Implementation Architecture and Multithreaded Runtime System of S-Net

Clemens Grelck[1,2] and Frank Penczek[2]

[1] University of Amsterdam, Institute of Informatics
Science Park 107, 1098 XG Amsterdam, Netherlands
c.grelck@uva.nl
[2] University of Hertfordshire, School of Computer Science
Hatfield, Herts, AL10 9AB, United Kingdom
{f.penczek,c.grelck}@herts.ac.uk

Abstract. S-Net is a declarative coordination language and component technology aimed at modern multi-core/many-core architectures and systems-on-chip. It builds on the concept of stream processing to structure networks of communicating asynchronous components, which can be implemented using a conventional (sequential) language. In this paper we present the architecture of our S-Net implementation. After sketching out the interplay between compiler and runtime system, we characterise the deployment and operational behaviour of our multithreaded runtime system for contemporary multi-core processors. Preliminary runtime figures demonstrate the effectiveness of our approach.

1 Introduction

The free lunch is over! Excessive power consumption and heat dissipation have eventually set an end to clock frequency scaling, and the current trend in processor architecture is to go multi-core [1]. Small-scale dual-core and quad-core processors already dominate the consumer market while the roadmaps of all major hardware manufacturers promise a steep rise in the number of cores [2]. At the same time, massively parallel processors (e.g. GPGPUs, accelerator boards) already offer a degree of parallelism in computing resources that very recently could only be found in dedicated supercomputing installations [3].

This hardware trend towards multi-core/many-core designs puts immense pressure on software manufacturers: For the first time in history software does not automatically benefit from a new generation of hardware as was characteristic for the era of clock frequency scaling. Software must become parallel in order to benefit from future processor generations! However, to the present day software is predominantly sequential adhering to the von Neumann model of computing. While parallel computing is an established discipline, it has always been confined to supercomputing application areas and installations. Now, parallel computing must go mainstream, but the existing tools and technologies were developed for experts in the niche, not for the mainstream.

S-Net [4] is a novel approach to ease parallelisation of existing and new applications. It builds on separation of concerns as the key design principle: an *application engineer* uses domain-specific knowledge to provide application building blocks of suitable granularity in the form of (rather conventional) functions that map inputs into outputs. In a complementary way, a *concurrency engineer* uses his expert knowledge on target architectures and concurrency in general to orchestrate the (sequential) building blocks into a parallel application. In fact, S-Net turns regular functions/procedures implemented in a conventional language into asynchronous, state-less components communicating via uni-directional streams. The choice of a component language solely depend on the application domain of the components itself. In principle, any conventional programming language can be used, and a single S-Net network can manage components implemented using different languages.[3] Fig. 1 shows an example of an S-Net streaming network.

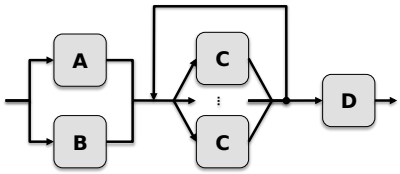

Fig. 1: Illustration of an S-Net streaming network of asynchronous components

Note that any base component is characterised by a single input and a single output stream. This restriction is motivated, again, by the principle of separation of concerns: The concern of a box is mapping input values into output values, whereas its purpose within a streaming network is entirely opaque to the box itself. Concurrency concerns like synchronisation and routing that immediately become evident if a box had multiple input streams or multiple output streams, respectively, are kept away from boxes. Our solution achieves a near-complete separation of computing and coordination aspects. We have identified four fundamental construction principles for streaming networks:

- *serial composition* of two (potentially) different components where the output stream of one component becomes the input stream of another component;
- *parallel composition* of two (potentially) different networks where some routing oracle decides on which branch data takes;
- *serial replication* of a single network where data is streamed through the same network a dynamically determined number of times; and
- *indexed parallel replication* of a single network where an index attached to the data determines which branch (or which replica of the network) is taken.

[3] There are, however, technical limitations on the interoperability of languages and on the technical interplay between coordination and computation layer.

These four construction principles allow concurrency engineers to define complex streaming networks of asynchronous components and to turn sequential code blocks into a parallel application ready to effectively exploit the capabilities of modern multi-core and many-core processors with very little effort.

This paper is the first to report on our implementation of S-Net. The architecture of our implementation features a target independent compiler that generates (C) code dominated by calling functions from the S-Net *common runtime interface*. Multiple implementations of the common runtime interface allow us to support varying concrete hardware platforms in a plug-in manner. In this paper, we provide an in-depth description of our multithreading based runtime system for contemporary multi-core processors with shared memory. We will describe this runtime system at a level of abstraction such that it may equally well serve as an operational semantics of S-Net.

It is a characteristic feature of our common runtime interface that it does not prejudice any concrete representation of the streaming network. This proves essential when it comes to supporting a wide range of target architectures with highly varying demands and capabilities, e.g. microthreading on the MicroGrid architecture [5, 6]. As a consequence, it is up to any concrete runtime system to define the dynamic representation of the streaming network, and runtime system functionality naturally falls into one of two categories: *network deployment* and *network operation*. The former is concerned with instantiating the streaming network on the target platform while the latter defines the operational behaviour of the network constituents. As we will see later, these two aspects are not necessarily consecutive, but interleave in practice.

The specific contributions of the paper are the

- presentation of the overall architecture of our S-Net implementation;
- formal description of network deployment;
- formal specification of the operational behaviour of components;
- implementation of guarantees for package ordering; and
- preliminary performance figures.

The remainder of the paper is organised as follows: In Section 2 we provide a more detailed introduction to S-Net. Section 3 sketches out the architecture of our S-Net implementation including compiler, runtime system and their interplay. Section 4 discusses network instantiation while the operational behaviour of network components is described in Section 5. In Section 6 we elaborate on package ordering. Eventually, we provide some preliminary runtime figures in Section 7, discuss related work in Section 8 and conclude in Section 9.

2 S-Net in a nutshell

As a pure coordination language S-Net relies on a separate component language to describe computations. Such components are named *boxes* in S-Net terminology, their implementation language *box language*. Any box is connected to the rest of the network by two typed streams: an input stream and an output

stream. Messages on these typed streams are organised as non-recursive records, i.e. label-value pairs. Labels are subdivided into *fields* and *tags*. Fields are associated with values from the box language domain. They are entirely opaque to S-Net. Tags are associated with integer numbers that are accessible both on the S-Net and the box language level. Tag labels are distinguished from field labels by angular brackets.

On the S-Net level, the behaviour of a box is declared by a *type signature*: a mapping from an *input type* to a disjunction of *output types*. For example,

```
box foo ({a,<b>} -> {c} | {c,d,<e>})
```

declares a box that expects records with a field labelled `a` and a tag labelled `b`. The box responds with a number of records that either have just a field `c` or fields `c` and `d` as well as tag `e`. Both the number of output records and the choice of variants are at the discretion of the box implementation alone.

As soon as a record is available on the input stream, a box consumes that record, applies its box function to the record and emits the resulting records on its output stream. In the simple but common case of a one-to-one mapping between input and output records the box function's result value may determine the output record. In the general case, our *box language interface* provides a box language specific abstraction named `snet_out` to dynamically produce output records during the execution of the box function. As soon as the evaluation of the box function is complete, the S-Net box is ready to receive and process the next input record.

S-Net boxes are stateless by definition, i.e., the mapping of an input record to a stream of output records is free of side-effects or, in other words, purely functional. We exploit this property for cheap relocation and re-instantiation of boxes; it distinguishes S-Net from conventional component technologies. In particular if boxes are implemented using imperative languages, S-Net, however, can only guarantee that box functions actually adhere to the *box language contract* as far as the box language supports such guarantees. This is in the end the same in any functional language that supports calling non-functional code.

In fact, the above type signature makes box `foo` accept *any* input record that has *at least* field `a` and tag `<b>`, but may well contain further fields and tags. The formal foundation of this behaviour is *structural subtyping* on records: Any record type t_1 is a subtype of t_2 iff $t_2 \subseteq t_1$. This subtyping relationship extends nicely to multivariant types, e.g. the output type of box `foo`: A multivariant type x is a subtype of y if every variant $v \in x$ is a subtype of some variant $w \in y$.

Subtyping on the input type of a box means that a box may receive input records that contain more fields and tags than the box is supposed to process. Such fields and tags are retrieved from the record before the box starts processing and are added to each record emitted by the box in response to this input record, unless the output record already contains a field or tag of the same name. We call this behaviour *flow inheritance*. In conjunction, record subtyping and flow inheritance prove to be indispensable when it comes to making boxes that were developed in isolation to cooperate with each other in a streaming network.

It is a distinguishing feature of S-Net that we do not explicitly introduce streams as objects. Instead, we use algebraic formulae to define the connectivity

of boxes. The restriction of boxes to a single input and a single output stream (SISO) is essential for this. As pointed out earlier, S-Net supports four network construction principles: static serial/parallel composition and dynamic serial/-parallel replication. We build S-Net on these construction principles because they are pairwise orthogonal, each represents a fundamental principle of composition beyond the concrete application to streaming networks (i.e. serialisation, branching, recursion, indexing), they naturally express the prevailing models of parallelism (i.e. task parallelism, pipeline parallelism, data parallelism) and, last not least, we believe that these four principles are sufficient to construct all useful streaming networks. The four network construction principles are embodied by *network combinators*. They all preserve the SISO property: any network, regardless of its complexity, again is a SISO component.

Let `A` and `B` denote two S-Net networks or boxes. Serial composition (denoted `A..B`) constructs a new network where the output stream of `A` becomes the input stream of `B` while the input stream of `A` and the output stream of `B` become the input and output streams of the compound network, respectively. As a consequence, instances of `A` and `B` operate asynchronously is a pipelined fashion. In the intuitive example of Fig. 1 serial composition can be identified between the left, the middle and the right subnetworks.

Parallel composition (denoted `(A|B)`) constructs a network where all incoming records are either sent to `A` or to `B` and the resulting record streams are merged to form the overall output stream of the compound network. By means of type inference [7] we associate each operand network with a type signature similar to the annotated type signatures of boxes. Any incoming record is directed towards the operand network whose input type better matches the type of the record itself. If both branches in the streaming network match equally well, one is selected non-deterministically. The example network in Fig. 1 features parallel composition in combining `A` and `B`.

Serial replication (denoted `A*`*`type`*) constructs a conceptually infinite chain of instances of `A`. The chain is tapped before every instance to extract records that match the type pattern given as right operand (i.e. the record's type is a subtype of specified type). Such records are merged into the output stream. In a simplifying view Fig. 1 illustrates serial replication as a feedback loop. While in a completely stateless setting feedback and replication are equivalent, the presence of synchronisation facilities (see below) requires us to make this subtle difference. From a conceptual point of view, their relationship resembles that of recursion and iteration; from a pragmatic point of view, the separation of data in different instances of the operand network contributes to an orderly system behaviour.

Indexed parallel replication (denoted `A!<`*`tag`*`>`) replicates instances of `A` in parallel. Unlike in static parallel composition we do base routing on types and the best-match rule, but on a tag specified as right operand. All incoming records must feature this tag; its value determines the instance of the left operand the record is sent to. Output records are non-deterministically merged into a single output stream similar to parallel composition. In Fig. 1 we can identify parallel

replication of network `C`. To summarise we can express the S-Net sketched out in Fig. 1 by the following expression:

```
(A|B) .. (C!<t>)*{p} .. D
```

assuming previous definitions of `A`, `B`, `C` and `D`. While this example remains in the abstract, concrete S-Net applications can be found in [8, 9].

Last not least, S-Net features a synchronisation component that we call *synchrocell*; it takes the syntactic form `[|type,type|]`. Similar to serial replication the types act as patterns for incoming records. A record that matches one of the patterns is kept in the synchrocell. As soon as a record arrives that matches the other pattern, the two records are merged into one, which is forwarded to the output stream. Incoming records that only match previously matched patterns are immediately forwarded to the output stream. Hence, a synchrocell becomes an identity after successful synchronisation and may be removed by a runtime system. The extremely simplified behaviour of synchrocells captures the essential notion of synchronisation in the context of streaming networks. More complex synchronisation behaviours, e.g. continuous synchronisation of matching pairs in the input stream, can easily be achieved using synchrocells and network combinators. See [8] for more details on this and on the S-Net language in general.

3 Implementation architecture

The implementation architecture of S-Net is designed in a modular fashion: We can identify two self-contained modules, the *compiler* and the *runtime system*. We will focus on the runtime system in the remainder of this paper, but nevertheless briefly present the entire system design in Fig. 2 to give a complete picture of our overall approach. In this two-layered architecture, a network is transformed into three conceptually different network representations while it is being processed by the system.

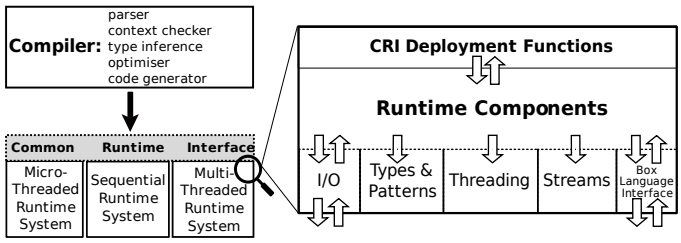

Fig. 2: The system consists of a compiler and various runtime system implementations

To illustrate this process of transformation from user-written code to an actually executed program, we apply the presented transformations to our running example. The S-Net compiler represents the example network from previous

sections as the tree shown in Fig. 3. On this abstract syntax tree (AST) representation, the compiler carries out various optimisation and annotation tasks in addition to the most important task of type inference [7]. From user-defined types and patterns and from the inferred information, the compiler generates decision functions. The runtime system applies these functions to incoming records to determine routing of records and match records against patterns. We illustrate the compilation process in Fig. 3.The final stage of the compiler is code generation. Here, the compiler generates a textual representation of the AST. This representation of the network is a portable format we refer to as *common runtime interface* (CRI) representation.

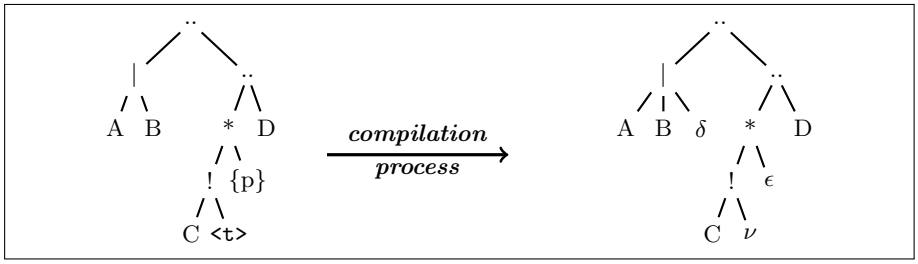

Fig. 3: The S-Net compiler reads in a network description and annotates the parse tree with runtime-specific information, as for example decision functions (ϵ, ν, δ), without dissolving the tree structure as such.

The CRI decouples the compilation process from the runtime system. The compiler does not dissolve the hierarchical structure of the network, which leaves this traditional compiler task to the runtime system. This achieves high flexibility for runtime system implementors: as the compiled structure still features a high level of abstraction, the actual decomposition, i.e. the interpretation of the CRI format, takes places in the runtime system according to the target architecture. This approach turns the compiler into a universal component which we can re-use for any concrete implementation of a runtime system.

The transformation from CRI format to the final and actually executed representation of the network is carried out by a component that any runtime-system implementation has to provide: the *deployer*. The deployer is specific to a runtime-system as it implements the final transformation of the network representation. As this is an integral part of the runtime system, it is presented in great detail in the following section. As mentioned above, the concrete implementation of the runtime representation is entirely decoupled from any stages above the deployer. For the time being, we are targeting three destination architectures:

- sequential execution,
- multithreading based on PThreads and
- microthreading based on μTC [5].

In this paper, however, we solely focus on the multi-threaded implementation based on PThreads. The main idea of this implementation is to break down the network into smaller runtime components which are connected to each other by buffered streams. The compiled program which is being executed in this implementation resembles the intuitive view of a network in which data elements are flowing from component to component for processing.

The general implementation design of the runtime system is illustrated in Fig. 2(right): Apart from the deployer, the runtime system consists of several smaller modules for type and pattern representation, thread management, communication, box language interfacing, and general I/O. These modules provide functionality for the runtime components, which form the core of the runtime system. These components implement the runtime behaviour of all S-Net combinators.

4 Network deployment

The deployment process transforms the compiler-generated CRI representation of a network into a network of runtime components. In this section we will take a closer look at this process for each S-Net entity. In the presented source code, we shall use a teletype font for identifiers if these refer to streams.

The deployment of box and synchrocell (Fig. 4) connects an inbound stream to the appropriate runtime component. From the stream, the box resp. synchrocell reads inbound records for processing. Result records produced by the component are sent out via an outbound stream, which is created by the deployer by calling **new** Stream(). In addition to these streams, both components require auxiliary parameters: For the deployment of the synchrocell, the compiler generates two decision functions, μ_a and μ_b, from the user-defined patterns of the synchrocell. For box deployment, the user-defined, internal behaviour f of the box (the box implementation) is required. Additionally, the box component also requires a compiler-generated type encoding τ of the box's input type. The purpose of these parameters are explained in greater detail in Section 5. The runtime components are started by a call to **spawn**.

The **new** and **spawn** functions are high-level abstractions of the rather low-level code of the PThreads implementation. We do this for the sake of presentation, as the concrete code is less concise (but trivial).

The simplest case of combinator deployment is the deployment of a *serial* combinator (Fig. 5). Both operands are deployed recursively. The inbound stream is connected to the first operand. The outbound stream of the deployed first operand is connected to the second operand, whose outbound stream constitutes the outbound stream of the compound runtime component network, representing this serial combination.

The deployment of the *choice* combinator (Fig. 5) requires two runtime components in addition to the components of its operands. These additional components implement the implicit splitting and merging points of streams in an S-Net *choice* combination. The dispatcher, a multi-outbound stream compo-

```
𝒟(box_τ f, in) =
  let out = new Stream()
      box = spawn Box( in, f, τ, out)
  in out

𝒟([| μ_a, μ_b |], in) =
  let out = new Stream()
      sync = spawn Sync( in, μ_a, μ_b, out)
  in out
```

Fig. 4: Deployment of box and synchrocell

nent, forwards inbound records to one of the operands. The output streams of the operands are connected to a complementary multi-inbound stream component, the collector. The collector aggregates the output streams of the operands and bundles these to a single output stream. The CRI representation of the *choice* combinator requires a decision function δ, which the compiler generates. The dispatcher evaluates this function for each inbound record to determine routing destinations. The deployment of the choice combinator also deploys the operands of the combinator. This recursive process only ends once an operand does not have any more operands, i.e., if it is a box or synchrocell.

```
𝒟(A .. B, in) = let out = 𝒟(A, in) in 𝒟(B, out)

𝒟(A |_δ B, in) =
  let opin_1 = new Stream()
      opin_2 = new Stream()
      disp = spawn ChoiceDispatch( in, δ, opin_1, opin_2)
      opout_1 = 𝒟(A, opin_1)
      opout_2 = 𝒟(B, opin_2)
      out = new Stream()
      coll = spawn Collector( nil, {opout_1,opout_2},out)
  in out

𝒟(A *_ε, in) =
  let ctrl = new Stream()
      bypass = new Stream()
      disp = spawn StarDispatch( in, A, ε, ctrl, nil, bypass)
      out = new Stream()
      coll = spawn Collector( ctrl, {bypass}, out)
  in out

𝒟(A !_ν, in) =
  let ctrl = new Stream()
      disp = spawn SplitDispatch( in, A, ν, ctrl, ∅)
      out = new Stream()
      coll = spawn Collector( ctrl, ∅, out)
  in out
```

Fig. 5: Deployment of combinators

Similar to the *choice* deployment, the deployment of a *star* combinator (Fig. 5) also sets up a dispatcher and a collector. The operand of the *star*, how-

ever, is not deployed yet — its deployment is fully demand-driven and postponed until runtime. For this reason, the operand is passed directly to the dispatcher, in conjunction with a compiler-generated decision function ϵ. The dispatcher calls the deployment function for the operand only when needed, driven by the outcome of the decision function. As the operand network of the *star* combinator may evolve (unfold) over time, the associated collector has to be able to manage a potentially growing set of inbound streams. A control stream between dispatcher and collector is set up to serve the purpose of communicating the appearance of new streams to the collector.

From the deployment perspective, the *split* combinator (Fig. 5) is very similar to the *star* combinator. The operand instantiation is demand driven and triggered by a decision function ν. After the initial deployment, no operand instance is present, and thus, the stream set of the collector is empty. A control stream is established between dispatcher and collector to register the outbound streams of dynamically created, new instances of the operand.

To illustrate the deployment process, the resulting component network for the running example is shown in Fig. 6(a). Instances of C and the surrounding splitter are not built by the deployment process until required. Instead, only an initial connection between the star dispatcher and its collector ensures that the network is fully connected. While processing records, instances of the star operand and the split operand are spawned demand-driven. The component network as it has developed after one instance of the star operand and three instances of the split operand have been built is shown in Fig. 6(b).

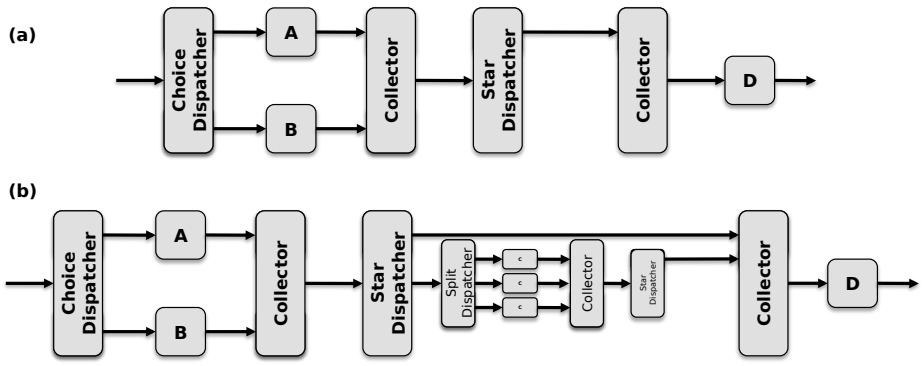

Fig. 6: Deployed component network of example network which initially only contains A, B, D (a) Remaining operands are deployed demand-driven (b)

5 Operational behaviour of components

We continue denoting streams by identifiers set in teletype. Concatenation of records and streams are denoted by $\lhd$ (append as prefix) and $\rhd$ (append as suffix). Each definition of a component starts with the keyword **Thread** to emphasise the fact, that the component is executed as a thread. Case differentiations are introduced by a | in the source code, guard expressions of cases by the keyword **when**.

The synchrocell component implements two main tasks. Firstly, it stores records if they match the specified synchronisation pattern. The procedure here again relies on match functions: The compiler generates one match function for each synchrocell pattern. Secondly, the component merges records, once all pattern have been matched. We model record storage as parameters to the component function, which serves a dual purpose. It stores the records if they match a pattern and also encodes the state of the synchrocell. The state determines the synchrocell's operations, depending on which pattern was matched by an inbound record. A synchrocell of two patterns (see Fig. 7; if a storage parameter does not hold a record yet, this is indicated by -) and two match functions μ_a and μ_b, has the following possible states and transitions:

State	μ_a	μ_b	Description (current)	Action	Next state
- -	•		initial state	store record	q -
- -		•	initial state	store record	- q
- -	•	•	initial state	output	*id*
q -	•		first pattern was matched	output	q -
q -		•	first pattern was matched	merge and output	*id*
q -	•	•	first pattern was matched	merge and output	*id*
- q	•		second pattern was matched	merge and output	*id*
- q		•	second pattern was matched	output	- q
- q	•	•	second pattern was matched	merge andoutput	*id*
id	n/a	n/a	sync replaced by identity	output	*id*

When a record is stored in the synchrocell, all constituents of the record which are not part of the pattern, are stripped out. This is implemented by the strip function which uses the decision functions to determine which record constituents are to be removed. Only the remainder is stored for the merging process. If a record matches a pattern for which a record has already been stored, the *output* action forwards the record to the outbound stream `out`. A record that matches the last remaining previously unmatched pattern, is merged if a record is available in storage, and simply output otherwise. Record merging is defined by the flow inheritance operator, which is presented in Fig. 7. For clarity, we presented an implementation where the sync component is replaced by an Id component after the last pattern has been matched. The Id component forwards all inbound records directly to the outbound stream. In practice, however, dead synchrocells are completely removed in a garbage collection step, where the cell's inbound stream is directly connected to its successor component.

The box component is a connector from the S-Net domain to the box language domain. The box component calls the box function and provides it with an inbound record and the inbound type τ of the S-Net box. The box function may produce an arbitrary amount of records during its execution, each of which needs to flow inherit fields from the original inbound record. To make this process convenient for a box programmer, an SNetOut function is provided. This function expects one result record at a time, carries out flow inheritance and writes the record to the outbound stream. For each output the box produces, it calls SNetOut. After the execution of the box functions finishes, control is returned to the box component.

```
Thread Box( r◁in, f, τ, out) =
        let t' = f( r, τ, out)
        in Box( in, f, τ, t')

fun SNetOut( r, τ, res, out) =
     let f  = r \ τ
         rf = f ⋈ res
     in out▷rf

Thread Sync( r◁in, μa, μb, -, -, out) when μa(r) ∧ μb(r) =
          Id( in, out▷r)
|      Sync( r◁in, μa, μb, -, -, out) when μa(r) =
       let q = strip( r, μa)
       in Sync( in, μa, μb, q, -, out)
|      Sync( r◁in, μa, μb, -, -, out) =
       let q = strip( r, μb)
       in Sync( in, μa, μb, -, q, out)
|      Sync( r◁in, μa, μb, q, -, out) when ¬μb(r) =
         Sync( in, μa, μb, q, -, out▷r)
|      Sync( r◁in, μa, μb, q, -, out) =
       let m = r ⋈ q
       in Id( in, out▷m)

Thread Id(r◁in, out) = Id(in, out▷r)

fun infix ⋈ f r = r ∪ (f \ r)
```

Fig. 7: Implementation of sync, box and flow-inheritance operator

The collector (Fig. 8) is a multi-inbound stream component. This component is used where multiple operand streams are merged into one single outbound stream. The collector keeps all streams that it monitors in a stream set S. When records become available on any of the streams in the set, the record is read from the stream and forwarded to the outbound stream `out`. The collector is always deployed as part of a dispatcher-collector pair, with a control stream connecting these two. The registration of new channels is implemented using this control stream: Dispatchers send streams of dynamically created operand instances via `ctrl` to the collector, where the streams are added to the stream set.

The choice dispatcher is a multi-outbound stream component. It reads records from its single inbound stream, and forwards the records over one of the outbound streams to the operand networks. The compiler-generated δ function is an

integral part of this process. The compiler generates this function from the input types of the *choice* operands. Applied to a record r, δ returns an integer value n, depending on which operand input type the record matched. The dispatcher shown in Fig. 8 reads a record r from the inbound channel `in`. If δ applied to r evaluates to 1, the record is forwarded to the inbound channel $\texttt{opin}_1$ of the first operand, and to the second operand via $\texttt{opin}_2$ otherwise. We chose to design δ as a function to integers and not to a binary set, which would be sufficient for this purpose. The integer domain enables us to implement an optimisation to reduce the overhead that multiple dispatcher would cause. The optimisation maps an n-fold choice combination to a single, n-channel choice dispatcher, as opposed to $n-1$ binary dispatchers.

The main purpose of the star dispatcher (Fig. 8) is to decide, whether an inbound record matches the exit pattern of the *star* combinator or not. If the record matches, the dispatcher sends the record to the outbound stream. If the record does not match the pattern, the dispatcher sends the record to the operand network. To make this decision, the dispatcher employs a decision function m. This function is generated by the compiler from the exit pattern of the *star* combinator. When applied to a record, the decision function evaluates to true, if the record matches the exit pattern, and to false otherwise. The instantiation of operands is demand-driven, and hence the star dispatcher is initially not connected to any operand. After deployment, the only connections the dispatcher maintains are an outbound channel (bypass, `bps`) and a control channel (`ctrl`) to the collector. The operand has not been deployed, and the continuation stream `cont` not yet been built. This setup does not change, as long as all inbound records match the exit pattern. The dispatcher immediately sends matching records via `b` to the collector. In case a record does not match the pattern, the operand is deployed.To do this, the `cont` stream is created and connected as inbound stream to the operand. The dispatcher now sends all records that do not match the exit pattern to this continuation stream for processing by the operand. As *star* is a feed-forward combinator, all output of the operand is sent to a new instance of the combinator. This is achieved by instantiating a new dispatcher, in the same way the current dispatcher was set up by the deployment function. No new collector needs to be instantiated: The already existing collector is notified via the control stream. The new dispatcher instance sends all records that match the exit pattern via stream `bps'` to the collector. If the pattern is not matched, the described process repeats itself.

The split dispatcher (Fig. 8) sends records to the proper instance of its operand. This instance is determined by the value of the given tag at runtime. To read the tag value from a record, the compiler generates a function ν. This function returns the integer value of the appropriate tag from a record. The split dispatcher deploys instances demand-driven, an thus, no instance is present initially. When a new instance is deployed, the inbound stream instance is added to the set of served channels. The dispatcher associates the tag value with the instance (more specifically, with the inbound stream) and sends the new outbound

stream to the collector. All future records, which carry the same tag value, will be forwarded to this instance.

```
Thread Collect( in◁ctrl, S, out) =
         Collect( ctrl, {in}∪S, out)
|      Collect( ctrl, {r◁in} ∪ S, out) =
         Collect( ctrl, {in} ∪ S, out▷r)

Thread ChoiceDispatch( r◁in, δ, out1, out2) when δ(r) = 1 =
         ChoiceDispatch( in, δ, out1 ▷r, out2)
|      ChoiceDispatch( r◁in, δ, out1, out2) =
           ChoiceDispatch( in, δ, out1, out2▷r)

Thread
  StarDispatch( r◁in, N, m, ctrl, cont, bps) when m(r) =
    StarDispatch( in, N, m, ctrl, cont, bps▷r)
| StarDispatch( r◁in, N, m, ctrl, nil, bps) =
  let cont = new Stream()
      out = 𝒟(N, cont▷r)
      bps' = new Stream()
      disp = spawn StarDispatch( out, N, m, ctrl, nil, bps')
  in StarDispatch( in, N, m, ctrl▷bps', cont, bps)
| StarDispatch( r◁in, N, m, ctrl, cont, bps) =
    StarDispatch( in, N, m, ctrl, cont▷r, bps)

Thread SplitDispatch( r◁in, A, ν, ctrl, {opin_ν(r)}∪S) =
         SplitDispatch(in, A, ν, ctrl, {opin_ν(r)▷r}∪S)
|      SplitDispatch( r◁in, A, ν, ctrl, S) =
         let opin_ν(r) = new Stream()
         opout = 𝒟(A, opin_ν(r)▷r)
         in SplitDispatch(in, A, ν, ctrl▷opout, {opin_ν(r)▷r}∪S)
```

Fig. 8: Implementation of combinator components

6 Guaranteeing causal record order

As explained in Section 2, parallel composition as well as serial and parallel replication involve merging output streams in a non-deterministic way, i.e., any record produced by some subnetwork proceeds as soon as possible. As a consequence, records travelling on different branches through the network may overtake each other, as Fig. 9 illustrates on the simple example of parallel composition. While merging streams in a non-deterministic way enables S-Net programs to adapt to load distribution in concurrent systems and leads to efficient runtime behaviour in general, there are situations where non-deterministic system behaviour is undesirable. Therefore, S-Net provides deterministic variants of the aforementioned combinators: `||`,`**`,`!!`.[4] Unlike their non-deterministic counterparts described so far, they are guaranteed to maintain the causal order along

[4] The choice of doubling the character of the non-deterministic combinator is motivated by the observation that the serial combinator (`..`) is the only original network combinator that does maintain causal order on streams.

branches of the streaming network: any record created in one branch of the network as a (potentially indirect) response to a record on the compound network's input stream precedes any other such record on the compound network's output stream that stems from a subsequent record on the input stream.

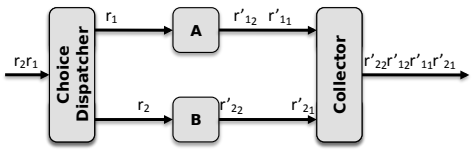

Fig. 9: Causal record order ($r'_{2_2} r'_{2_1} r'_{1_2} r'_{1_1}$) is lost if records overtake each other

Both compilation and deployment are largely unaffected by the introduction of deterministic combinators. They merely produce deterministic variants of the dispatcher and collector components with identical argument sets as the non-deterministic counterparts. However, the operational behaviour of deterministic components deserves our attention. Fig. 10 shows our solution. We leave out deterministic star and split dispatchers; their definition follows the same pattern as illustrated by means of the choice dispatcher.

Conceptually, each record that enters a deterministic subnetwork is mapped to a separate substream. Within the network, all records that are produced from the inbound record, remain in the same substream. The dispatcher-collector pair ensures, that any substream is completely output before any elements of another substream are forwarded to the merged output stream. We implement substreams by help of control records that act as stream delimiters. A control record $[l,c]$ has two attributes: a *level* l and a *counter* c. Only deterministic dispatch components create control records. The counter value is increased for each new control record to distinguish consecutive substreams. The purpose of the level value is to identify correct dispatcher-collector pairs in the presence of recursively nested deterministic and non-deterministic network combinators. When a new record arrives at a deterministic dispatcher, a fresh control record is sent ahead of the record itself to the appropriate output stream following consultation of the oracle function δ. Inbound control records are broadcast to all branches with the level value incremented by one.

The deterministic collector, as shown in Fig. 10, complements the deterministic (choice) dispatch, but in fact this collector is used to implement deterministic replication combinators as well. This collector ensures that different substreams appearing on its inout streams are forwarded to its output stream without interleaving and in the right order. To achieve this, the collector maintains two stream sets: The *ready* set R contains all streams on which the collector actively snoops for input while the *waiting* set W contains those input streams that are currently blocked.

```
𝒟(A ||_δ B, in) =
let opin_1 = new Stream()
    opin_2 = new Stream()
    disp = spawn DetChoiceDispatch( in, δ, 1, opin_1, opin_2)
    opout_1 = 𝒟(A, opin_1)
    opout_2 = 𝒟(B, opin_2)
    out = new Stream()
    coll =
     spawn DetCollect(nil,{opout_1,opout_2},∅,1,-,out)
in out

Thread
  DetChoiceDispatch( [l,c]◁in, δ, cnt, opin_1, opin_2) when l > 0 =
   DetChoiceDispatch( in, δ, cnt, opin_1▷[l+1,c], opin_2▷[l+1,c])
| DetChoiceDispatch( r◁in, δ, cnt, opin_1, opin_2) =
   if δ(r) = 1
   then DetChoiceDispatch( in, δ, cnt+1, opin_1▷[0,cnt]▷r, opin_2)
   else DetChoiceDispatch( in, δ, cnt+1, opin_1, opin_2▷[0,cnt]▷r)

Thread
  DetCollect( ctrl, {[0,c]◁in} ∪ R, W, cnt, tosend, out) =
    if c = cnt
    then DetCollect( ctrl, {in} ∪ R, W, cnt, tosend, out)
    else DetCollect( ctrl, R, {[0,c]◁in} ∪ W, cnt, tosend, out)
| DetCollect( ctrl, {[l,c]◁in} ∪ R, W, cnt, tosend, out) =
    DetCollect( ctrl, R, {in} ∪ W, cnt, [l,c], out)
| DetCollect( ctrl, {r◁in} ∪ R, W, cnt, tosend, out) =
      DetCollect( ctrl, {in} ∪ R, W, cnt, tosend, out▷r)
| DetCollect( ctrl, ∅, W, cnt, -, out) =
    DetCollect( ctrl, W, W, cnt+1, out)
| DetCollect( ctrl, ∅, W, cnt, [l,c], out) =
    DetCollect( ctrl, W, W, cnt, -, out◁[l−1,c])
| DetCollect( in◁ctrl, R, W, cnt, tosend, out) =
    DetCollect( ctrl, {in} ∪ R, W, cnt, tosend, out)
```

Fig. 10: Deployment and implementation of deterministic choice combinator

When a control record appears on one of the ready input streams that was emitted by the dispatch component corresponding to this collector (level 0), we check its counter: if the counter coincides with the internal counter of the collector (*cnt*), it marks the beginning of the next substream to be sent to the collector's output. If so, the corresponding input stream remains in the ready set and the control record is discarded. Otherwise, the input stream is moved from the ready set to the waiting set without consuming the control record.

Any control record that belongs to an outer dispatcher-collector pair (level > 0) appearing on a ready input stream causes that stream to be moved to the waiting set while the control record is stored in the collector. Keep in mind that the corresponding dispatcher had broadcast this control record to all its output streams. So, the collector must retrieve them sooner or later from all of its input streams. Only after the last such instance of the control record has been received by the collector, it may issue a single instance on the output stream.

Any regular record appearing on a ready input stream is immediately forwarded to the output stream. Note that any normal record is preceded by the control record identifying the substream the subsequent regular records belong to. If an input stream is still in the ready set when a regular record arrives, this

means that this is the active substream to be issued on the output stream. Only one such active input stream exists at a time. If there are still further input streams in the ready set, then only because the corresponding control record has not yet arrived.

If the ready set becomes empty, i.e. a followup control record appeared on the previously active input stream indicating the end of that substream, we restore a fresh ready set from the waiting set and increment the internal counter of the collector. This step will make the collector identify the next active substream. In case we have a pending control record belonging to an outer dispatcher-collector pair, we forward it to the output stream with a decremented level counter.

Last not least, we may at any time receive a new input stream via the control stream. In this case we add the new input stream to the ready set. This feature of the collector is only used for implementing the deterministic replication combinators, that lead to dynamically evolving networks.

To make this scheme work, some minor extensions are required for non-deterministic dispatchers and collectors: Dispatchers must broadcast control records to all output streams without touching them. Collectors must gather control records on the various input streams and discard all by one, which is issued on the output stream.

7 Performance Evaluation

For a very preliminary performance evaluation we present runtimes obtained for an application from the radar signal processing domain. In essence, the application is a serial composition of signal processing functions, which are applied to an incoming radar echo. The purpose of the application is to identify slowly moving objects on the ground from the signal of an aircraft mounted radar antenna. As classical Doppler radar approaches fail to produce good results in this area, this application employs an adaptive technique, where signal filters are computed at runtime, depending on incoming data. More detail about the concrete implementation of the application is available in [8].

For the presented runtime measurements we have used three different platforms: a conventional unicore processor (Machine A, Intel Celeron M with 1GB of memory running Linux), a modern dual-core processor (Machine B, Intel Core Duo with 2GB of memory running Linux) and a twofold dual-core multiprocessor (Machine C, 2x AMD Opteron 275 with 8GB of memory running Linux). Fig. 11 shows the outcome of our experiments.

At first, we measure the processing time for one single record by feeding one record at a time into the network. Any subsequent record is only sent once the result of the previous record has been received. This results in sequential processing and serves as a baseline for performance. In a second experiment, we continuously feed records into the network and, thus, take advantage from multiple processing resources. As Fig. 11 shows, we indeed achieve considerable speedups on multi-core hardware by simply organising sequential legacy components into an S-Net streaming network.

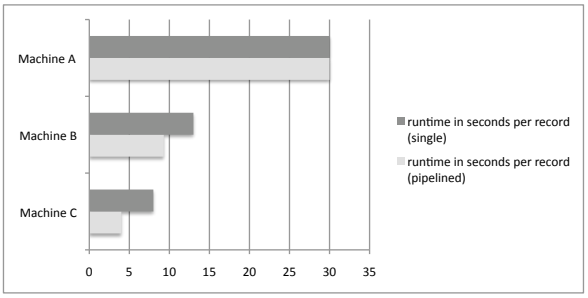

Fig. 11: Preliminary performance measurements

8 Related work

The concept of stream processing has a long history (see [10] for a survey). The view of a program as a set of processing blocks connected by a static network of channels goes back at least as far as Kahn's seminal work [11] and the language Lucid [12]. Kahn introduced the model of infinite-capacity, deterministic process networks and proved that it had properties useful for parallel processing. Lucid was apparently the first language to introduce the basic idea of a block that transforms input sequences into output sequences. For a more theoretical treatment of stream processing we refer to [13] and [14].

Multi-threaded execution of streaming networks with PThreads is offered by StreamIt [15]. In StreamIt, streaming networks are implemented in a high-level, Java like language. A network is defined by implementing boxes (called *filters* in StreamIt) and connecting these by extending the stream structure classes *SplitJoin*, *Pipeline* and *FeedbackLoop*. Filters are directly encoded in StreamIt itself. Neither coordination of legacy code nor a clear separation between computing and coordination layers can be achieved. Access to box implementations and consumer and producer rates enables the StreamIt compiler to optimise scheduling. Hence, StreamIt falls into the category of synchronous dataflow approaches like Lustre [16] and Esterel [17], whereas S-Net advocates asynchronous dataflow.

The separation between coordination and computation in S-Net is closely related to data-driven coordination approaches, of which [18] gives an overview. The earliest related proposal, to our knowledge, for a complete separation as advocated by S-Net, is the coordination language HOPLa [19].

Relating to our modular design of the runtime system, we cite the work on Borealis [20]. The Borealis stream processing engine is based on typed streams of attribute-value pairs (based on [21]), not unlike S-Net records. The values, however, are accessible by Borealis operators and are consequently limited to a pre-defined set of data types. The architecture of the system is composed of several modules which provide a wealth of features as for example load balancing, runtime optimisations and failure recovery. Networks are described in an XML based query language. Compared to S-Net, the description language is rather low-level, but it does not require a compiler as such, as networks can be directly

deployed to a Borealis node. Another recent advancement in stream-based coordination technology is the language Reo [22]. It concerns itself primarily with issues of channel and component mobility and does not exploit static connectivity or type-theoretical tools for network analysis.

In the specific area of functional programming we mention Eden [23], that extends Haskell with process abstraction and process instantiation facilities. Processes communicate via FIFO channels, just as in S-Net, but process topologies are completely dynamic including mobile channels. On the other end of the spectrum we see Hume [24], which combines Haskell-like boxes with synchronous data flow processing. The emphasis with Hume lies in the inference of exact bounds on resource consumption for applications in embedded systems.

9 Conclusion and future work

We have presented the architecture of our implementation of the coordination language S-Net that allows legacy code to be assembled into a streaming network of asynchronous components in a minimally intrusive way. The concept of a common runtime interface proves essential in separating our target-independent compiler from target platform specific runtime system implementations. In the sequel we have put the emphasis on describing a single implementation in-depth: a runtime systems that targets contemporary multi-core processors. We have recognised the distinction between deployment of networks and the operational behaviour of components within the network and explicated their mutual dependence and interplay.

We have developed both a high-level coordination language and a complete, portable tool chain that enables users to harness the computational power of modern multi-core architectures while at the same time they can stick to their familiar (sequential) programming environment for the bulk of an application. Some runtime figures based on an application from the radar signal processing domain demonstrate the effectiveness of our approach.

While the design of S-Net as a language is an area of active research, we can identify three main directions of current and future work with respect to the implementation of S-Net: implementations of the common runtime interface for on-chip microgrids on the one hand and distributed memory workstation clusters on the other hand as well as a refinement of the multithreaded runtime system described here that exercises tighter control on the usage of threads by managing thread scheduling, etc, in the S-Net runtime system ourselves rather than delegating this vital task to the operating system.

Acknowledgements

The development of S-Net is funded by the European Union through the FP-VI Integrated Project Æther, *Self-adaptive Embedded Technologies for Pervasive Computing Architectures,* (www.aether-ist.org).

References

1. Sutter, H.: The free lunch is over: A fundamental turn towards concurrency in software. Dr. Dobb's Journal **30** (2005)
2. Held, J., Bautista, J., Koehl, S.: From a few cores to many: a Tera-scale computing research overview. Technical report, Intel Corporation (2006)
3. Owens, J., Houston, M., Luebke, D., Green, S., Stone, J., Phillips, J.: GPU Computing. Proceedings of the IEEE **96** (2008) 879–899
4. Grelck, C., Scholz, S.B., Shafarenko, A.: A gentle introduction to S-Net. Parallel Processing Letters **18** (2008) 221–237
5. Jesshope, C.: μTc: an intermediate language for programming chip multiprocessors. In: ACSAC'06, Shanghai, China. (2006)
6. Bousias, K., Jesshope, C., Thiyagalingam, J., et al.: Graph walker: implementing S-Net on the self-adaptive virtual processor. In: Æther-Morpheus Workshop: From Reconfigurable to Self-Adaptive Computing, Lugano, Switzerland. (2008)
7. Cai, H., Eisenbach, S., Grelck, C., Penczek, F., Scholz, S.B., Shafarenko, A.: S-Net Type System and Operational Semantics. In: Æther-Morpheus Workshop: From Reconfigurable to Self-Adaptive Computing, Lugano, Switzerland. (2008)
8. Grelck, C., Shafarenko, A.: S-Net Language Report. University of Hertfordshire, School of Computer Science, Hatfield, United Kingdom (2006)
9. Grelck, C., Scholz, S.B., Shafarenko, A.: Coordinating Data Parallel SAC Programs with S-Net. In: 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS'07), Long Beach, USA, IEEE Press (2007)
10. Stephens, R.: A survey of stream processing. Acta Informatica **34** (1997) 491–541
11. Kahn, G.: The semantics of a simple language for parallel programming. In: Information Processing 74, Stockholm, Sweden, North-Holland (1974) 471–475
12. Ashcroft, E.A., Wadge, W.W.: Lucid. CACM **20** (1977) 519–526
13. Broy, M., Stefanescu, G.: The algebra of stream processing functions. Theoretical Computer Science (2001) 99–129
14. Stefanescu, G.: Network Algebra. Springer (2000)
15. Thies, W., Karczmarek, M., Amarasinghe, S.P.: StreamIt: A language for streaming applications. In: Computational Complexity. (2002) 179–196
16. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data-flow programming language LUSTRE. Proceedings of the IEEE **79** (1991) 1305–1320
17. Berry, G., Gonthier., G.: The Esterel synchronous programming language: Design, semantics, implementation. Science of Computer Programming **19** (1992) 87–152
18. Papadopoulos, G.A., Arbab., F.: Coordination models and languages. In: Advances in Computers. Volume 46. Academic Press (1998)
19. Florijn, G., Bessamusca, T., et al.: Ariadne and HOPLa: flexible coordination of collaborative processes. In: Coordination'96, Cesena, Italy. LNCS 1061. (1996)
20. Abadi, D., Ahmad, Y., Balazinska, M., et al.: The design of the Borealis stream processing engine. In: CIDR. (2005) 277–289
21. Abadi, D.J., Carney, D., Çetintemel, U., et al.: Aurora: a new model and architecture for data stream management. VLDB Journal **12** (2003) 120–139
22. Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical. Structures in Comp. Sci. **14** (2004) 329–366
23. Loogen, R., Ortega-Mallén, Y., Peña-Marí, R.: Parallel functional programming in Eden. Journal of Functional Programming **15** (2005) 431–475
24. Hammond, K., Michaelson, G.: The design of Hume: a high-level language for the real-time embedded systems domain. In: Domain-Specific Program Generation. LNCS 3016, Springer (2004) 127–147