Parallel Signal Processing with S-Net

Frank Penczek, Stephan Herhut, Sven-Bodo Scholz, Alex Shafarenko
University of Hertfordshire, United Kingdom
Clemens Grelck
University of Amsterdam, The Netherlands
Rémi Barrère, Eric Lenormand
Thales, France

Abstract

We argue that programming high-end stream-processing applications requires a form of coordination language that enables the designer to represent interactions between stream-processing functions asynchronously. We further argue that the level of abstraction that current programming tools engender should be drastically increased and present a coordination language and component technology that is suitable for that purpose. We demonstrate our approach on a real radar-data processing application from which we reuse all existing components and present speed-ups that we were able to achieve on contemporary multi-core hardware.

1 Introduction

The needs of signal processing rarely receive attention of contemporary computer science. The arsenal of tools for programming DSP and systems on chip tends to be limited to low-level languages, such as C and assembler. Whenever DSP problems grow large enough to warrant an extraordinary hardware and software investment, as they do in massively parallel applications of radar technology, radio astronomy, military communications and control, etc., bespoke tools are being developed (e.g. Thales' SPEAR). The tools target certain types of specialised hardware and help the programmer to express algorithms in a more abstract way that exposes algorithmic properties and promotes concurrency, and a reduction in implementation overheads.

Yet why not consider stream processing from first principles? Its defining features in DSP applications are

- first and foremost, the existence of a static dataflow graph whose nodes represent components that process data streams and whose edges represent those streams;
- the fact that the nodes are effectively stream-processing functions as their output streams depend solely on the data they receive on the input streams;

- computational algorithms as the nodes and the fact that they process arrays of data in some regular manner with a high degree of data parallelism;
- the stream communications, which can be either synchronised and statically defined in terms of their throughput, or asynchronous and highly dynamic, or indeed something in between;

Those are purely technical features whose pre-eminence stems from the fact that data-processing schemes are *naturally* described in terms of some fixed stagewise, highly pipelined application of several algorithms to array data. There are, additionally, other features that reflect general design requirements of any complex system, and those high-end DSP systems tend to be quite complex at the top of their range:

- Adaptability. That is defined narrowly as the ability of a system to tolerate variations in latency/throughput requirements due to changing parameters, and the ability of the processing nodes to receive such parameters from time to time.
- Component reuse. A stream-processing system is almost by definition a componentised system: the nodes are components that represent standard and bespoke algorithms. It should be possible to replace components by their improved versions without destroying the consistency of the links, interfaces and protocols across the component network.
- Abstraction and hierarchical design. Some flavour of encapsulation, inheritance and extension should be supported *by the programming technology/tools themselves*. The challenge is: to do so in a very different setting that is dramatically parallel and devoid of shared global memory. Moreover, this setting, as we argue below should also be permitted to be asynchronous.

Why asynchronous behaviour? Section 6 will enumerate some existing solutions for synchronous stream processing, which are being used in practice or are intended for such use by tool researchers. The common feature of these systems is the static production rate of all nodes and generally a static schedule of all activities. Although quite effective in many scenarios, this approach offers limited adaptability in the situation of node behaviour variation. What is needed is the ability to respond to the readiness of data to be processed by using it as a trigger. That is what asynchrony is all about, and that requires a mechanism of synchronisation and data-driven communication. The HPC community has already wised up to these factors and introduced message-driven computing and one-sided communication. Similar notions are required in stream processing, though they may appear in a different form.

This paper is about using a new tool: The coordination language S-Net[1], for the purposes of asynchronous stream processing. S-Net addresses the technical features listed above and provides convenient solutions for the aforementioned design requirements taking the specific nature of stream processing into

account. S-Net provides for a drastic separation of concerns: the network structure and component interfaces are described hierarchically in S-Net, while the components themselves are reused from the original C application.

This work is a joint project of THALES Research, Paris, France and University of Hertfordshire, UK. The rest of the paper is organised as follows. Section 2 describes the application that THALES Research have provided as a representative example of a high-end DSP problem. Section 3 briefly describes the relevant features of the coordination language. The next section shows how the application can be coded in that language. Section 5 presents a performance comparison between the original application and the S-Net version on multi-core hardware. Section 6 surveys related work, and finally there are some conclusions.

2 The MTI Application

The radar application we present here is an implementation of MTI (Moving Target Indication). The purpose of MTI is to detect moving objects on the ground from an aircraft. The main feature here is the detection of slow moving objects, whereas non-adaptive, classical radar processing is limited to the detection of fast moving objects.

The MTI application (Fig. 2) receives a ground echo of a periodic sequence of radar pulses and attempts to distinguish moving objects from all other, generally still, reflecting surfaces (ground clutter) under the radar beam. The position of a target is estimated by measuring the delay between the transmission of a radio pulse and the reception of its echo. The speed of an object is measured by analysing the Doppler effect that affects echoes of several identical pulses which are sent periodically. The movement of the object results in small variations of its distance to the radar. This distance variation is detectable as a phase shift of the radar signal, e.g. at around 10GHz. In this basic approach, Doppler processing consists of a bench of filters, each tuned towards a particular phase shift between successive echoes. This kind of Doppler processing is in some situations sufficient to separate reflecting objects on the basis of their speeds. When the beam is directed towards the ground, the largest part of the echoed energy is assumed to be a reflection of static objects that compose the ground (clutter). The moving object we are interested in send a weak, phase shifted echo. However, as the radar beam is not perfectly sharp and has a width of a few degrees, some still objects at the border of the beam appear to be moving with a speed relative to the aircraft's speed. This causes undesired interference over the moving target's echoes and creates an ambiguity between intrinsic speeds and azimuths of targets.

More acurate are adaptive filtering techniques, where filters are computed at runtime: In this paper we use an implementation of 'Space Time Adaptive Processing' (STAP) [2, 3], which computes a set of filters from signals received by the antenna array at different time steps. Fig. 1 shows the setup of the antenna array. The antenna array consists of a number (nant) of equidistant

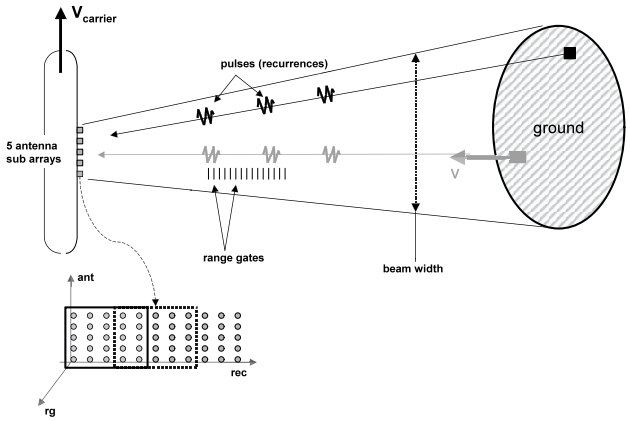

Figure 1: The antenna consists of 5 sensors, which receive a ground echo of the radar beam of periodically sent pulses. The signal is a 3D array of axes ant (sensor), rec (pulse sequence) and rg (range gate).

aligned sensors. The aircraft illuminates the ground with a beam orthogonal to its velocity. The sequences of periodic pulses (bursts) may vary in timing and amount of bursts. The reception time of an echoed pulse depends on the number of the pulse and the distance of the reflecting surface on the ground. Measuring the latter is achieved by sampling the received signal at a given frequency, resulting in the distance being sampled into range gates (rg), which is typically 15 meters for 10MHz sampling. The detected signal is received by the radar processing chain as a 3D array with dimensions $Nrg, Nrec, Nant$. The processing chain is subdivided into independent modules, as shown in Fig. 2.

For simulation purposes, we also implement a 'Stimuli Generation' module, which simulates the signal received by the radar antenna array. This is achieved by computing a 2D array representing the Radar Cross Section (RCS) of the ground surface situated on range gate rg and angle θ. The clutter model of 'CreateClutter' is computed from random, positive values with a given average and adding peak reflectivity values of a given probability. The returned signal from a burst of pulses of the ground surface to which targets with a given RCS and radial velocity have been added, is computed by 'EchoRaf'. The final processing step in this module is the addition of white noise to the signal.

The presented processing chain contains some naive, well-known radar processing techniques for legacy reasons. Nevertheless, the characteristics, i.e. the main challenges from an implementers point of view, are representative for the important industrial domain of embedded signal-processing applications on parallel hardware, as: a) The processing chain uses multiple operators with different requirements on precision and/or dynamic ranges. b) The static processing graph represents a dynamic processing chain, as algorithm parameters, such as array sizes, loop boundaries, etc., change (multi-mode radar [4]). c) The computational load is high enough to require parallel computing hardware. d) Performance is one of the key requirements, both in terms of computational throughput and latency, which may be due to operational requirements or architecture constraints such as memory limitations.

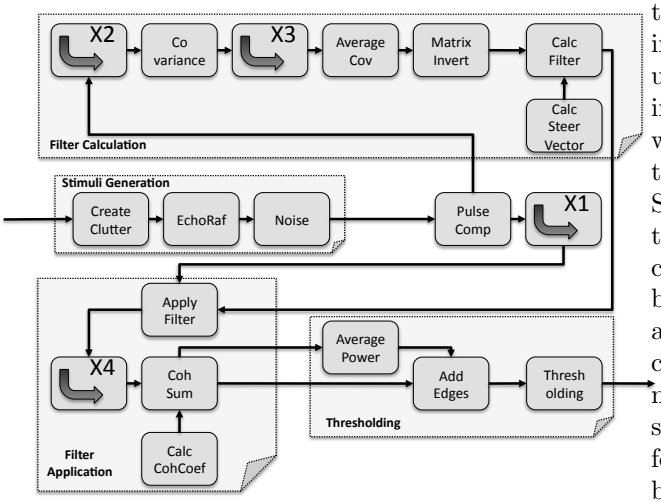

Figure 2: The data processing graph of the MTI application is subdivided into several modules. Modules are indicated by boxes with folded bottom right corners. Small boxes with text denote processing functions, boxes containing an arrow and a capital X with a number denote structure transformers. These boxes are used to re-arrange data in a matrix without effecting the actual values.

3 Introducing S-Net

S-Net is a coordination language based on stream processing. It turns functions written in a computation language (C, for example) into asynchronously executed, stateless stream-processing components, named *boxes*. Each 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 domain that are entirely opaque to S-Net; tags are associated with integer numbers that are accessible both on the S-Net and the box level. Tag labels are distinguished from field labels by angular brackets.

A box expects a record on its input stream to which it applies its associated 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.

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

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

declares a box that expects records with a field labeled `a` and a tag labeled `b`. The box responds with an unspecified number of records that either have just a field `c` or fields `c` and `d` as well as tag `e`. The associated box function `foo` is supposed to be of arity two: the first argument is of type `void*` to cover any opaque data; the second argument is of type `int`.

The box signature naturally induces a *type signature*. Whereas a concrete sequence of fields and tags is essential for the proper specification of the box interface, we drop the ordering when reasoning about boxes in the S-Net domain and turn tuples of labels into sets of labels. Hence, the type signature of box `foo` is `{a,<b>} -> {c} | {c,d,<e>}`. We call the left hand side of this type mapping the *input type* and the right hand side the *output type*. To be precise, this type signature makes `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 $w \in y$. Again, the variant v is a subtype of w iff every label $\lambda \in v$ also appears in w.

Subtyping on input types of boxes raises the question what happens to the excess fields and tags. Subtyping relations would be satisfied if we simply discarded them. Instead, we retrieve excess fields and tags from incoming records and attach them to any output record produced in response to this very input record, unless the respective label is already present in the record. We call this behaviour *flow inheritance*. Note that due to the presence of subtyping, flow inheritance is type-safe as it produces subtypes of the output type, which cannot violate type constraints.

Type inference algorithms developed for S-Net take full account of subtyping and flow inheritance, which can be dealt with statically. In conjunction record subtyping and flow inheritance prove to be indispensable when it comes to making boxes that were originally unaware of each other cooperate 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 connectivity in streaming networks. The restriction of boxes to a single input and a single output stream (SISO) is essential for this. S-Net provides four different network combinators: static serial and parallel composition of two networks and dynamic serial and parallel replication of a single network. These combinators preserve the SISO property, i.e., any network, regardless of its complexity, again is a SISO component.

Let `A` and `B` denote two S-Net networks or boxes. Serial combination (`A..B`) constructs a new network where the output stream of `A` is directed to the input stream of `B`, and the input stream of `A` and the output stream of `B` become the input and output streams of the combined network, respectively. As a consequence, `A` and `B` operate in a pipeline mode.

Parallel combination (`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 combined network. Each network is associated with a type signature. However, unlike box signatures they are inferred by the compiler. Network types control the flow of records in the case of parallel combination. Any incoming record is directed towards the subnetwork 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 parallel and serial combinators have their infinite counterparts: serial and parallel replicators for a single subnetwork. The serial replicator `A*(type)` constructs an infinite chain of replicas of `A` connected via serial combination. The chain is tapped before every replica to extract records that match the type specified as second operand.

The parallel replicator `A!<tag>` also replicates network `A` infinitely far, but this time the replicas are connected in parallel. All incoming records must have the tag specified and the value of this tag decides to which replica a record is sent to.

In practice, we often see boxes that mostly or entirely serve housekeeping purposes, such as renaming, duplication or elimination of fields and tags or simple arithmetic operations on tag values. While all this can be easily accomplished using a user-implemented box, it is often more convenient to do this housekeeping on the S-Net level as it directly affects network construction. The construct we introduce for these purposes is called a *filter* and it looks as follows:

$$[pattern \rightarrow record_1; record_2; \ldots record_n] \,.$$

The type pattern on the left is a set of labels while each of the record specifiers on the right defines the output.

For example, the following filter consumes a record with fields a,b and the tag c and creates two new records: The first record has field a with the original value, field z with the same value and a tag $\langle t \rangle$ set to zero. The second record has fields b with the original value, a with the same value as b and the tag $\langle c \rangle$, whose value is incremented by 1:

```
[{a,b,<c>} -> {a,z=a,<t>};
              {b,a=b,<c>=<c>+1}]
```

There is one "stateful" box in S-Net: the *synchrocell*. It provides the only means in S-Net to combine two existing records into a single one, whereas the opposite direction, splitting a record into two or more records can easily be achieved by any box. Syntactically, a synchrocell consists of an at least two-element comma-separated list of type patterns enclosed in `[|` and `|]` brackets, for example `[| {a,b,<t>}, {c,d,<u>} |]`. The principle idea behind the synchrocell is that it keeps incoming records which match one of the patterns until all patterns have been matched. Only then the records are merged into a single one that is released to the output stream. Matching here means that the type of the record is a subtype of the type pattern. The pattern also acts as an input

type specification of the synchrocell: a synchrocell only accepts records that match at least one of the patterns.

For space reasons we have to refer interested readers to [5, 1] for a more thorough treatment of all language components.

4 Modelling MTI in S-Net

The starting point of the design process of the MTI application in S-Net is the data-flow graph of the original implementation shown in Fig. 2. We use the structure of this graph to derive the structure of our application: Each signal processing function, i.e. the small boxes in Fig. 2, becomes an S-Net box that we build from the existing components. The modules translate to individual networks which connect the boxes using combinators according to the connections within the module. The same combinators also enable us to connect these networks to each other to form the MTI application. This hierarchical approach allows us to implement and test networks, i.e. the modules of the application, independently, as each network is a fully functional application itself when deployed individually.

In the remainder of this section we illustrate the design process of the application modules as networks, using module 'Stimuli Generation' as starting point.

The module in Fig. 2 contains three processing functions, and so does the network we implement. The boxes are arranged in a serial combination (*CreateClutter* .. *EchoRaf* .. *Noise*), implementing the function composition of these funtions.

On first glance, the definition of the box and network signatures is as straight forward as the definition of the network structure. On second glance, however, there is some design space to be explored. The algorithms of these boxes consume a wealth of parameters. As an example, the parameters of *CreateClutter* are shown in Table 1. Most of these parameters are semi-static; they usually do not change during runtime but are only adjusted between runs of the application.

The box returns one single value, a 2D array representing the Radar Cross Section (RCS) of the ground surface. From this knowledge, we may construct the box signature as $\{\sigma\} \rightarrow \{\text{array\_2d}\}$, where σ is shorthand for all entries of the second column of Table 1.

The box *EchoRaf* has a similarly extensive input type. Not only does it require *CreateClutter*'s result, it also requires a set of parameters to produce a 3D array of values representing the ground echo including targets. Box *Noise* is more frugal with respect to parameters and requires only two 3-dimensional matrices of random values to compute white noise, which it applies to the result of *EchoRaf*. If we combine these boxes in sequence to form a network, the parameters propagate to the network signature. This results in an extremely unwieldy input type of the network, composed of field 'rnd_values' plus the union of required parameters of all three boxes. The exposure of local parameters

to the outside is not only unaesthetic, but also problematic from a software engineering perspective: The parameters propagate through any surrounding context and eventually manifest in the global input type of the application network. To avoid this, we confine parameters to their local contexts.

parameter name	static
rnd_values	no
σ_0	yes
rg_{min},rg_{max},rg_{size}	yes
angle_carrier	yes
beam_width	yes
size	yes
targets	yes
$N\theta$	yes

Table 1: Parameters of Create-Clutter

One trivial approach to achieve this, is to generate all required parameters within the box implementation itself. The problem is, that this is not always possible: If a box is provided as an opaque object, e.g. as pre-compiled library from a third party, we do not have access to its implementation. In this case, we have to embed the function call into wrapper code that computes all parameters before invoking the actual box function. By generating the parameters within the box, this approach solves the problem of exploding signatures. The downside of this approach is, that parameters are completely invisible on the S-Net layer. This potentially impedes component reuse, as a user of such a network is now unaware of the presence of the parameters.

Therefore, we take a more flexible approach by generating parameters from within stand-alone S-Net boxes. This is done in such a way, that a parameter generating box does not require any input fields or tags and delivers parameters for one specific box as output. The non-static input is merged to the generating paremeter set by exploiting the power of flow-inheritance: For a box A with signature $\alpha \cup \rho \rightarrow \beta$, where α is the set of input labels and ρ is the set of static parameters, a corresponding parameter generating box P_A has signature $\{\} \rightarrow \rho$. Because of the empty input type, this box can be inserted as predecessor of A without adding any label constraints to surrounding contexts. Any record that arrives at P_A is accepted as input and is enriched by parameter set ρ. More specifically, due to flow-inheritance, a record with label set α as input to P_A results in $\alpha \cup \rho$ (of course, any excess label is carried over as well: $\alpha \cup \gamma \rightarrow \alpha \cup \rho \cup \gamma$).

Using this technique, the original box implementation is left untouched and parameters are visible on the S-Net level without being propagated. In this setting our recently developed reconfiguration and adaptivity features for S-Net [6] come in handy. In short, these features allow us to replace boxes in a deployed network *at runtime*. With these features we offer a solution to maintain semi-static parameters: If new parameter values are required, we replace the generating box by a new version which generates desired results. As this is possible at runtime, multiple parameter sets may be used within the same run of the application. Here, our design decisions become clear: By implementing parameter generators as stand-alone boxes, an adjustment of parameters is merely a replacement of the generator, which leaves the box implementation consuming the parameters untouched if parameter changes occur.

The network that implements the 'Filter Calculation' module of the MTI

```
net Thresholding
{
  box ApplyFilter( (array_3d_signal, array_4d_filter) ->
                      (array_4d_filtered));
  box X5( (array_4d_filtered) -> (array_4d_filtered));
  box CalcCohCoeff( () -> (coh_array_2d));
}
connect [{array_4d_filter, array_3d_signal} ->
            {array_4d_filter, array_3d_signal}; {}]
        .. ( ( ApplyFilter .. X5)
             | CalcCohCoeff)
        .. [|{array_4d_filtered},{coh_array_2d}|];
```

Figure 3: Implementation of the Thresholding network in S-Net

application is a sequence of boxes as defined by the order of tasks shown in Fig. 2. Parameters of boxes are, as above, supplied by parameter generators if required. Special treatment is necessary for boxes *CalcSteerVect* and *CalcFilter*, as the former lacks an input channel, whereas the latter requires two input channels. One possibility to model this, is to assign an empty input type *CalcSteerVect* and place it as direct predecessor of *CalcFilter*. A record that arrives at this box triggers execution of the box without any of the record's fields or tags being read. Flow-inheritance inserts all inbound-record constituents to the resulting record, ensuring that *CalcFilter* receives all required input fields in one record. This approach, however, does not overlap computations where it would be possible: *CalSteerVect* does not require any input, and can therefore begin its computation much earlier. To do this, the box is arranged in parallel to the rest of the network and computation is triggered at the earliest possible moment, i.e. when a record arrives at the first box. The output of the box is then combined to a result record at the latest possible stage, i.e. a synchro-cell merges the box's result to the result of the remaining network. This created record contains all required fields for *CalcFilter*. Fig. 4 (a) and (b) illustrate these techniques.

The remaining networks implementing 'Filter Application' and 'Thresholding' use the same techniques. For brevity we refrain from describing them in detail here but show the concrete S-Net implementation of network 'Thresholding' in Fig. 3 and its graphical representation in Fig. 4(c).

The final step of the implementation phase is to combine all modules to form the MTI application. The complete application network is shown in Fig. 4(d).

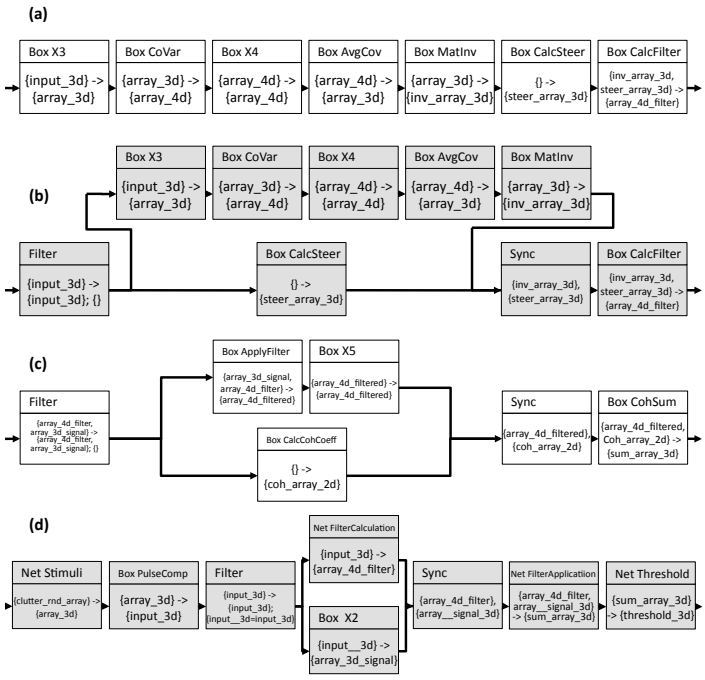

Figure 4: (a) This variant of 'FilterCalculation' uses *CalcSteerVect* as direct predecessor to *CalcFilter*. (b) shows an alternative implementation of (a) where *CalcSteer* is arranged in parallel to the remaining network to overlap computations. (c) shows the implementation of network 'Thresholding'. (d) shows the final MTI application

5 Performance

The measurements we present here compare the original, sequential C implementation with the S-Net implementation that we have developed. Both programs were given several sets of input samples and for each set the total runtime was recorded. The numbers presented in Fig. 5 show the average amount of time that was required to process *one* input sample, computed by dividing the total runtime by the number of data samples in the input set.

We employed two separate machines for measurements: Representative for consumer-grade hardware, we chose a laptop equipped with an Intel Core 2 Duo processor at 2.4GHz and 4GB of memory running Darwin 10.2.0 (Mac OS X 10.6.2). We refer to this host as *Machine A*. The second machine, *Machine B*, is a computation server featuring four Quad-Core AMD Opteron 8356 processors and 16GB of memory running Linux 2.6.18-128.1.10.el5 (CentOS 5.3).

On both machines we conducted the experiments using input data that was identical in size and value. Time was measured from beginning to end of a program run, i.e., including all I/O operations. However, to keep disk access to a minimum we reused input data from memory after reading it in once from disk.

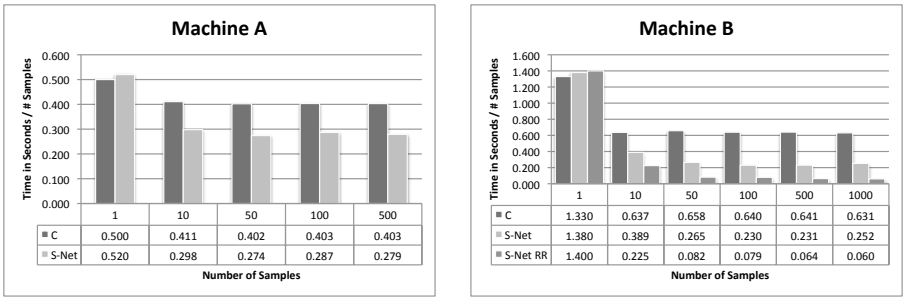

Figure 5: Timings of the original C and S-Net implementation on a dual-core (left) and 16-core (right) machine

As was to be expected, the S-Net runtime system adds runtime overhead whose effect is most prominently seen on the runtimes for one single input sample. This overhead, however, is amortised by the fact that the S-Net runtime system executes multiple stages of the computational pipeline in parallel on systems with more than one core. As both systems contain multiple cores the S-Net implementation outperforms the original C implementation on both machines from as few as five input samples.

The best speed-up on Machine A is about 1.47 which we deem acceptable for a dual-core laptop machine. On Machine B, however, the best speed-up we were able to achieve was 2.78. This dissapointing figure is mainly due to sequential box code which does not take advantage of multiple cores. As these strictly sequential code blocks inevitably limit the achievable speed-up, we have also measured a round-robin distribution of input records to five instances of the network, labelled "S-Net RR" in Fig. 5. This experiment came with almost no additional development cost, as the `!` combinator can be used to achieve exactly this: By applying the split combinator `!` to the outermost level of the network, the runtime system automatically creates multiple instances of the network (demand driven). The combinator determines based on a tag value which instance a data item is dispatched to. In this case, `stap!<n>` requires all data items to carry a tag `<n>` whose value determines the instance. By tagging the input data with `<n>` using values between 1 and 5, and thereby dispatching data items in a round-robin fashion to the instances, a much better utilisation of the computing resources was possible, and hence the speed-up increased to a more satisfying value of 10.5.

6 Related Work

Synchronous data-flow languages, e.g. Lustre [**?**] and Esterel [**?**], are in industrial use today, where they are employed for safety-critical, reactive systems [**?**]. Recent developments in this area include MIT's StreamIt [**?**]. Unlike these,

S-Net is based on asynchronous communication, which allows us to model highly dynamic systems as there is no static schedule of computation, but comes with the price of a more complex memory and communication management.

The coordination aspect of S-Net is related to a large body of work in so-called data-driven coordination, see [**?**]. Unlike most data-driven coordination languages, here we have a *complete* separation of coordination and computation. The earliest related proposal, to our knowledge, is the coordination language HOPLa from the Utrecht University's Areadne project [**?**]. It is a Linda-like [7] coordination language, which uses record subtyping in a manner similar to S-Net, but does not handle variants as we do, and has no concept of flow inheritance. Another early source to mention is the language Sisal [**?**], which pioneered high-performance functional array processing with stream communication. Sisal was not intended as a coordination language, though, and no attempt at the separation of communication and computation was made in it. Still it is important to acknowledge the stream variables of Sisal as an early example of task decomposition using streams. Among more recent papers, we cite the work on the language Eden [**?**] as related to our effort, since it is based on the concept of stream communication. Like S-Net, Eden defines a connection topology for the processing entities; it however deploys the processes completely dynamically and even allows completely dynamic channels. Eden has no provision for subtyping and does not integrate topology with types.

Also functionally based is the language Hume [8]. Hume's conceptual design is not that of a pure coordination language, but a fully-featured programming language, primarily aimed at embedded and real-time systems. Programming in Hume follows a layered approach. Values and functions are defined in a fully-functional expression language, and interaction between functions is defined in a coordination language. The finite-state machine based coordination language connects any desired amount of inbound and outbound "wires" to a function to allow for interaction between the components (i.e. the functions) of a program. Originating from Hume's primary domain and the related necessity for space- and time bound analysis [9], the expression language is an inherent part of the system and cannot be freely chosen as in S-Net. For the same reason, dynamically evolving network structures as are possible in S-Net using serial and parallel replication, are not expressible in Hume.

Another recent advancement in coordination technology is the language Reo [**?**], whose focus is on streams but which concerns itself primarily with issues of channel and component mobility and which does not exploit static connectivity and type-theoretical tools for network analysis.

7 Conclusion

An approach to programming high-end stream processing applications from first principles has been demonstrated. We have shown how an application can be assembled from unit algorithms using a coordination layer. The language S-Net, delineated in the paper, provides an abstract "glue", which connects (existing)

components in a flexible manner, by utilising asynchrony and nondeterminism in order to simplify the representation of connectivity. As the example shows, the network aspect of the application, including its hierarchical encapsulation in a variety of subnetworks, has been given prominence in the designer's view. We hope that prospective users may feel enthused by the elegance and succinctness of the coordination code and a high degree of customisability that S-Net adds to existing code.

Future work will proceed in two directions. We will investigate which impact parallel box code has on performance. An implementation of this application in a functional, auto-parallelising array-processing language [10, 11] is underway and will be experimented with on shared- and distributed memory [12] systems. Secondly, analysis tools will be added to the toolkit to enable the prediction of statistical characteristics of networks, such as latency, throughput etc., which should make it possible to use our approach in intricate real-time and resource-limited settings.

We would like to thank the anonymous reviewers for their constructive comments and suggestions. EU financial support under project Æther, IST-02761 and project Apple-CORE, IST-215216 is acknowledged.

References

[1] C. Grelck, S.-B. Scholz, A. Shafarenko, A Gentle Introduction to S-Net: Typed Stream Processing and Declarative Coordination of Asynchronous Components, Parallel Processing Letters 18 (2) (2008) 221–237.

[2] F. Le Chevalier, S. Maria, Stap processing without noise-only reference: requirements and solutions, Radar, 2006. CIE '06. International Conference on (2006) 1–4doi:10.1109/ICR.2006.343482.

[3] J.-P. Hardange, P. Lacomme, J.-C. Marchais, Radars aéroportés et spatiaux, Masson, 1995.

[4] Y. Hwang, D. Hong, Y. Kwag, Real-time o.s. based radar controller for multi-mode phased array radar system, Radar 97 (Conf. Publ. No. 449) (1997) 558–562.

[5] C. Grelck, Shafarenko, A. (eds):, F. Penczek, C. Grelck, H. Cai, J. Julku, P. Hölzenspies, Scholz, S.B., A. Shafarenko, S-Net Language Report 1.0, Technical Report 487, University of Hertfordshire, School of Computer Science, Hatfield, England, United Kingdom (2009).

[6] F. Penczek, S.-B. Scholz, C. Grelck, Towards Reconfiguration and Self-Adaptivity in S-Net, in: S.-B. Scholz (Ed.), Implementation and Application of Functional Languages, 20th International Symposium, IFL'08, Hatfield, Hertfordshire, UK, Technical Report 474, University of Hertfordshire, UK, 2008, pp. 330–339.

[7] D. Gelernter, A. J. Bernstein, Distributed communication via global buffer, in: PODC '82: Proceedings of the first ACM SIGACT-SIGOPS symposium on Principles of distributed computing, ACM, New York, NY, USA, 1982, pp. 10–18. doi:http://doi.acm.org/10.1145/800220.806676.

[8] G. Michaelson, K. Hammond, Hume: a functionally-inspired language for safety-critical systems, in: Draft proceedings from the 2nd Scottish Functional Programming Workshop (SFP00), University of St Andrews, Scotland, July 26th to 28th, 2000, Vol. 2 of Trends in Functional Programming, 2000.

[9] K. Hammond, Exploiting purely functional programming to obtain bounded resource behaviour: the Hume approach, in: Z. Horváth (Ed.), First Central European Summer School, CEFP 2005, Budapest, Hungary, July 4-15, 2005, Revised Selected Lectures, Vol. 4164 of Lecture Notes in Computer Science, Springer-Verlag, 2006, pp. 100–134.

[10] S.-B. Scholz, Single Assignment C — efficient support for high-level array operations in a functional setting, Journal of Functional Programming 13 (6) (2003) 1005–1059.

[11] C. Grelck, Shared memory multiprocessor support for functional array processing in SAC, Journal of Functional Programming 15 (3) (2005) 353–401.

[12] C. Grelck, J. Julku, F. Penczek, Distributed S-Net, in: M. Morazan (Ed.), Implementation and Application of Functional Languages, 21st International Symposium, IFL'09, South Orange, NJ, USA, Seton Hall University, 2009.