of the Philips X-Ray processing use case

due to differences in the computational complexity of the feature tracking task. This computational complexity is predicted from a feature that is derived from the last few input images.

2.3.1 S-Net model

Figure 2.5 shows the S-Net that models the X-Ray image processing use case. The S-Net consists of the following 7 user-defined boxes:

1. Init Generator: provides the initial state for the image generation subnetwork.

2. Image Generator: reads X-Ray images from a source video and feeds them into the remainder of the network. Images are duplicated and sent to the image enhancement box in record field denoted 'img$_{\mathrm{NR}}$' (i.e., noise

reduction) and the feature tracking task in record field 'img_{FT}' (i.e., feature tracking).

3. Init Tracking: provides the initial state for the feature tracking subnetwork.

4. Feature Tracking: determines the location of features in the X-Ray image, using the previously known location as a heuristic. The output record is tagged by a filter with a mode (mode = 0) to indicate that the record contains a *coordinate*.

5. Image Enhancement: enhances the image by applying a noise reduction filter. The output record is tagged by the filter with mode = 1, indicating that the record contains an image.

6. Init Display: provides the initial state for the subnetwork that manages the display of images on screen.

7. Image Display: displays an image on screen.

There are three synchrocells (one synchrocell in each serially replicated subnetwork) in the S-Net model, denoted by double lines on the left and right hand sides. Each of these synchrocells ensures synchronisation of the next input record with the previous state of the subnetwork. The current state of each of the subnetworks is fed forward into the next replication of the network, as specified by the '*{state}' expression.

Configuration is provided through *stdin*, along with a clock signal (clk) that controls the rate at which X-Ray images are processed.

2.3.2 Synchronous dataflow model

The dataflow model corresponding to the S-Net specification of the X-Ray image processing use case is depicted in Figure 2.6. In this figure, the dataflow actors numbered 1 through 7 represent the 7 boxes specified in the S-Net model and described earlier.

The three synchrocells in the S-Net are represented by dataflow actors labelled with 'S'. The three serially replicated subnetworks result in three cycles in the dataflow graph. Each of this cycles connects the *star collector* with the corresponding *star dispatcher*. Also note that a number of dataflow actors labelled with '[ ]' occur in the dataflow graph. These actors model the filter boxes that are both explicitly (between the tracking subnetwork and the display subnetwork) or implicitly (e.g., the parallel bypass around the image generator user-defined box) defined.

Furthermore, there are a number of dataflow actors denoted with *ns*. These actors correspond to tasks executed by the S-Net runtime system, LPEL, and perform so-called *flow inheritance* (see [10, 7] for more details).

The synchronous dataflow graph explicitly models the three *initialisation boxes* numbered 1, 3 and 6. The boxes represented by these dataflow actors are

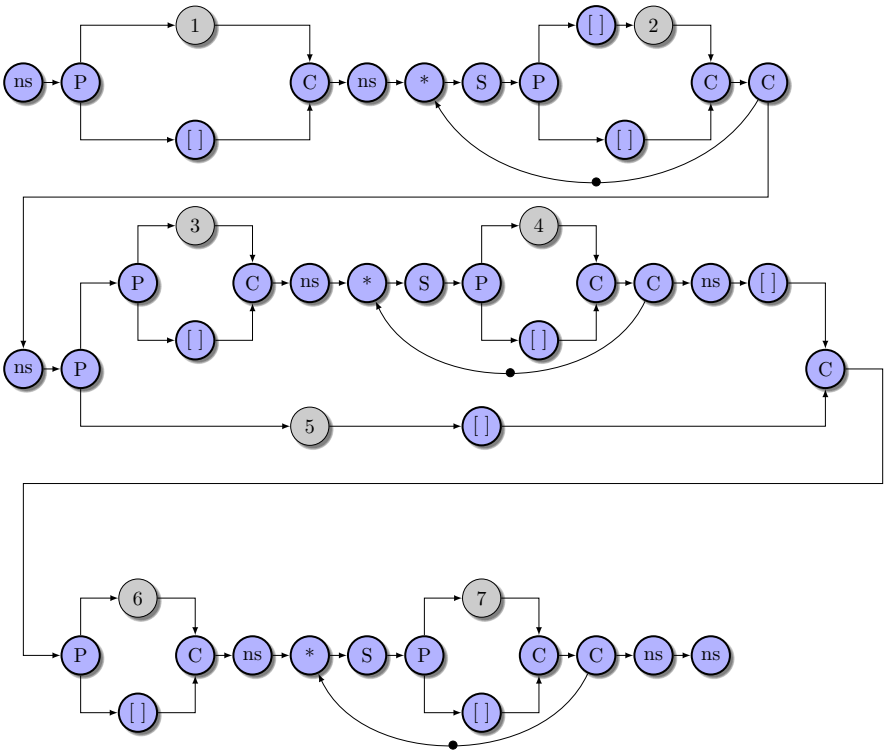


Figure 2.6: The synchronous dataflow graph representing the S-Net from Figure 2.5

executed only once (in the first, initialisation phase). As a result, the resource usage of these dataflow actors converges to the resource usage (i.e., their execution time becomes 0) of the, much longer, streaming phase.

Chapter 3

Placement and Scheduling

Performance analysis of a dataflow graph (or any timed synchronous sytem) involves modelling the precedence constraints as a linear time-invariant system in max-plus algebra. These precedence constraints follow from the data dependencies that exist between the tasks. Task schedules are essentially precedence constraints that are not specifically related to data dependencies: a schedule defines per core the set of tasks that are executed on that core, plus the order in which these tasks are executed.

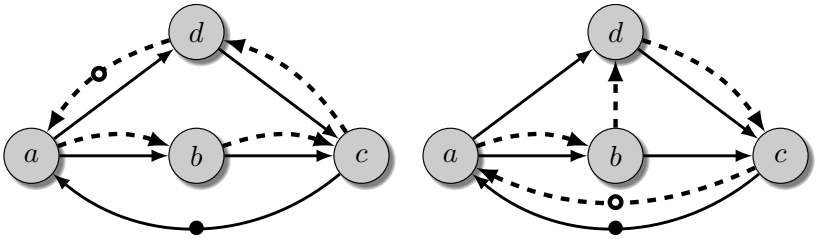
Because a (single scenario of a) dataflow graph represents a repeated computation that is applied to an infinite stream of data, it may be scheduled statically, with a static-order schedule. A schedule for a core may be represented in the dataflow graph by adding a cycle of *virtual data dependencies* (i.e., dataflow channels) to the homogeneous synchronous dataflow graph. Adding cycles to a dataflow graph may create deadlock as new circular dependencies are introduced. The following section therefore describes in more detail how core schedules are created.

3.1 Modelling schedules

In a dataflow graph, channels represent data dependencies. These data dependencies impose timing restrictions on the dependent tasks: a dependent task may not start before the dependee has finished and produced data. Data dependencies may be translated into precedence constraints in a straightforward way.

A (total) schedule is essentially a fixed, repeating sequence of task executions. Such a schedule may be represented as a set of 'virtual' channels in the dataflow graph. The virtual channels that belong to the schedule of the same core collectively form a cycle.

Cycles in a dataflow graph impose throughput constraints. The maximum achieveable throughput of the whole graph is determined by the graph's *critical cycle*. Adding cycles that represent core schedules may introduce extra cycles



(a) Deadlocked schedule (b) Live schedule

Figure 3.1: Two different schedules (indicated by the dashed arrows) that involve four actors in the dataflow graph. In both schedules actor a is the first one to execute, as indicated by the initial

beyond the cycle that represents the schedule. An example is shown in Figure 3.1, where the virtual channels that represent the schedule are dashed. The schedule in Figure 3.1(a) introduces two new cycles in the graph: ada and cdc.

Cycle ada contains a so-called *delay* on (virtual) channel da. This delay denotes a precedence constraint between the k^{th} invocation of d and the $(k+1)^{\text{th}}$ invocation of a. Cycle cdc creates a deadlock situation: actor d may not start its k^{th} invocation before actor c has completed its k^{th} invocation, which may not occur before actor d has completed its k^{th} invocation. Note that without the virtual cycle the graph is not deadlocked.

The deadlocked situation of Figure 3.1(a) is fixed in Figure 3.1(b), which enforces that both actor b and d have completed their k^{th} invocation before actor c may start its k^{th} invocation.

Cases where deadlock is introduced may be avoided easily by considering the subgraph of the dataflow graph that is induced by the edges that contain no delays or tokens. This graph is sometimes referred to as the *acyclic precedence expansion graph* [11].

For example, consider the acyclic precedence expansion graph of the dataflow graph that was depicted in Figure 3.1. There exist two possible topological ordering of the actors: $abdc$ and $adbc$. There are thus two possible live schedules (for a single core), one of which was shown in Figure 3.1(b). The other schedule is obtained by swapping actors b and d.

We may thus always construct a periodic schedule for any set of actors on a single core by following the steps listed above.

1. Construct the graph's acyclic precedence expansion graph (APEG).
2. Construct a valid topological ordering of the APEG
3. Apply the topological ordering to order the set of n actors $v_1 \dots v_n$ that need to be scheduled.
4 . Add (delay-less) edges $v_i v_{i+1}$ to the dataflow graph.

5. Add edge $v_n v_1$ with a single delay to the dataflow graph.

After a schedule for one core has been constructed, the APEG is updated with the new tokenless edges that are due to the schedule. Periodic schedules for multiple cores may thus be constructed iteratively. It is thus always possible to schedule any partitioning of the dataflow graph.

3.2 Optimising for performance

For a given dataflow graph, the size of the set of possible schedules onto an n-core hardware platform is too large to be searched entirely for the schedule that has best performance. Furthermore, the problem of finding the optimal schedule is not a *convex* optimisation problem in that we may start with some solution, which is then improved by pivoting.

Because of this, we employ the popular and powerful technique of *simulated annealing* to obtain a near-optimal solution from the huge search space. In this search, each possible schedule is analysed for its performance. Perfomance may be any metric that formally captures the costs of various factors and represents its with some ordinal number. In the use cases that are targeted within the ADVANCE project, we mainly use throughput as a measure. Throughput is a long-term performance metric and expresses the average amount of data that is processed by the system per time unit.

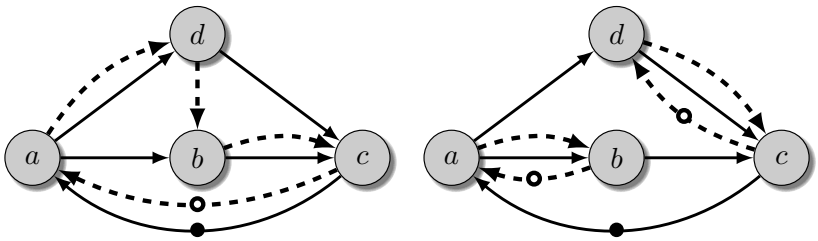
For synchronous dataflow graphs, throughput analysis consists of computing the steady-state behaviour. In case execution times are fixed (e.g., worst-case timing analysis), this may be done by either exploring the state-space of a simulated execution, or through the analysis of the linear time-invariant max-plus system, which represents the dataflow graph, for its eigenvalue [8, 2, 1, 3]. In case execution times are in fact stochastic variables, bounds on the throughput may be obtained by an iterative method that is based on the subadditive ergodic theorem [4, 14].

3.2.1 Constructing an initial schedule

In a previous section it was shown that we may always (i.e., for a deadlock-free dataflow graph) construct a valid schedule for any subset of actors and run this schedule on a single core. Given a target architecure that has n cores, we may thus construct an initial schedule by partitioning the set of actors into n proper, disjoint, subsets. In the remainder of this section we assume that an initial schedule was constructed.

3.2.2 Finding a near-optimal schedule

A scheduled dataflow graph may be analysed for its maximum achieveable throughput. Searching the vast space of possible schedules is infeasible. We therefore employ simulated annealing to find a near-optimal solution.



(a) Changing the scheduling order (b) Moving tasks to another core

Figure 3.2: The two possible adjustments that may be made to the schedule in Figure 3.1(b)

Simulated annealing attempts to minimise the costs by making random adjustments to the current candidate solution. Decreases in costs are always accepted, and increases in costs are accepted probabilistically. The likelihood of accepting an increase in costs depends on the temperature[1]. Initially, temperature is high and as a result, local optima are avoided by allowing the search to visit worse mappings. As temperature decreases, the search process is more and more confined to a local optimum.

In the approach that we take, there are two possible adjustments that can be made: First of all, an actor can be *moved* from one core to another. This involves removing the actor from the schedule's core and inserting it in the schedule of the target core. Care must be taken that no deadlock is introduced when doing this. Figure 3.2(b) shows the schedule that is obtained by moving actors a and b to another core, while actors c and d remain on the same core.

The second possible adjustments is a (random) rearrangement of a core's schedule. This involves inspecting the topological ordering of the involved actors for possibilities to swap two actors. Note that this is not always possible. Figure 3.2(a) is the result of swapping actors b and d in the schedule shown in Figure 3.1(b).

The two possible adjustments to the schedule allow the simulated annealing process to traverse the search space. In each step of the simulated annealing process, a random choice is made to either perform a rearrangement of some schedule, or a reassignment of a random actor to another core. In case rearranging is not possible, a reassignment is chosen instead.

Each adjustment that is made yields a new schedule, with changed performance behaviour. For the newly obtained scheduled graph, maximum throughput is estimated by exploring the statespace of a (self-timed) execution of the graph, combined with superposition steps as described in [4, 14]. After a number of iterations this yields a confidence interval for the scheduled graph's maximum throughput. The maximum throughput is then estimated pessimistically, i.e. the lowerbound of the confidence interval is taken as a conservative

[1]Temperature is a well-established concept in the theory of simulated annealing[12]

(safe) estimate of the throughut.

The pessimistically estimated maximum throughput is translated to a cost metric by the cost function, where higher throughput means lower costs. Costs may have increased due to the adjustment that was made. In order to prevent the search to get stuck in local cost minima, mappings with an increased cost are accepted at a probability that depends on the temperature. As temperature decreases, the likelihood of accepting worse (i.e., solutions with higher costs) mappings decreases as well.

The pseudocode for the simulated annealing search is listed in Figure 3.3. The search attempts to find the best (maximum throughput) mapping of the dataflow graph onto N cores. The temperature at which the search starts is given by T_0. At each temperature level, L adjustments are made, each of which may or may not lead to a revised mapping M. After making these L adjustments, temperature is lowered by a factor of k. The search stops as soon as a minimum temperature T_{stop} is reached.

The semantics of the functions *initial mapping*, *move* and *estimated-costs* are as follows: *initial mapping* gives a random, initial mapping of the dataflow graph onto N cores. *Move* makes an adjustment to the current mapping M by either rearranging the schedule of a single core or reassignment of a dataflow actor to another core, both of which have been explained earlier in this section. Finally, *estimated-costs* conservatively estimates the maximum throughput, as described above.

3.3 Example: scheduling the Philips use case

As an example, we have applied simulating annealing to explore the possible placements and per-core schedules of the X-Ray image processing use case, mapped onto a shared-memory multicore architecture using 2 cores. The timing behaviour was retrieved from the monitoring data that was recorded for 5 minutes.

Simulated annealing was set up with the following parameters: the initial temperature, T_0 was set to 1.0. At each temperature, $L = 1000$ moves (adaptations) were made, after which the temperature was decreased by 10%, until the temperature reached $T_{\text{stop}} = 0.1$. Figure 3.4 shows the resulting mapped dataflow graph. The 2 dashed red and blue cycles represent the two core schedules.

simulated-annealing-mapper(G, T_0, T_{stop}, k, N, L)
$C_{\textbf{opt}} := \infty$
$M_{\textbf{opt}} :=$ undefined
$T := T_0$
$M :=$ **initial mapping**(G, N)
$C :=$ **estimated-costs**(M)

While $(T > T_{\text{stop}})$ **do**
  For $i = 1 \ldots L$ **do**
    $M' :=$ **move**(M)
    $C' :=$ **estimated-costs**(M')
    $\Delta C := C' - C$
    If $(C' < C)$or $\left(\frac{1}{1+e^{\frac{10(C'-C)}{T}}} > \text{rnd}() \right)$
      $C := C'$
      $M := M'$
    endif
    If $(C < C_{\text{opt}})$
      $C_{\text{opt}} := C$
      $M_{\text{opt}} := M$
    endif
  endfor
  $T := T \cdot k$
endwhile
return M_{opt}

Figure 3.3: The simulated annealing-based search for a near-optimal mapping of a synchronous dataflow graph G onto N cores.

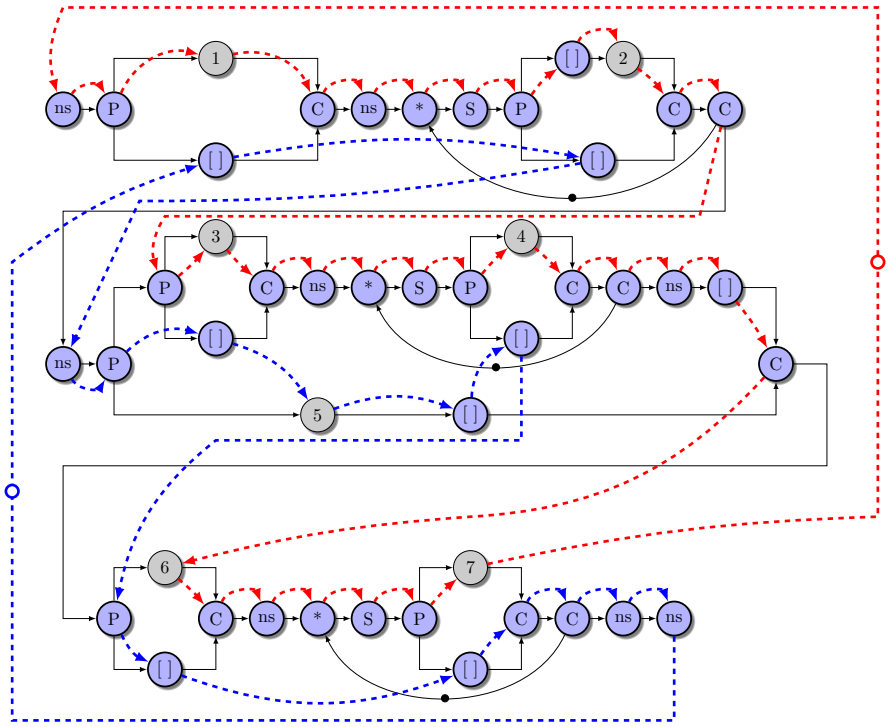


Figure 3.4: The synchronous dataflow graph from Figure 2.6 mapped onto 2 cores

Chapter 4

Feedback to higher layers

Whenever an updated mapping (i.e., placement of tasks onto cores and one schedule per core) for an S-Net application is found, the information is made available to the higher layers. This information consists of a repeating, static schedule per core, as described in the previous chapter.

4.1 Integration with the LPEL layer

The runtime system of S-Net, LPEL, is responsible for the scheduling of an S-Net onto a (shared memory) multicore hardware platform. This involves each of the S-Net's boxes and synchrocells, as well as the routing tasks such as the collectors and dispatchers involved in the parallel and star combinators. Each of these mapped entities is called a *task*.

Because each task is represented by a single actor in the (scheduled) dataflow graph, the scheduled dataflow graph contains all the information necessary to allow LPEL to place tasks on the proper cores.

Each of the tasks that needs to be scheduled by the runtime system is included in the dataflow model of the S-Net application. In order to associate timing behaviour with a box or synchrocell, the corresponding tasks need to be identified from the logfiles provided by the runtime system's monitoring subsystem. This is done through the *topological identity* of a task: because each S-Net has a single input and single output, each task is uniquely identified by a path from the global input. This path is represented as a sequence of literals. For example, "S5P1R12S1" denotes the fifth network in serial composition, in which the first branch is taken in parallel composition, the 12th replica in serial replication and finally the first replica in serial composition.

In the dataflow graph, we assign each actor with the topological identity of the corresponding task. This allows us to associate monitored information with specific actors. Note that multiple tasks may map to the same actor in the synchronous dataflow graph.

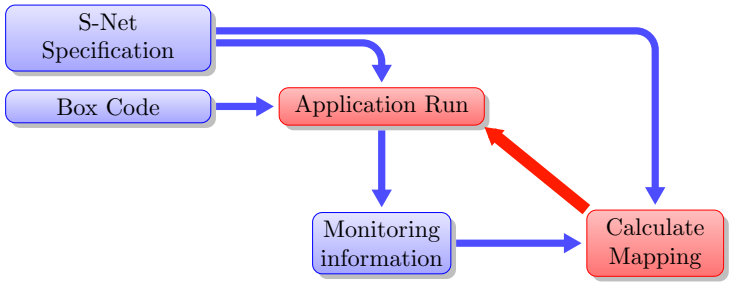


Figure 4.1: The workflow to complete the feedback loop in order to improve an observed S-Net application run.

Now that LPEL tasks can be associated with dataflow actors through their topological identity, the mapping information can be readily made available to the LPEL layer through the following 2 functions specified in the C programming language:

```
int core_schedule(const int core_id, char **tasks);
int affinity(const char *task);
```

4.2 Workflow

The translation of S-Net into synchronous dataflow, complemented with the monitored execution time profiles gathered from the LPEL worker logs, and finally the application of the simulated annealing-based search, collectively yield the workflow shown in Figure 4.1. Each mapping that is adopted by the runtime system of S-Net yields new behaviour that is stored in the monitoring information. This information may give rise to changes in the proposed mapping, resulting in a new traversal of the feedback loop.

Bibliography

[1] Guy Cohen, Stéphane Gaubert, and Jean-Pierre Quadrat. Max-plus algebra and system theory: Where we are and where to go now. *Annual Reviews in Control*, 23:207–219, January 1999.

[2] Guy Cohen, Geert Jan Olsder, and Jean-pierre Quadrat. *Synchronization and linearity*. Wiley New York, 1992.

[3] Robert de Groote, Jan Kuper, Hajo Broersma, and Gerard J.M. Smit. Max-Plus Algebraic Throughput Analysis of Synchronous Dataflow Graphs. In *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*, pages 29–38. IEEE, September 2012.

[4] Rob M.P. Goverde, Bernd Heidergott, and Glenn Merlet. A fast approximation algorithm for the Lyapunov exponent of stochastic max-plus systems. In *2008 9th International Workshop on Discrete Event Systems*, pages 49–54. IEEE, 2008.

[5] C. Grelck. The essence of synchronisation in asynchronous data flow. In *25th IEEE International Parallel and Distributed Processing Symposium (IPDPS'11), Anchorage, USA*. IEEE Computer Society Press, 2011.

[6] C. Grelck, Shafarenko, A. (eds):, F. Penczek, C. Grelck, H. Cai, J. Julku, P. Hölzenspies, Scholz, S.B., and A. Shafarenko. S-Net Language Report 2.0. Technical Report 499, University of Hertfordshire, School of Computer Science, Hatfield, England, United Kingdom, 2010.

[7] Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. A Gentle Introduction to S-Net: Typed Stream Processing and Declarative Coordination of Asynchronous Components. *Parallel Processing Letters*, 18(2):221–237, 2008.

[8] B. Heidergott, Geert Jan Olsder, and Jacob van der Woude. *Max Plus at Work: modeling and analysis of synchronized systems*. Princeton University Press, 2006.

[9] Raimund Kirner, Clemens Grelck, Frank Penczek, and Alex Shafarenko. D11 report describing vr-net and interfaces, May 2011.

[10] Daniel Prokesch. A light-weight parallel execution layer for shared-memory stream processing. Master's thesis, Technische Universität Wien, Vienna, Austria, Feb. 2010.

[11] Sundararajan Sriram and Shuvra S. Bhattacharyya. Embedded Multiprocessors: Scheduling and Synchronization. February 2009.

[12] Peter JM Van Laarhoven, Emile HL Aarts, and Jan Karel Lenstra. Job shop scheduling by simulated annealing. *Operations research*, 40(1):113–125, 1992.

[13] V.Wieser, B. Moser, S. Scholz, S. Herhut and J. Guo, editor. *Combining High Productivity and High Performance in Image Processing Using Single Assignment C*, volume SPIE 8000. Proceedings of SPIE - The International Society for Optical Engineering, 2011.

[14] Xiao-Lan Xie. Superposition properties and performance bounds of stochastic timed-event graphs. *IEEE Transactions on Automatic Control*, 39(7):1376–1386, July 1994.