Statistical Performance Analysis with Dynamic Workload using S-NET

Volkmar Wieser*
Software Competence Center
Hagenberg
Softwarepark 21
4232 Hagenberg
volkmar.wieser@scch.at

Philip K.F. Hölzenspies
University of St Andrews
School of Computer Science
North Haugh, St Andrews,
KY16 9AJ
pkfh@st-andrews.ac.uk

Raimund Kirner
University of Hertfordshire
Hatfield, AL10 9AB
United Kingdom
r.kirner@herts.ac.uk

Michael Roßbory
Software Competence Center
Hagenberg
Softwarepark 21
4232 Hagenberg
michael.rossbory@scch.at

ABSTRACT

In this paper the ADVANCE approach for engineering concurrent software systems with well-balanced hardware efficiency is adressed using the stream processing language S-Net. To obtain the cost information in the concurrent system the metrics throughput, latency, and jitter are evaluated by analyzing generated synthetical data as well as using an industrial related application in the future. As fall-out an Eclipse plugin for S-Net has been developed to provide support for syntax highlighting, content assistance, hover help, and more, for easier and faster development. The presented results of the current work are on the one hand an indicator for the status quo of the ADVANCE vision and on the other hand used to improve the applied statistical analysis techniques within ADVANCE. Like the ADVANCE project, this work is still under development, but further improvements and speedups are expected in the near future.

General Terms

Design, Algorithms, Performance, Load Balance

Keywords

S-NET, SAC, ADVANCE, Auto-Parallelization, Image Processing

1. INTRODUCTION

*Corresponding author, for questions please mail to volkmar.wieser@scch.at

Image processing in industrial environments, especially in the field of quality inspection, e.g., for the production of foils, industrial woven fabrics, or stainless steel plates for end-consumer devices, has to cope with a complex phenomenology of textures and defects and in addition with real time requirements on high speed installations, e.g., with an achievable scanning speed up to 300m/min, i.e. about 80MB/sec per camera systems. This requires the application of advanced cost-intensive algorithms for image processing as well as machine learning, the use of high-performance computational hardware like GPUs or multi-core systems, and the exploitation of parallelization potentials. Regarding performance the analysis of the whole processing pipeline in Figure 1 (image acquisition, preprocessing, feature extraction, registration, defect detection and classification) using standard languages is a resource- and time-intensive challenge.

Especially in the field of quality inspection, blob analysis is an elementary step in defect detection (module "Candidate Detection" in Figure 1) to extract features for defect classification, e.g., using support vector machines (module "Defect Classification" in Figure 1). Furthermore, for huge data sets the statistical evaluation of distinct regions in images is a time consuming task.

What is needed is a convenient abstract language that supports automatic parallelization on different architectures without the need of source code changes but gains robust performance benefits on the one hand, and a stream processing language, which orchestrates various numbers of modules to achieve an optimal workload balance, on the other hand.

One of the visions in the IST-FP7 supported ADVANCE project is to realize both an automatic and optimal distribution of workload on heterogeneous platforms to achieve the best performance gain during application run, especially if the amount of input data is changing during inspection, e.g., due to varying numbers or sizes of regions of interest (ROIs). To analyze such a workload behavior we have de-

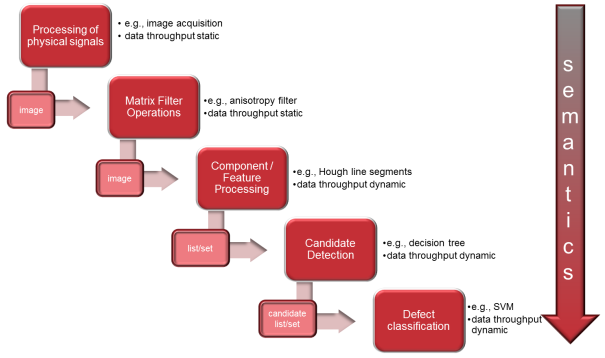


Figure 1: General image processing pipeline for quality inspection

veloped a simple and easy to understand blob analysis tool (see Figure 2) to simulate an industrial use case. First, this tool reads images from an image database and preprocesses them applying an anisotropic filter [7, 11]. In a second step an "ImageLabeler" is used to identify the regions of interest. These two steps must be executed in a sequential order only once. In contrast, the statistical blob analysis itself (calculation of perimeter, area, centroid, compactness, and moments) must be accomplished for each labeled region. Since some modules rely on the output of others, e.g., to calculate the compactness, the results of perimeter and area are necessary, synchronization points are needed, depicted as vertical lines in the lower dashed-bordered rectangle in Figure 2. Finally, the results of all modules are written to the output.

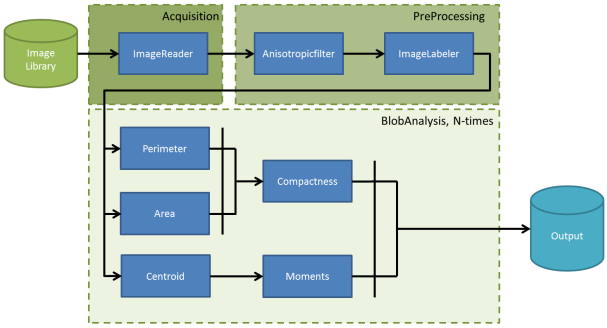


Figure 2: Abstract blob analysis tool for statistical performance analysis

The workload of individual modules as shown in Figure 2 as well as the workload of the used CPU/GPU units is not predictable in advance. This means that in most industrial use cases the occurrence, the size, and the complexity of such regions of interest have an arbitrary behavior. Additionally, the processing time of the single modules in the blob analysis tool differ because of the varied mathematical complexity.

In general, the route within ADVANCE leads towards dynamic specialisation and optimisation of the concurrent application based on feedback of statistic performance data [4]. So far a **L**ight-weight **P**arallel **E**xecution **L**ayer (LPEL), explained in section 3, has been developed for S-Net that improves the efficiency of S-Net application by using user-level threading on top of workers, which are pinned to processing resources. Even though first experiments with LPEL have shown that it can in most cases outperform the traditional implementation based on threads on the operating system level for each S-Net work unit. Towards further optimisation we will work on the optimisation outlined by the ADVANCE agenda. The novel approach we will use is based on property aggregation over individual S-Net work units in order to optimise the resource management. For this we have developed a property description language that allows to characterise individual S-Net work units, called Constraint Aggregation Language (CAL) [9]. We will extend LPEL with a hardware virtualisation technique that exports key parameters of the hardware platform in order to allow for an application-wide resource optimisation based on CAL units and statistical performance data.

In this paper an analysis of the static and dynamic workload of the blob analysis tool in Figure 2 using S-Net is performed with respect to throughput, latency and jitter. The remainder of the paper is structured as follows. After a general overview of the used language S-Net in section 2 and its underlying runtime layer LPEL in section 3, a theoretical explanation of the use cases is given in section 4. Next, a general discussion of the developed S-Net Eclipse plugin is presented in section 5. Afterwards, the S-Net implementation and the practical experiments with the used image database of the applied use cases are shown in section 6 to analyse the results of the statistical workload on a multi-core system. Finally, section 7 summarizes the published work and gives a short outlook of the ADVANCE project.

2. S-NET: ASYNCHRONOUS COMBINATORIAL STREAM PROGRAMMING

Networked stream programming goes back to Kahn's networks [6] which are fixed graphs with message streams flowing along the edges and stream-processing functions placed at the vertices. The importance of this type of computing is in its simple fixed-point semantics and the static nature of task distribution. It is due to these characteristics that networked stream programming is used widely in control systems (for example the Airbus software [2] is written in a stream processing language ESTEREL [1]). However, with the advent of multicore systems and especially large, heterogeneous, many-/multicore architectures, the synchrony found in most programming tools of this kind will become more and more of a limiting factor for throughput and utilization maximization. Consequently asynchronous stream-processing languages, such as S-Net [5] are likely to prove to be useful. The principles behind asynchronous stream-processing can be found in [10]; here we only restate some ideas required to understand the work presented in this paper.

S-Net is a declarative coordination language for asynchronous stream programming. Every network in S-Net is Single-Input, Single-Output (SISO). This means that every network transforms an input stream to an output stream. A stream is a (potentially infinite) sequence of non-overlapping, discrete data items, called *records*. The basic networks are *primitive networks*, that can be combined by using *network combinators* into (non-primitive, SISO) networks.

Primitive networks perform either *processing* or *synchronization*. Processing networks are stateless functions, defined by the user in one of two possible ways: A *box*, implemented in a programming language (referred to as the *box language*), or a *filter*, specified in S-Net terms. Synchronization networks, known as *synchrocells*, combine records based on their type.

Records are sets of name-value pairs. Values come in two variants: *fields*, which are values in terms of the box language and are opaque at the S-Net-level, and *tags*, which are integer values that can be read and written by both the box language and S-Net. The *type* of a record is the set of all the names it contains. A subtype relation on record types is defined as the superset relation on sets, i.e. when record type t contains strictly more names than record type t', then t is a subtype of t'. This subtype relation is transitive, e.g. $\{A, B\}$ is a subtype of $\{B\}$, which is a subtype of $\{\}$, so $\{A, B\}$ is a subtype of $\{\}$ also.

A network takes records from its input stream and results in records on its output stream. Thus, *network types* are defined in terms of record types. Networks take records of one type and result in zero-or-more records of possibly different types. Networks can take different types of records as input. These are referred to as *input variants*. The types of the records on a network's output stream depend on the input variant and (often) also on the *values* contained in the corresponding record on the network's input stream. Thus, for every input variant, a set of *output variants* (record types of records that the network can produce in response to the input) is given.

Networks defined for a specific input type can be fed records of that type or of any subtype thereof. Values corresponding to names not specified in the input type of a primitive network are *flow inherited*, i.e. added to all outputs of that primitive network, produced in response to the corresponding input record. S-Net's type system and the mechanism of flow inheritance provide the user with a powerful compositionality and enable reusability in a broad sense. Furthermore, it provides means for routing records through a network, based on the strongest match (most names) between a record's type and that of the possible networks' input types.

The primary motivation for S-Net is the separation of concerns between application engineering on one hand and concurrency engineering on the other. Also, it creates a portability across different *system* architectures (different granularities of computing resources, memory hierarchies, degrees of heterogeneity, etc.) much in the same way higher level programming languages provide portability across different processor architectures.

3. LPEL

The Light-weight Parallel Execution Layer (lpel) [8] is a user-level work distribution abstraction, which was developed as a run-time system for S-Net. Instead of threads on the operating system level for all independent work units in an S-Net-program, lpel manages a set of *workers*. If possible, one worker corresponds to one processing resource to which it is pinned, so as to exploit cache locality and avoid expensive thread migration. Workers execute tasks, where—on the S-Net-level—one task corresponds to the computation of one box firing for one record. By using user-level threads instead of kernel-level threads, task schedulers can take into account specific (predictable) behaviour of a specific S-Net-program. In other words, it allows for an informed choice of scheduler and for the scheduler, in turn, to be more informed than an operating system scheduler.

Since a key goal of this work is to perform statistical performance analysis, lpel also gathers performance data cheaply and (nearly) transparently. Monitoring can be disabled when not required, to save what little overhead it incurs. One way of ensuring little overhead and transparent measurement of performance is by letting workers use co-operative multitasking, i.e. workers can not be forcefully interrupted. Communication between workers is facilitated by asynchronous message passing via many-writers-single-reader queues. Each worker has one such queue, that it consults between the execution of any two tasks. Because of this predetermined structure of communication, all inter-thread communication can be implemented using concurrent data structures with lock-free techniques. This incurs minimal overheads for the multi-threading coordination.

4. USE CASES

This section gives a brief theoretical introduction on the blob analysis tool and its modules. As shown in Figure 2 the tool has a simple design based on elementary modules which are commonly used in the field of binary blob analysis. A region of interest R as shown in Figure 3 is charaterized by different features to identify R positive. One is the perimeter P of R which is defined as the number of pixels of its contour C. Another is the area A which is defined as the number of pixels in region R (see Figure 2 and Equation 1 and 2). To calculate the coordinates x_s and y_s of the centroid S, the pixels in x and in y direction have to be summed up separately and normalized by A (see Equation 3 and 4). The second moments in Equation 5, 6, and 7 describe the rotation of the region in the defined direction, where σ_{xx}, σ_{yy}, and σ_{xy} are calculated using the standard deviation. Finally, the compactness K in Equation 8 defines the roundness of a region, whereat $K = 1$ denotes a circle and $K > 1$ denotes a line. To calculate the compactness K, P and A have to be known.

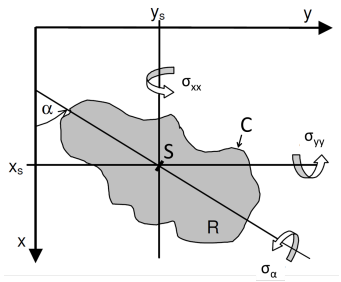


Figure 3: blob analysis

$$P = \sum_{(x,y)\in C} 1 \tag{1}$$

$$A = \sum_{(x,y)\in R} 1 \tag{2}$$

$$x_s = \frac{1}{A} \sum_{(x,y)\in R} x \tag{3}$$

$$y_s = \frac{1}{A} \sum_{(x,y)\in R} y \tag{4}$$

$$\sigma_{xx} = \frac{1}{A} \sum_{(x,y)\in R} (x - x_s)^2 \tag{5}$$

$$\sigma_{yy} = \frac{1}{A} \sum_{(x,y)\in R} (y - y_s)^2 \tag{6}$$

$$\sigma_{xy} = \frac{1}{A} \sum_{(x,y)\in R} (y - y_s) \cdot (x - x_s) \tag{7}$$

$$K = \frac{P^2}{4 \cdot \pi \cdot A} \tag{8}$$

5. S-NET ECLIPSE PLUGIN

The syntax of S-Net makes heavy use of all kinds of brackets and symbols like pipes, exclamation marks, dots and others. Furthermore these brackets and symbols can be combinated which gives them additional meanings. This makes the syntax of S-Net network definitions confusing and error prone. To support the development of S-Net applications a plugin for eclipse has been developed, which provids many features a developer is used to when working with IDEs.

To develop the S-Net plugin, Xtext[1] has been used, which is a kind of language development framework, mainly used to create domain-specific languages. To define the different aspects of a language, Xtext provides a set of APIs and domain-specific languages, within which the grammar language (very close to EBNF) builds the corner stone. Based on that information the core components of the language are generated, which include a parser, an abstract syntax tree, a code formatter, compiler checks and static validation and a code generator or interpreter.

The foundation of the S-Net plugin is the formal definition of the syntax and metadata of S-Net implemented using the grammar language of Xtext. Based on that definition the above mentioned runtime components are generated. These generated components have been enhanced since most of them only provide some basic implemetation. Enhancements include for example the scope definition, the outline view or the syntax highlighting.

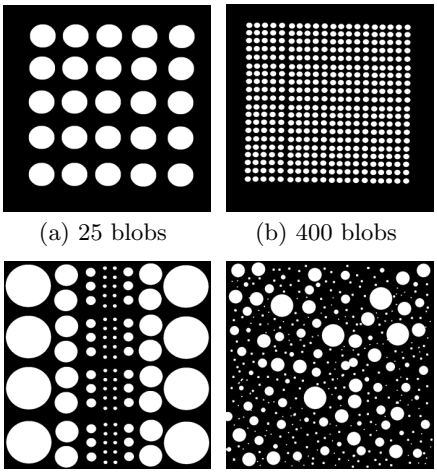
The plugin is still under development, but it already provides features like static validation (including error marking in the editor as ususal in Eclipse), an outline view, code assistance (aka code completion), syntax coloring, code templates, code folding, linking or reference finding. In conjunction with the CDT plugin for C/C++ development the whole S-Net application development can be done using Eclipse.

The vision is to provide a graphical editor for S-Net applications including code generation as known from e.g., tools for class diagram development that generate the source code for the defined classes. In the case of S-Net networks could be developed graphically and subsequently the network definition and the box declarations can be generated automatically for a given box language. Since the runtime components generated with Xtext integrate with and are based on the **E**clipse **M**odeling **F**ramework (EMF)[2], this effectively allows the use of Xtext together with other EMF frameworks like for instance the **G**raphical **M**odeling **F**ramework (GMF)[3].

[1]Xtext: `http://www.eclipse.org/Xtext/`

6. EXPERIMENTS AND EVALUATION

In our experiments we use a small set of images (see Figure 4(a) to Figure 4(d)), to analyse the behavior of S-Net comparing sequential versus parallel execution. The main focus in our evaluation is on throughput (T), latency (L) and jitter (J). The first two images (see Figure 4(a) to Figure 4(b)) include blobs with constant sizes to simulate continuous steady workload whereas the other two images (see Figure 4(c) to Figure 4(d)) include blobs with different sizes, simulating varying workload. The images demonstrate the characteristics of a static versus a dynamic workload within S-Net. Furthermore, to exploit the nature of a stream processing pipeline and to simulate the continued acquisition and inspection process of a real-world application the images are processed 5 times in a loop.



(a) 25 blobs (b) 400 blobs

(c) 88 blobs (d) 389 blobs

Figure 4: Test images for benchmarking

The benchmarks should demonstrate the CPU workload balance using S-Net. For the benchmark tests a SONY VAIO™ PCG-81112M with an Intel® Core™ I7-740QM Processor, 8GB RAM and a NVIDIA GeForce GT 425M graphics card is used. The operating system is Ubuntu 10.10. The used frameworks are OpenCV2.3 [3], SaC_1.00_17510 frameworks, S-Net-1.x.20120110 and lpel-1.x.20120110.

6.1 S-NET Source Code

The S-Net implementation of the blob analysis tool is done as follows

[2]EMF: `http://www.eclipse.org/modeling/emf/`
[3]GMF: `http://www.eclipse.org/modeling/gmp/`

Source code of blob analysis tool using S-NET

```
1  #define REPLICATION_NUMBER 2
2
3  net blobanalysis
4  {
5    box ImageReader( (imagePath, <imageReplicationCnt>) ->
6                     (image, <imageNum>) );
7    box PrintStats( (area, perimeter, compactness, moments,
8                     centroidX, centroidY, <blobNum>) -> );
9
10   net PreProcessing
11   {
12     box AnisotropicFilter( (image) -> (filteredImage) );
13     box ImageLabeler( (filteredImage, <blobReplicationCnt>) ->
14                       (blobList, <blobCount>) );
15   }
16   connect
17     (AnisotropicFilter!<imageNum>) ..
18     (ImageLabeler!<imageNum>)
19   ;
20
21   net Analysis
22   {
23     box SplitBlobList( (blobList, <blobCount>) -> (blob, <blobNum>) );
24     net Properties
25     {
26       box AreaComputation( (blobCalculationCopy) -> (area) );
27       box PerimeterComputation( (blobCalculationCopy) -> (perimeter) );
28       box CompactnessComputation( (areaCopy, perimeterCopy) ->
29                                   (compactness) );
30     }
31     connect
32       [{} -> {<area>}; {<perimeter>}] ..
33       (
34         ([{<area>} -> {}] .. AreaComputation) ||
35         ([{<perimeter>} -> {}] .. PerimeterComputation)
36       ) ..
37       ([| {area}, {perimeter} |] * {area, perimeter}) ..
38       [{area, perimeter} -> {area, perimeter, areaCopy = area,
39         perimeterCopy = perimeter}] ..
40       CompactnessComputation
41     ;
42     net Geometry
43     {
44       box MomentsComputation((blobCalculationCopy) -> (moments));
45       box CentroidComputation((momentsCopy) -> (centroidX, centroidY));
46     }
47     connect
48       MomentsComputation ..
49       [{moments} -> {moments, momentsCopy = moments}] ..
50       CentroidComputation
51     ;
52   }
53   connect
54     SplitBlobList ..
55     [{blob, <blobNum>} -> {blob, blobCalculationCopy = blob, <blobNum>}] ..
56     (
57       [{blob, blobCalculationCopy, <blobNum>} -> {blob, blobCalculationCopy,
58         <blobNum>, <k = blobNum % REPLICATION_NUMBER>}] ..
59       (
60         (
61           [{} -> {<properties>}; {<geometry>}] ..
62                 (
63             ([{<properties>} -> {}] .. Properties) ||
64             ([{<geometry>} -> {}] .. Geometry)
65           ) ..
66           ( [| {area, perimeter, compactness},
67                {moments, centroidX, centroidY} |] *
68                {area, perimeter, compactness, moments,
69                centroidX, centroidY} )
70         ) ! <k>
71       )
72     )
73   ;
74 }
75 connect
76   ImageReader ..
77   PreProcessing ..
78   Analysis ..
79   PrintStats
80 ;
```

The listing above shows the implementation of the blob analysis tool (described in Section 1) using S-Net. As depicted in Figure 2 the tool consists of several modules (blue boxes) which are connected either sequential or parallel according to their dependencies to the results of other modules. Each of these modules corresponds to a box definition (e.g. ImageLabeler in line 6) in the S-Net network. Those modules/boxes hold the sequential code for image reading, filtering, labeling and calculating a blobs properties. Furthermore the boxes are grouped into networks the same way as the modules in Figure 2.

The boxes AnisotropicFilter and ImageLabeler build up the Preprocessing network and are connected sequentially inside of this net. The output of this net is a list of all blobs found in the given image. The Analysis network consists of all boxes responsible for analyzing a blob. This network is split into a Properties network, which calculates the area, perimeter and compactness of a blob, and a Geometry network, to compute the moments and centroid. This way those two sub networks can easily be connected for parallel execution, because those values do not rely on each other. Inside of the Properties network the boxes for area and perimeter computation are also connected parallel. The output of the Analysis network are all the properties of one blob.

After each parallel execution of networks or boxes a synchronization box is placed to synchronize the computed properties of a box, so that values of different blobs do not get intermixed. The box SplitBlobList is an additional box between preprocessing and analysis, which takes the list of blobs produced by the ImageLabeler box and writes the single blobs to the output stream. Since those blobs are independent they can be analyzed in parallel. This is done using the parallel replication combinator (!), which replicates the whole Analysis network dynamically during runtime. The maximum replication number is defined by REPLICATION_NUMBER (defined in the line 1 of the listening).

In the outermost network (bloblanalysis) the ImageReader box, the PreProcessing and Analysis networks are connected sequential, since they rely on the output of each other. Finally the PrintStats box prints the results to standard out.

Source code of area computation S-NET box

```
1
2  void* AreaComputation(void* hnd, /*CBlob blob*/ c4snet_data_t* blob)
3  {
4    void* blobData = (void*)C4SNetDataGetData(blob);
5    unsigned int* area = (unsigned int*)malloc(sizeof(unsigned int));
6    *area = ComputeArea(blobData);
7
8    c4snet_data_t* areaData = C4SNetDataCreate(CTYPE_uint, area);
9    C4SNetOut(hnd, 1, areaData);
10   return hnd;
11 }
```

The code listening above shows an easy but representative example of a box implementation using C as box language. The main part of the implementation deals with handling input and output data of the box. All data fields, except integer values, have to be passed between the boxes using pointers. Additionally these pointers have to be boxed into a special container (*c4snet_data_t*), which holds size and type of data and the pointer to the data itself. To retrieve the pointer from that container, the function *C4SNetDataGetData* has to be used (line 4), in this case the pointer to a *CBlob*. The area of this blob is calculated (line 6) and stored (memory allocated in line 5). The pointer to the area again has to be boxed (line 8) and finally written to the output stream (line 9). The pointer named *hnd* is a pointer to a structure internally used by S-Net, which has to be the first parameter of every box declaration and has to be returned in the end.

For the blob analysis OpenCV and another external library (cvblobslib)[4] have been used, whereas the latter is imple-

[4]CvBlobsLib: `http://opencv.willowgarage.com/wiki/cvBlobsLib`

mented in C++. To be able to use that library, a wrapper in C has to be written and provided as shared library.

6.2 Evaluation - Sequential Execution (SE)

For behavior analysis we have the proposed blob analysis tool implemented with OpenCV2.3 [3], which is a common used library in computer vision and encapsulated with S-Net. First, we analyse the sequential execution of the S-Net blob analysis tool (see Table 1 to Table 3), whereas afterwards the analysis of the parallel execution is done (see Table 4 and Table 6).

Table 1: Throughput (T) of seq. exec. [recs/s]

	Figure:blobs			
Module	4(a):25	4(b):400	4(c):88	4(d):389
Perimeter	3558	7754	5062	7365
Area	3139	5529	4281	6771
Compactness	6452	8349	7259	8319
Centroid	13699	8758	6830	9283
Moments	4097	5832	4922	6360
Total	3139	5832	4281	6360

Table 2: Latency (L) of seq. exec. [ms]

	Figure:blobs			
Module	4(a):25	4(b):400	4(c):88	4(d):389
Perimeter	281	129	198	164
Area	319	181	234	53
Compactness	155	120	138	151
Centroid	73	114	146	114
Moments	244	171	203	223
Total	1072	715	919	705

Table 3: Jitter (J) of seq. exec. [ms]

	Figure:blobs			
Module	4(a):25	4(b):400	4(c):88	4(d):389
Perimeter	410	93	249	164
Area	0	50	24	53
Compactness	229	144	185	151
Centroid	40	106	385	114
Moments	139	277	252	223
Total	818	670	1095	705

6.3 Evaluation - Parallel

S-Net is used to connect the C++/OpenCV modules, which are exemplarily defined in Section 6.1 to a blob analysis composition defined in Figure 2. In detail, the Figure 2 contains the composition "BlobAnalysis" which is for behavior analysis the main focus because these modules must be calculated for each blob, i.e., for n blobs n times. The vision is, if one module has over a period of time a lower throughput or higher latency or jitter the system should provide more resources to it.

Table 4: Throughput (T) of par. exec. [recs/s]

	Figure:blobs			
Module	4(a):25	4(b):400	4(c):88	4(d):389
Perimeter	7771	10642	10167	11148
Area	10846	12141	14934	13872
Compactness	14060	14324	13832	13043
Centroid	7320	19430	16120	19287
Moments	4304	9239	7161	8992
Total	4304	9239	7161	8992

Table 5: Latency (L) of par. exec. [ms]

	Figure:blobs			
Module	4(a):25	4(b):400	4(c):88	4(d):389
Perimeter	311	211	204	197
Area	194	194	142	149
Compactness	150	141	148	170
Centroid	137	106	146	107
Moments	484	220	284	233
Total	1276	872	924	856

Table 6: Jitter (J) of par. exec. [ms]

	Figure:blobs			
Module	4(a):25	4(b):400	4(c):88	4(d):389
Perimeter	250	171	130	133
Area	355	136	110	102
Compactness	179	186	196	210
Centroid	169	86	206	108
Moments	241	145	217	219
Total	1194	724	859	772

7. CONCLUSION

In this article we have shown some first results for the IST-FP7 project ADVANCE. First, we have pointed out the need for high performance image processing in industrial applications and the high demand of abstract modeling tools to support development on heterogeneous platforms with multi-core CPUs or many-core GPUs. As exposed, this support is provided by the stream processing language S-Net. Furthermore, we introduced the **L**ight-weight **P**arallel **E**xecution **L**ayer LPEL which was developed as run-time system for S-Net to gather performance data during application execution. For demonstration, a blob analysis tool has been developed to analyse the static as well as the dynamic run-time behavior of S-Net. The achieved performance results are showing the nature of the sequential and parallel execution by means of throughput, latency and jitter as well as the equal work load on all modules.

In the future, LPEL will be extended with a hardware virtualisation technique that exports key parameters of the hardware platform in order to allow for an application-wide resource optimisation based on CAL units and statistical performance data. Additionally, a vision is to have a graphical editor for S-Net applications including code generation using Xtext together with the **G**raphical **M**odeling **F**ramework.

8. ACKNOWLEDGMENTS

The work has been funded by the EU FP7-project ADVANCE.

9. REFERENCES

[1] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, Nov. 1992.

[2] J. Binder. Safety-critical software for aerospace systems. *Aerospace America*, pages 26–27, August 2004.

[3] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[4] C. Grelck, K. Hammond, H. Hertlein, P. Hölzenspies, C. Jesshope, R. Kirner, B. Scheuermann, A. Shafarenko, I. T. Boekhorst, and V. Wieser. Engineering concurrent software guided by statistical performance analysis. In *Proc. International Conference on Parallel Computing*, Ghent, Belgium, Aug./Sep. 2011.

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

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

[7] P. Perona and J. Malik. Scale-space and edge detection using anisotropic diffusion. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12:629–639, 1990.

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

[9] A. Shafarenko and R. Kirner. CAL: A language for aggregating functional and extrafunctional constraints in streaming networks. Technical report, University of Hertfordshire, Hatfield, UK, Jan. 2011. available at `http://arxiv.org/abs/1101.3356`.

[10] S-NET. S-NET declarative coordination language, 2008.

[11] V. Wieser, C. Grelck, H. Schoener, P. Haslinger, and B. Moser. GPU-based Image Precessing Use Cases: A High-Level Approach. In *Advances in Parallel Computing*. IOS Press, 2011.