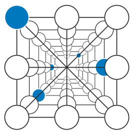


Project no. 248828

ADVANCE

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

Tool for Dynamic Placement
D28

Due date of deliverable: September 30, 2013
Actual submission date: November 3, 2013

Start date of project: February 1st, 2010

Type: Deliverable
WP number: WP6
Task number: WP6e

Responsible institution: TWENTE
Editor & and editor's address: Jan Kuper
University of Twente
Department of EEMCS
7500 AN Enschede, The Netherlands

Version 1.0 / Last edited by Robert de Groote / November 3, 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	09/16/2013	Robert de Groote	Twente	Initial version
0.6	10/31/2013	Robert de Groote	Twente	First revision
0.7	11/02/2013	Jan Kuper	Twente	Second revision
1.0	11/03/2013	Robert de Groote	Twente	Final version

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
WP6e	Dynamic Placement	HERTS, HWU, TWENTE*, UvA

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

*Task leader

Executive Summary

This document describes the final version of the dynamic placement tool, as developed within work package 6 in the ADVANCE project. The document gives a brief overview of the theoretical aspects used to compile the tool and describes its place in the overall workflow. Finally, this document describes the application programmer's interface of the software that makes up the placement tool.

Contents

Executive Summary 1

1 Introduction **3**
1.1 Dynamic placement workflow 4
1.2 Workflow Implementation and Document Outline 4

2 Application modeling using synchronous dataflow graphs **6**
2.1 From S-Net to Synchronous Dataflow 6
2.1.1 Serial composition 7
2.1.2 Parallel composition 7
2.1.3 Serial replication 8
2.1.4 Parallel replication 8
2.2 Implementation 8

3 Finding good placements **11**
3.1 Summary of the placement approach 11
3.2 Implementation of scheduling 13
3.3 Simulated annealing 15
3.4 Implementation of the Simulated Annealing Mapper 16

4 Concluding Remarks **19**

Chapter 1

Introduction

Multi-core hardware platforms allow applications that are inherently parallel to be run efficiently. In case the number of cores is limited, however, finding a near-optimal schedule of the application's subtasks involves the clever balancing of sequentialization of less critical tasks on the one hand, with exploiting task or data-level parallelism other hand. This scheduling task is by far not straightforward, as many possible feasible schedules may exist.

Scheduling consists of two main tasks: *placement*, which is the assignment of tasks to a processing element where it is to be executed, and *task ordering*, which determines the order in which tasks are executed locally on a processing element. Task ordering has to obey the *data dependencies* that exist in the application; if task B computes a funciton on data provided by task A, then task A should be scheduled to execute before task B.

Data dependencies and task durations are the two main inputs to scheduling. Dependencies are (usually) fixed, and give the *control flow* of the application. Task durations are typically unknown beforehand, and are learnt when tasks are actually run on the target hardware platform. Durations of tasks may depend on many factors, such as cache misses, arbitration on the memory bus, features of the data, etc. The dynamic placement approach taken in the ADVANCE project tackles this variation in task duration by observing the task durations while the application is running, and using these observations when deciding an updated placement.

In order to reason about the performance of an application that is to be placed onto a hardware platform, an application model is needed. Because each of the use cases in the ADVANCE project is a *stream processing* application, and since the language used to specify the coordination between an application's tasks is S-Net, which models applications as stream processing networks, we have used the well-established *synchronous dataflow* (SDF) model [22] as our input application model. A strong advantage of using SDF as an underlying model, is that performance analysis can be computed relatively easy. Furthermore, it is straightforward to model a statically ordered schedule of such an SDF graph in that same representation.

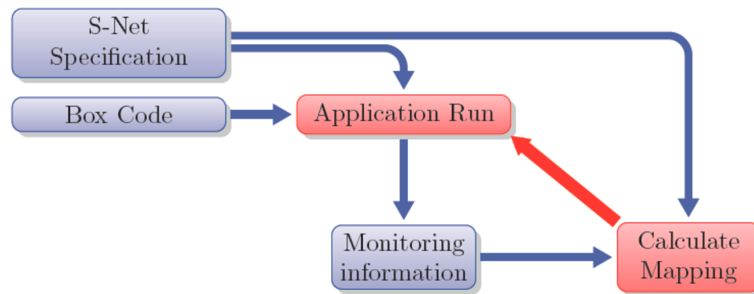


Figure 1.1: The workflow to complete the feedback loop in order to improve an observed S-Net application run.

1.1 Dynamic placement workflow

The workflow of the dynamic placement tool is as follows: the application that is to be dynamically placed onto a hardware platform, implemented as an S-Net, is modelled as an SDF graph. This is an automated process, the details are provided in Chapter 2.

Both placement of the SDF graph's tasks, and the ordering of tasks per processing element, is modelled by extra depencies that are added to the SDF graph. Each schedule that is constructed in this way is analysed for its performance by running a state-of-the-art algorithm on the graph.

After computing the performance of the scheduled application, changes are made to the schedule with the aim to obtain a better performance. Simulated annealing [29] is used to guide this process; as the number of explored schedules increases, the deviation allowed from the best schedule found so far decreases, such that the search converges to a near-optimal solution.

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 1.1. 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.

1.2 Workflow Implementation and Document Outline

The workflow described in the previous section has been implemented in the Java programming language. The different aspects of dynamic placement and scheduling are distributed over a number of Java classes, which are described in further detail in the following two chapters. A textual description of the abstract syntax tree of the S-Net application to be mapped, as well as a recorded logfile with data monitored by the LPEL subsystem [25], serve as the input of the workflow. These

two inputs are compiled into a synchronous dataflow representation as a first step. This SDF representation is stored in a file, in the JSON (Javascript Object Notation) format. Chapter 2 describes this first step in further detail. The second step in the workflow is the search for a near-optimal schedule, which is implemented by a single Java class, described in Chapter 3.

The outline of this document follows the outline of ADVANCE deliverable D21 [6], which described the protocol and interface to provide feedback to higher levels in the toolchain. Technical details concerning placement and performance analysis will be less elaborate, the purpose of this document is to link the functional specifiation, as has been covered largely in D21, with the technical implementation given in this document.

Chapter 2

Application modeling using synchronous dataflow graphs

In this chapter we discuss the transformation of an S-Net specification to a dataflow graph (Section 2.1), and the implementation thereof (Section 2.2). Here we only shortly mention some steps in the transformation, in deliverable D21 the underlying assumptions are discussed in detail. Execution times of the actors in the dataflow graphs needed for the analysis of the graphs are extracted from the log-files that are generated by the S-Net run-time systems.

2.1 From S-Net to Synchronous Dataflow

In order to be able to transform an S-Net specification into a synchronous dataflow graph, and thus to be able to analyse the various combinators of S-Net statically, we make certain assumptions, as described in deliverable D21. These assumptions include assumptions on the multiplicity of output records from networks and on the possibility to deal with non-determinism in an S-Net specification by means of scenarios. These assumptions are sufficient to allow for a large class of S-Net specifications to be transformed into SDF graphs, for example, the X-Ray application of Philips can be dealt with in this way.

Since system tasks introduced by the S-Net run-time system introduce extra latency, we choose to transform the *intermediate network representation* (INR) of an S-Net specification into SDF. This gives rise to the occurrence of nodes like *parallel split* and *parallel collect* in the resulting SDF graph.

The transformation is recursively defined, where the atomic units in the construction of an S-Net network are user defined boxes (computational boxes and filter boxes) and synchrocells. Boxes may simply be represented by a single actor (task) in a synchronous dataflow graph, whereas synchrocells are modelled as a dataflow actor with a self-loop.

The recursive clause of the transformation of S-Net into SDF is consists of the definition of the transformation for the S-Net combinators. The four essential

combinators in an S-Net network description are: *serial* and *parallel* composition of two (different) networks, and *serial* and *parallel* replication of a single network. In the following sections we give the transformation rule for three of these four S-Net combinators by means of a graphical representation, assuming that the pictures are self-explanatory. The fourth combinator (parallel replication) is taken care of by the parallelism in the box language (Single Assignment C, SAC). In case further explanations are needed, we refer to deliverable D21.

2.1.1 Serial composition

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. The transformatuon of serial composition into synchronous dataflow is depicted in Figure 2.1.

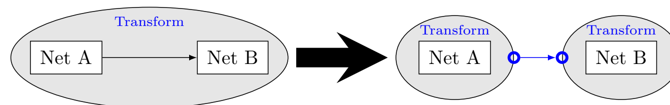


Figure 2.1: Transformation of serial composition into synchronous dataflow

2.1.2 Parallel composition

Parallel composition of two networks combines the two networks (or boxes) in parallel, and a record that is sent into the parallel composed networks is sent to exactly one of the operand networks. Since this choice may depend on teh content of the data, it is at this point that scenarios are needed to transform a parallel network into an SDf graph. Furthermore, for parallel composition the S-Net runtime system introduces two extra tasks to *split* (P) and to *collect* (C) records.

Figure 2.2 shows the dataflow transformation of parallel composition in S-Net.

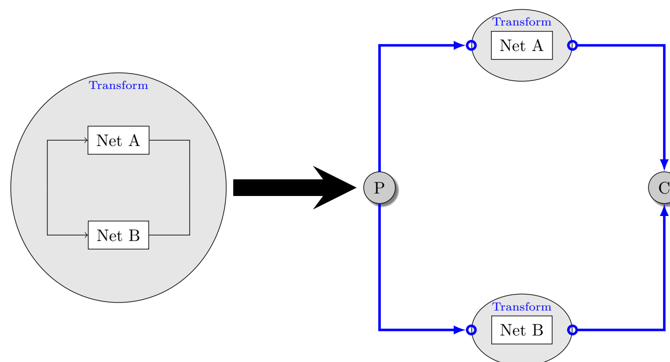


Figure 2.2: Transformation of parallel composition into synchronous dataflow

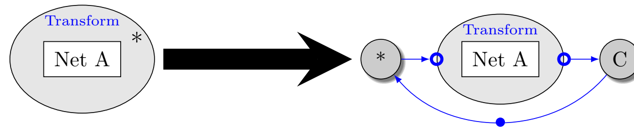


Figure 2.3: Transformation of serial replication into synchronous dataflow

2.1.3 Serial replication

The serial replication combinator is applied to a single network, resulting in a network that is serially replicated infinitely many times. Here too, scenarios are used to capture for the fact that answering the *termination pattern* of a record is data dependent. A *star dispatcher* (*) and *star collector* (C) take care of the routing of records.

Figure 2.3 shows the dataflow transformation of serial replication in S-Net.

2.1.4 Parallel replication

As mentioned above the combinator of parallel replication is captured by the data paralellism of the box language (Single assignment C).

2.2 Implementation

The recursive transformation of an S-Net into a synchronous dataflow graph is implemented by two Java classes. The first class, `SNetASTTransformer`, which is listed in Listing 2.1, is responsible for the actual recursive transformation. In this class, the protected method `singletonHSDF` takes care of the atomic units in the construction of an S-Net network, i.e.: user defined boxes and synchro-cells. The other three protected methods `starToHSDF`, `parallelToHSDF` and `serialToHSDF` implement the transformation of respectively serial replication, parallel composition and serial composition.

The public method `toHSDF(root)` is the entry point to transform an S-Net AST, rooted in root, into a synchronous dataflow graph.

Listing 2.1: Class SNetTransformer

```
public class SNetTransformer {

    public SNetTransformer(Random rnd);

    public DirectedGraph<TimedActor, HSDFChannel>
        toHSDF(ASTNode root);

    public DirectedGraph<TimedActor, HSDFChannel>
        toHSDF(ASTNode root, StringBuilder path);

    protected DirectedGraph<TimedActor, HSDFChannel>
```

```
        singletonHSDF(ASTNode root, StringBuilder path);

    protected DirectedGraph<TimedActor, HSDFChannel>
        starToHSDF(ASTNode root, StringBuilder path);

    protected DirectedGraph<TimedActor, HSDFChannel>
        parallelToHSDF(ASTNode root, StringBuilder path);

    protected DirectedGraph<TimedActor, HSDFChannel>
        serialToHSDF(ASTNode root, StringBuilder path);
}
```

The actual processing of an input S-Net network and its observed timing information, as recorded by the LPEL monitoring subsystem into log files, and the compilation of these inputs into an output synchronous dataflow graph, is implemented by the class `SNetASTParser`, listed in Listing 2.2. This class has a main method, which allows the class to be executed on input files, in the following way:

```
java -cp . SNetASTParser network-ast.txt n00_tasks.map
```

The first input file expected by the class's main method is a text file containing the abstract syntax tree of the target S-Net network. The second input file is the *task mapping* file, which is produced by the monitoring subsystem of LPEL when the S-Net is run.

When the input files are processed successfully, the execution produces an output text file, which contains the synchronous dataflow graph model, stored in the JSON format. This output file will be used as the input for the mapping (simulated annealing) process, to be described in Chapter 3.

Listing 2.2: Class SNetASTParser

```
public class SNetASTParser {

    /**
     * Constructors: parse a file
     *
    **/
    public SNetASTParser(String filename)
        throws IOException, ASTParseException;

    public SNetASTParser(File file)
        throws IOException, ASTParseException;

    /**
     * returns the parseTree of the parsed SNet AST
    **/
    public ASTNode getParseTree();

    /**
     * Arguments:
     *   path to S-Net AST file (*-ast)
```

```
     *   path to Task Mapping file produced by LPEL (*.map)
     *
     * Output:
     *   JSON file containing timed SDF graph
    **/
    public static void main(String[] args) throws Exception;

    /**
     * Helper class around the abstract syntax tree
    **/
    class ASTNode {
        public ASTNode(String name, List<String> args,
            ASTNode parent);
        public ASTNode getParent();
        public String getName();
        public List<String> getArgs();
        public List<ASTNode> getChildren();
        public String toString();
    }
}
```

A developer that needs access to the parsed S-Net may do so simply by constructing an instance of `SNetASTPArser`, passing a reference to the file (either the file's name or a `File` object to the constructor). After constructing the instance, the internal representation of the abstract syntax tree of the S-Net is obtained by calling `getParseTree()`. The (internal) class `ASTNode` (see Listing 2.2) can be used to traverse this abstract syntax tree.

Chapter 3

Finding good placements

In this chapter we present the implementation of the placement tool, focusing on two aspects: scheduling and optimization, where for the second aspect we choose simulated annealing as optimization technique. As described in deliverable D21 and also shortly discussed in Chapter 2 of the present deliverable, we start from the dataflow representation of the intermediate network representation (INR) of an S-Net specification. Below we first give a short summary of the placement approach (Section 3.1), followed by the implementation of scheduling (Section 3.2). Then we give a short sketch of the usage of simulated annealing in our context (Section 3.3), followed by the implementation of simulated annealing for placement optimization (Section 3.4). We conclude this chapter with a presentation of the feedback mechanism.

For a more detailed description of the methods discussed in this chapter, we refer the reader to deliverable D21.

3.1 Summary of the placement approach

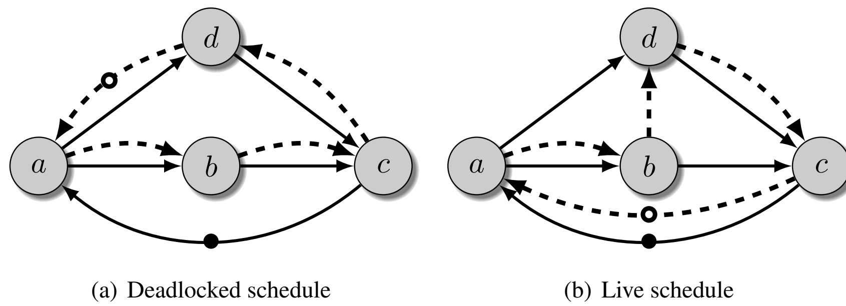
We model the precedence constraints in the graph that results from transforming an S-Net specification towards a dataflow graph as a linear time-invariant system in max-plus algebra. We then analyse the performance of the system in the framework of the max-plus specification.

A (total) schedule is essentially a fixed, repeating sequence of task executions and may be represented as a set of 'virtual' channels in the dataflow graph (the dashed arrows in the examples in Figures 3.1 and 3.2). Such a collection of virtual channels together forms a cycle, thus imposing throughput constraints. Deadlock arising from adding these virtual channels can be avoided by considering the subgraph (the "acyclic precedence expansion graph", APEG) of the dataflow graph that is induced by the edges that contain no delays, where a *delay* on a channel (data dependency) ab denotes a precedence constraint between the kth invocation of a and the $(k{+}1)^{th}$ invocation of b. We remark that the maximum achieveable throughput of the whole graph is determined by the graph's *critical cycle*.

As described in deliverable D21, a valid schedule may be constructed for any subset of actors so that these actors can be run on a single core according to this schedule. Given a target architecure that has n cores, we may thus construct an initial schedule by partitioning the set of actors into n disjoint proper subsets.

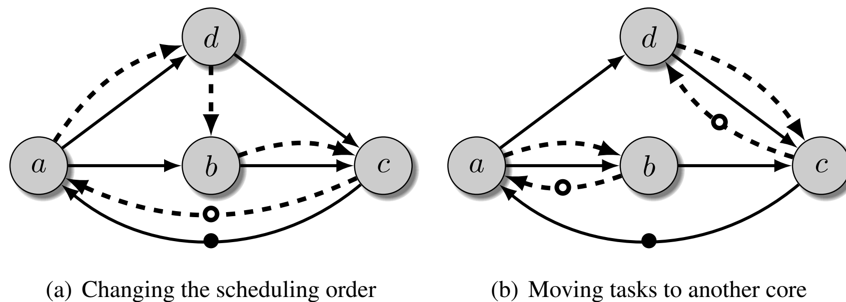
A periodic schedule for any set of actors on a single core can be constructed according the following steps:

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 $v_1 \ldots v_n$ 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.



(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 token placed on the dashed arrow entering actor a.



(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)

3.2 Implementation of scheduling

The management of schedules in dataflow graphs, by adding virtual channels, as explained in the previous section, is implemented by a single class. Java class `HSDFGraphMapper` encapsulates the modelling and management of schedules, by virtual channels, in an HSDF graph. Listing 3.1 lists the public methods exposed by the class.

The class's methods can be divided in three different groups. The first group contains the (protected) methods that manage the virtual channels that model the actual schedules, ensure no deadlock is introduced, etc. These methods are:

`addSchedulingConstraint(from, to):` adds a virtual channel (i.e., a channel that contains no token) between the two specified actors.

`removeSchedulingConstraint(from, to):` removes the virtual channel connecting the two specified actors.

`rebuildPrecedenceGraph():` recomputes the *acyclic precedence expansion subgraph* of the graph, after the set of virtual channels has been changed.

`setSchedule(core, schedule):` adds a cycle (that consists entirely of virtual channels) to the dataflow graph. A single token is placed onto the virtual channel connecting the last actor in the schedule with the first, such that the first actor is scheduled to execute next.

The second group consists of public methods to query the current mapping:

`getMapping():` returns the current mapping as a list of schedules, where each schedule is a list of actors that are to be executed in that order. Note that the first schedule in the list is associated with the first core, the second schedule with the second core, etc.

`getCoreOfActor(a):` returns the core (identified by an integer) onto which actor a is currently scheduled to execute.

`getCoreSchedules():` returns the current mapping, similar to `getMapping()`, but as a `Map` from core to schedule.

`getActorToCoreMappings():` returns the current placement (i.e., actor to core assignments only) as a `Map` from actor to core.

`getPrecedenceGraph():` returns the *acyclic precedence expansion graph* (see previous section).

`getMappedSDFGraph():` returns the dataflow graph, where the virtual channels that model the schedules of the different cores are treated as regular dataflow channels. Prior to calling this function, the method `map()` must be called, otherwise this method will return `null`.

getActiveCores(): returns the number of cores that are actually used (i.e., have actors scheduled to execute on them).

Finally, the last group consists of public methods to initialize or change the current mapping:

clear(): removes all the virtual channels from the graph.

map(): transforms all virtual channels currently in the graph into regular dataflow channels, thereby finalizing the schedules.

setMapping(mapping): adds virtual channels to the graph to model the specified mapping: each list is translated into a cycle of virtual channels, with a single initial token placed in such a way that the first actor in each list is scheduled to execute next.

removeFromSchedule(core, actor): removes the specified actor from the specified core, and updates the core's schedule. As a result, the specified actor is not scheduled on any core.

addToSchedule(core, actor, pos): inserts the specified actor into the specified core, at the specified position, pos.

sequentialize(actors, core, rnd): schedules (orders) a set of actors on one core, such that no deadlock situation is created. Any scheduling freedom that remains, is resolved by choosing randomly, which is why a random number generator (rnd) needs to be provided as a third argument.

Listing 3.1: Class HSDFGraphMapper

```
public class HSDFGraphMapper<V extends TimedActor> {

  public HSDFGraphMapper(
        final DirectedGraph<V, SDFChannel> sdfg);

    // Methods to query the current mapping
  public List<List<V>> getMapping();
  public Integer getCoreOfActor(V a);
  public Map<Integer, List<V>> getCoreSchedules();
  public Map<V, Integer> getActorToCoreMappings();
  public DirectedGraph<V, SDFChannel> getPrecedenceGraph();
  public DirectedGraph<V, SDFChannel> getMappedSDFGraph();
  public List<Integer> getActiveCores();

    // Methods to manipulate the current mapping
  public void clear();
  public void map();
  public void setMapping(List<List<V>> mapping);
  public void removeFromSchedule(int core, V actor);
  public void addToSchedule(int core, V actor, int pos);
```

```
  public void sequentialize(Set<V> actors, int core,
        Random rnd);

    protected void setSchedule(int core, List<V> schedule);
    protected void rebuildPrecedenceGraph();

  protected void addSchedulingConstraint(V from, V to);
  protected void removeSchedulingConstraint(V from, V to);

  public static void main(String[] args)
        throws NoSuchAlgorithmException;
}
```

3.3 Simulated annealing

For reasons described in D21 we employ *simulated annealing* to obtain a near optimal schedule for cores, using throughput as a performance measure. The pseudocode for simulated annealing is shown in Figure 3.3 in which

- The function **initial mapping** gives a random, initial mapping of the dataflow graph onto N cores,

- The function **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. For example, this is illustrated in the transformation of Figure 3.1(b) to either Figure 3.2(a) or 3.2(b).

- The function **estimated-costs** conservatively estimates the maximum throughput.

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')
    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.

3.4 Implementation of the Simulated Annealing Mapper

The process of searching the possible schedules for a dataflow graph for a near-optimal one, using simulated annealing, is implemented by the Java class `SimulatedAnnealer`, which is listed in Listing 3.2. In this class, the five main methods are:

`changeAssignment(a, rnd):` this method removes actor `a` from its current core, randomly picks a different core for the actor, and inserts it in the schedule of the chosen core such that deadlock is avoided.

`modifySchedule(core, rnd)` this method randomly rearranges the schedule (i.e., the order in which the set of assigned tasks are executed) of the core.

`move(rnd)` implements the *move* function found in the pseudocode listed in Figure 3.3. It randomly (both options are equally probable) chooses between changing the scheduling order of one core (by calling `modifySchedule`), or moving a task to another core (by calling `changeAssignment`).

`estimateCosts(steps):` this method esimates the *costs* of the current mapping by simulating steps graph iterations of the dataflow graph. As a cost measure, throughput is used: an increased throughput gives a decreased cost. This method implements function *estimated-costs* in the pseudocode listed in Figure 3.3.

`getInitialMapping(rnd):` implements pseudocode function *initial mapping*, and returns a random initial mapping in which on every core at least one task is scheduled to execute.

Listing 3.2: Class SimulatedAnnealer

```
public class SimulatedAnnealer {

  public SimulatedAnnealer(
        DirectedGraph<TimedActor, SDFChannel> sdfg,
        final int cores);
    public MappedApplication<TimedActor, SDFChannel> getMapping
       (cont.)();
    public HSDFGraphMapper<TimedActor> getInitialMapping();
    public void changeAssignment(TimedActor a, Random rnd);
    public void changeAssignment(TimedActor a, int newcore,
        Random rnd);
    public boolean modifySchedule(int core, Random rnd);
    public void move(Random rnd);

  protected double estimateWaitingTime(final int steps);
  protected double estimateGraphPeriod(final int steps);
  public double estimateCosts(final int steps);

  /**
   *
   */
  public void runSimulatedAnnealing(
        final double T0, final double Tstop,
        final double k, final int steps,
        final int evalDepth, Random rnd);

  public static void main(String[] args) throws Exception;
}
```

The `SimulatedAnnealer` class has a main method, which allows the class to be executed on a in input file containing a dataflow graph model of an S-Net network, by invoking:

```
java -cp . SimulatedAnnealer dataflowgraph.json n
```

Here, n is the number of cores onto which the dataflow graph is to be mapped. Note that the input file is expected to be a text file, structured as a JSON file. This could be the file as delivered by the `SNetASTParser` class described in the previous chapter, or a hand-crafted file.

The result of executing the class's main method on an input dataflow graph file, is twofold. First of all, a dataflow graph file (structured as JSON), containing the original dataflow graph *mapped* with the best mapping found, is created. Second, the a C-file containing the two specified functions is generated, and may be compiled and linked with the S-Net network, to allow the runtime system of S-Net (i.e., LPEL) to query for where to place tasks. Mapping information is made available to the LPEL layer through the following 2 functions specified in the generated C file:

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

The first function fills the tasks parameter with a list of tasks that are to be run in order on the core identified by core_id. The second function returns the core (by its identifier) on which the given task should be placed according to the mapping that was found. Tasks are identified by a so-called *topology string*, see ADVANCE deliverable D21 [6] for more details.

Running the S-Net application with the proposed mapping will result in possibly different timing behaviour of the S-Net boxes, as they are now run with a different placement. Through the monitoring subystem, this observed timing behaviour can be used to update and rerun the simulated annealer, completing the feedback loop as illustrated in Figure 1.1.

Chapter 4

Concluding Remarks

This document presents the technical implementation of the dynamic placement tool, and its position in the overall workflow. The dynamic placement tool consists of several components:

- A *translator* to model an S-Net as a synchronous dataflow graph
- A mapper to model placement and per-core task ordering in a dataflow graph
- A simulated annealer that guides the search through the space of possible mappings towards a near-optimal mapping onto a given number of cores

The technical details of the approach to the scheduling problem are linked to specific methods that are made available to the application programmer's interface of the placement tool. The placement tool requires as input two text files: a file containing the abstract syntax tree of the S-Net application that is to be mapped, and a file that contains monitored performance data of a run of the application on the target hardware platform. The latter file is usually named `n00_tasks.map`.

If the abstract syntax tree of the S-Net application is stored in file `snet.ast`, then computing a placement using the placement tool is a two-step process: In the first step, a synchronous dataflow graph representation is created by running the following command:

```
java -cp . SNetASTParser network-ast.txt n00_tasks.map
```

This step produces the file `sdfgraph.json`. A placement for this graph (on n cores) is then computed by running:

```
java -cp . SimulatedAnnealer dataflowgraph.json n
```

The latter command produces a C file named `mapping.c`, which can be compiled and linked to the S-Net application, such that the computed mapping is instantiated by the S-Net runtime system.

Bibliography

[1] Haoxan Cai, Susan Eisenbach, Alex Shafarenko, and Clemens Grelck. Extending the S-Net Type System. In *Proceedings of the Æther-Morpheus Workshop From Reconfigurable to Self-Adaptive Computing (AMWAS'07), Paris, France*, 2007.

[2] Jean Cochet-terrasson, Guy Cohen, Stéphane Gaubert, Michael Mc Gettrick, and Jean-pierre Quadrat. Numerical Computation of Spectral Elements in Max-Plus Algebra, 1998.

[3] 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.

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

[5] Ali Dasdan. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 9(4):385, 2004.

[6] Robert de Groote and Jan Kuper. D21 design of protocol and interface for feedback mechanism to upstream layers, Feb 2013.

[7] 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.

[8] F. M. Dekking, C. Kraaikamp, H. P. Lopuhaä, and L. E. Meester. *A Modern Introduction to Probability and Statistics*. Springer-Verlag, 2010.

[9] 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.

[10] 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.

[11] C. Grelck. Single assignment c (sac): High productivity meets high performance. In Z. Hórvath and V. Zśok, editors, *4th Central European Functional Programming Summer School (CEFP'11), Budapest, Hungary*, volume 7241 of *Lecture Notes in Computer Science*. Springer, 2012. to appear.

[12] 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.

[13] 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.

[14] Kevin Hammond and Philip Hölzenspies. ADVANCE project deliverable D7: Statistical model of resource usage, 2011.

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

[16] Robert V. Hogg, Allen Craig, and Joseph W. McKean. *Introduction to Mathematical Statistics, 7th Edition*. Pearson, 2012.

[17] P. K. F. Hölzenspies. *On run-time exploitation of concurrency*. PhD thesis, Univ. of Twente, Enschede, The Netherlands, April 2010.

[18] Steffen Jost, Hans-Wolfgang Loidl, Kevin Hammond, and Martin Hofmann. Static determination of quantitative resource usage for higher-order programs. In *POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 223–236, New York, NY, USA, January 2010. ACM.

[19] G Kahn. The semantics of a simple language for parallel programming. In L Rosenfeld, editor, *Information Processing 74, Proc. IFIP Congress 74. August 5-10, Stockholm, Sweden*, pages 471–475. North-Holland, 1974.

[20] Raimund Kirner. ADVANCE project deliverable D4: Report on performance annotations/interfaces, 2010.

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

[22] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[23] Sorin Manolache. *Analysis and optimisation of real-time systems with stochastic behaviour*. PhD thesis, Linköping, 2005.

[24] G. J. Olsder. Performance analysis of data-driven networks. pages 33–41, October 1990.

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

[26] A. V. Shafarenko. Streaming networks for coordinating data-parallel programs (position statement). In *Asia-Pacific Computer Systems Architecture Conference*, volume 4186 of *Lecture Notes in Computer Science*, pages 2–5. Springer, 2006.

[27] SNet. Snet declarative coordination language, 2008.

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

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

[30] 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.

[31] Volkmar Wieser, Philip K.F. Hölzenspies, Raimund Kirner, and Michael RoSSbory. Statistical Performance Analysis with Dynamic Workload using S-NET. Technical report, November 2011. FD-COMA 2012: Workshop on Feedback-Directed Compiler Optimization for Multi-Core Architectures.

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

[33] N Young, R Tarjan, and J Orlin. Faster Parametric Shortest Path and Minimum Balance Algorithms. *ArXiv Computer Science e-prints*, May 2002.