FD-COMA 2012

Workshop on
Feedback-Directed Compiler Optimization for Multicore Architectures

Clemens Grelck, Kevin Hammond, Sven-Bodo Scholz (eds.)

7th International Conference on
High-Performance and Embedded Architectures and Compilers
HiPEAC 2012

Paris, France
January 25, 2012
Workshop Programme:

14:30 – 15:30  Key note: Alex Shafarenko:  
Compilers must speak properties, not just code

15:30 – 16:00  Apan Qasem, Dan Tamir: 
Memory Performance Diagnosis Through Feedback Synthesis

16:00 – 16:30  Coffee break

16:30 – 17:00  Wei Cheng, Frank Penczek, Clemens Grelck, Raimund Kirner, Bernd Scheuermann, Alex Shafarenko: 
Modeling Streams-based Variants of Ant Colony Optimisation for Parallel Systems

17:00 – 17:30  Volkmar Wieser, Philip Hölzenspies, Raimund Kirner, Michael Roßbory:  
Statistical Performance Analysis with Dynamic Workload using S-NET

17:30 – 18:00  Vu Thien Nga Nguyen, Raimund Kirner, Frank Penczek:  
Monitoring Framework for Stream-processing Networks
Compilers must speak properties, not just code

[Extended Abstract]

Alex Shafarenko
University of Hertfordshire
a.shafarenko@herts.ac.uk

ABSTRACT
This is a work-in-progress presentation in which we will argue that approaches based on declaratively coordinated components, such as S-Net, are ideally placed for platform-sensitive distributed computing. We will present our case for a component compiler producing not only object code, but also symbolic constraints describing the functional, and more importantly extra-functional, properties of the source component, such as computational complexity, resource and configuration requirements. We also argue that if the coordination language is well structured, it is possible to aggregate the constraints yielded by the components stage-wise and to represent the multicore optimisation problem in standard CSP form to use general-purpose constraint solvers. If successful our approach will achieve an unprecedented degree of flexibility in programming for a wide range of heterogeneous multi-/many-core architectures.

Summary
Coordination language S-Net[1] has been developed and experimented with by an international consortium led by the University of Hertfordshire, UK, since 2006. It is a fully-fledged, purely declarative language unlike most other attempts at coordination. It has a type system that is geared towards supporting message and component hierarchies without interfering with the data abstraction. It has construction mechanisms that support hierarchical networks and make the communication topology a type issue, too. The language has been tried by industry (Thales[3], SAP, Philips) on some relevant example applications with a view to determining its expressivity, usability and efficacy. Some of these experiments are still ongoing. The intention of S-Net is very similar to Intel's CnC[2] in that it provides declarative means for describing the connection between components and specifies their interaction. However, while CnC defines a computation as a collection of interrelated steps, S-Net espouses dataflow philosophy, presenting the overall computation as an asynchronous network of stateless components and non-computing synchrocells.

In this presentation we argue that due to the specific choices made in designing S-Net, chiefly its Single-Input-Single-Output connection principle and the separation of stateful but non-computing synchrocells from stateless components, the coordination layer is in a good position to gather and aggregate information about resource demands and performance expectations from individual components. We propose that these be provided in symbolic (predicate) form.

S-Net components respond to a single input message with zero, one or more output messages, with the input and the output using a single, unidirectional channel each. Since an output message of one component becomes an input message of another, implicative predicates can be chained over connection primitives to aggregate functional constraints over the whole system. Furthermore, the platform can be abstracted as a virtual platform that combines run-time capabilities and configuration awareness, the latter also in predicate form. We propose that the interaction between the virtual platform and the component descriptions should be via a shared set of environment variables, or evars for short. These evars can be named in the implicative predicates that define the functional relationships between inputs and outputs, thus querying and/or constraining the environment in which the component will eventually operate. In doing so, extrafunctional requirements, properties and expectations may also be communicated in and out of components.

When these constraints are gathered, properly aggregated with respect to the coordination structure and then combined with the virtual platform predicates, we expect that the resulting constraint satisfaction problem can be solved completely or partly by an off-the-shelf constraint solver. The solution could then be channeled back to the component compilers to inform them about the system-wide platform-specific situation in order to generate a better code. Thus the conventional compile-link-load scheme (with possible run-time adaptation) is replaced by compile-resolve/adapt-load/configure, where the loading and configuration take into account integrated properties of the whole application, which may include statically defined placement strategies, core choice techniques for a heterogeneous system, scheduling and resource management strategies, etc. All of these can be expressed as platform intelligence in predicate form alongside other resource constraints and configuration parameters. This dramatically augments the concept of virtual hardware, which is currently all but synonymous with API.
We have defined a simple logic programming language (named the Constraint Aggregation Language or CAL by us) for the purpose of communicating component constraints and are working on extending a component compiler with it as well as on a tool for aggregating CAL output over S-Net structures. The main challenge in our approach is in capturing the state of synchronisation data in the proposed representation of functional and extrafunctional constraints. We have some ideas how this could be done: we might associate evars with individual cells and use inductive predicates to describe their regular structures. The advantage of constraint solving over other possible methods is that at worst the solution is too loose to be useful for the component compilers and platform configurers, but it can never be incorrect. In consequence, it may well be possible to involve the programmer (or the coordination designer) in the process of iterative refinement, whereby insufficiently tight constraints are identified interactively and further properties solicited in order to tighten them up. This would require a collaboration with a constraint programming community and extensive experiments with industrially-relevant examples starting with the ones accumulated with the S-Net project over the recent years.

Acknowledgement
The approach detailed above forms the conceptual basis of the ongoing EU Framework 7 project ADVANCE that the author coordinates. The EU financial support to the project is acknowledged.

1. REFERENCES
Memory Performance Diagnosis Through Feedback Synthesis

Apan Qasem
Department of Computer Science
Texas State University
San Marcos, TX, USA
apan@txstate.edu

Dan Tamir
Department of Computer Science
Texas State University
San Marcos, TX, USA
dan.tamir@txstate.edu

ABSTRACT
Harnessing the full potential of complex architectures has been a perennial challenge for the high-performance computing community. The shift towards multicore and manycore architectures has greatly aggravated this problem. With increasing number of cores per socket, deep hierarchies of shared and distributed caches and exa-scale parallelism on the horizon, multicore platforms pose unprecedented challenges for software development and application tuning. Feedback directed optimization (FDO) has emerged as a promising strategy for extracting portable performance out of today’s complex systems. A major challenge in feedback-directed optimization is to find suitable methods for navigating the large and complex multi-dimensional search space of alternate code variants. This position paper describes a strategy for tackling the search space problem by improving the quality of feedback and enhancing its diagnostic capabilities. The key idea is to utilize a wide array of synthesized performance indicators to look deeper into application behavior and identify causes - rather than symptoms - of performance loss. The insight gained from this inspection is used to both guide the search process and prune away unfavorable search space regions on-the-fly. The paper highlights key features of the proposed framework and justifies the use of enhanced feedback in an FDO system.

1. INTRODUCTION
The memory hierarchy in modern computer systems plays a crucial role in application performance and much of the responsibility of exploiting the memory hierarchy lies with software tools like the compiler. The advent of multicore systems has both increased the importance of the memory hierarchy in application performance and also created new challenges in its exploitation. Generating efficient code on a chip multiprocessor system requires a deep understanding of the structure and organization of its complex memory hierarchy. The combination of shared and private caches along with different coherence protocols on different machines make efficient utilization of cache resources a daunting challenge. For this reason, recent research in feedback-driven optimization has had a strong emphasis on the memory subsystem [33, 6, 10, 31]. Many different strategies for efficient and effective feedback-driven optimization have been proposed. These include applying fast and sophisticated search algorithms [4, 21], limiting program execution time through checkpointing [6] or static estimation [31], using domain-specific knowledge for guidance [33], developing new machine learning algorithms [10, 23, 29], and using analytical models for both pruning search spaces and developing new search heuristics [12, 21, 22, 31]. In spite of these efforts, optimal or near-optimal utilization of the memory hierarchy still requires a fair amount of manual tuning in many cases. One aspect that has received relatively less attention in this context is the collection and synthesis of feedback metrics. Our proposed framework aims to close this gap.

The diagnostic capabilities of feedback metrics play an important role in the success of any FDO-based strategy. Focusing on one particular metric provides a myopic view of application performance and does not lead to effective code transformation. This is particularly significant when optimizing for the memory hierarchy. Many aspects of memory performance can have an impact on overall application performance, including low bandwidth utilization, excessive register pressure, increased cache misses, false sharing and shared-cache pollution. Not all of these issues manifest themselves in every program and even if they do, their relative impact on performance is often hard to determine through raw performance data. Thus, a smart optimizer needs to isolate these effects and pinpoint the bottlenecks so that it can apply the most suitable transformation. For example, if the optimizer is informed that a program suffers from a high number of cache misses then it can attempt a bevy of techniques to alleviate the situation, including tiling, loop interchange, unroll-and-jam, fusion and array padding. However, if the nature of the misses is revealed to the compiler then it can apply a more focused strategy. For conflict misses the optimizer may adjust padding factors, while for capacity misses it may re-evaluate the tile sizes.

In this paper, we propose an FDO framework where efficiency is achieved through enhanced knowledge of the problem domain, program features and architectural characteristics. To this end, we first develop and extend a set of tools that allow specification, collection and synthesis of optimization related information. The key idea is to utilize a wide array of performance metrics to look deeper into application
behavior and identify the causes - rather than symptoms - of performance loss. The insight gained from this inspection is used to both develop heuristics for search strategies and create improved feature sets for machine learning algorithms. Figure 1 outlines our proposed information-rich FDO framework. The framework consists of several components seen in a typical autotuning or FDO system, with special emphasis on information gathering. The gathering of information permeates through multiple phases. At the very onset, before the input program is fed into the system, a specification language, allows domain experts to express characteristics of the problem domain. The information gathering continues in the program analysis phase, where a feature extraction tool extracts program features that are subsequently used in ML-based modeling and pruning of the search space. A key novelty in the feature extraction mechanism is the inclusion of both high-level and dynamic architecture-sensitive attributes. The next step in information gathering occurs during program execution when performance feedback is collected using HW performance counters.

Although the idea of enhanced information touches all parts of our framework, in this paper, we limit the discussion to enhancements specifically tied to the collection and analysis of feedback during program execution. We propose three types of enhancements to the feedback mechanism: (1) granularity (2) range and (3) diagnostic capabilities. The notion of feedback diagnostics is particularly compelling as it provides a fresh perspective on the optimization search space and has potential for dramatically reducing its size.

2. ENRICHING PERFORMANCE FEEDBACK

We propose enhancements to three aspects of feedback collection and analysis. In this section, we provide justifications for each type of enhancement. The implementation of the feedback enhancement ideas rely heavily on HPC-Toolkit [2], an open-source tool suite for performance measurement and analysis. In addition, we develop a new tool, Feedback Parser and Synthesizer (FeedSynth) that parses, synthesizes and delivers to the search engine, the large collection of performance metrics produced by HPC-Toolkit or similar software. The code transformer and search engine is extended and modified to process this enhanced performance data.

2.1 Sharpened Granularity

Inspecting performance metrics at program level is sufficient when tuning small, sequential kernels, where one loop nest dominates the entire execution. However, for larger, parallel applications, where execution time is distributed across thousands of threads, procedures and loops, whole program granularity proves to be too coarse. Currently, the use of fine-grain performance metrics in state-of-the-art FDO systems is limited to the triage phase, where procedure-level feedback is used to identify hot code segments [6]. This, however, is a missed opportunity, since fine-grain feedback can play a much more significant role in tuning applications. Consider the performance variations in three major loop nests in the NAS MG benchmark, as shown in Fig. 2. For the same block sizes (e.g., 8 and 40), the relative performance of the three nests varies significantly. In this scenario, any search strategy, based on whole program execution time (which shows less variation for MG) will be inherently flawed, as it would attempt to vary optimization parameters without knowledge of how the change is affecting different portions of the code. This situation can be mitigated by constructing a partitioned search space for the application and by including the execution time of each loop nest as feedback, during the search process.

In the proposed FDO framework, we provide support for collection of feedback metrics at thread-, procedure-, loop- and statement-levels and allow construction and exploration of search spaces at each of these levels. To do this, we leverage the capabilities of HPC-Toolkit, which profiles application binaries and correlates performance data back to source code structures, such as procedures and loops. The correlated profile data is output in XML format, and sent to a performance database for subsequent inspection using a viewer supplied with the toolkit. We are currently implementing an interface between FeedSynth and HPC-Toolkit that allows us to capture this information and make it available to other components within the framework.

2.2 Increased Range

Modern microprocessors provide a wealth of information on application performance through a large set of HW performance counters. For instance, AMD Phenom exposes 119 major native counter events, while Intel Nehalem exposes over 130. Software for probing these counters has matured significantly and counter values are now frequently used in manual tuning and performance modeling [15, 30]. Looking
at a variety of different metrics allows performance tuners to pinpoint problem areas and pick the right mode of attack to remedy the situation. For instance, a sudden spike in register spills might indicate that the loop unroll amount is too large and needs to be reduced for better performance. We develop a new scheme that allows us to transplant similar strategies in the realm of feedback-directed optimization. Specifically, we extend the FeedSynth-HPC Toolkit interface to harness the comprehensive set of performance counter values extracted by HPCToolkit and PAPI. FeedSynth then channels this information to the search module.

### 2.3 Improved Diagnostics

Fine-grain and auxiliary performance data provides the building blocks for an innovative approach of feedback gathering in which performance metrics are collected and systematically transformed into performance diagnostics. The strategy allows search algorithms to make fine-tuned and focused decisions that previously could only be made by human experts [9]. Fig. 3 outlines our approach for feedback synthesis. The collected metrics are piped through three levels of synthesis as they become progressively more complex and informative. Level 1 uses formulas based on linear combinations of metrics, level 2 combines machine information and accumulated instruction and data address of hardware events, and finally level 3 utilizes source code characteristics to produce the set of performance diagnostics. The use of data addresses associated with counter events, is a major move forward because it provides information about the nature of a cache miss; an insight that can be of immense value for feedback-directed optimization.

Our preliminary investigation shows that the diagnostic metrics listed in Table 2.3 can be synthesized from performance counter values and is currently the main focus of this work. We believe this approach is general enough to automatically diagnose a range of performance issues. Upon successful integration of the diagnostics in Table 2.3, we plan to expand the framework to include metrics that relate to stalled cycles and CPI.

<table>
<thead>
<tr>
<th>Component</th>
<th>Diagnostic Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache</td>
<td>miss classification: compulsory, capacity, conflict (un)exploited locality true and false sharing, redundancy, pollution</td>
</tr>
<tr>
<td>Memory</td>
<td>bandwidth utilization [15]; contention (socket-level)</td>
</tr>
<tr>
<td>TLB</td>
<td>conflicts and thrashing inter-core contention</td>
</tr>
<tr>
<td>HW Prefetcher</td>
<td>utilization</td>
</tr>
</tbody>
</table>

### 3. RELATED WORK

The idea of feedback directed optimization can be traced back to Knuth’s 1971 study of Fortran programs [20]. The proliferation of new architectural models has rekindled interest in this area. Work in FDO can be broadly classified into two categories based on their scope: strategies that work on domain specific kernels and those that attempt to tune whole applications.

A number of successful empirical tuning systems provide efficient library implementations for important scientific domains, such as those for dense and sparse linear algebra [33, 14, 8], signal processing [16, 27] and tensor contraction [7]. Among these, ATLAS [33], is the most widely used within the scientific community and has become the de facto standard for evaluating other autotuning systems. The ATLAS model has even found its way into commercial compilers in the form the Math Kernel Library (MKL) distributed with the Intel compiler suite [1]. Unlike ATLAS, the SPIRAL [27] and FLAME [8] projects have looked at the problem at a higher level and concentrated more on the issue of algorithmic choice rather than exploring options of alternate implementations of the same algorithm. In the SPIRAL framework, signal-transform routines are expressed by mathematical formulas using a special purpose language [34] and a suitable implementation is chosen based on matrix factorization calculations and a simple sequential search. More recently, the Peta Bricks project has adopted the SPIRAL and FLAME approach for general algorithmic tuning [5]. Although the proposed methods of enhanced feedback-driven search can be used for tuning domain-specific libraries, we do not explicitly address this issue.

Research efforts in whole application tuning can be broadly classified into two categories based on the parameter search space on which they operate. Several ongoing research projects

![Figure 2: Execution time variation of loop nests in NAS MG, as a function of block size](image)
tackle the phase-ordering problem using empirical methods [4, 21, 25, 31, 24]. That is, they aim to find the best sequence of transformations that minimizes some objective function such as execution time or power. On the other hand, some of the work in autotuning concentrate on finding the best parameter values for transformations that use numerical parameters [11, 13, 19, 28]. More recent efforts strive to combine the two methods to provide a more unified solution that involve compile-time tuning with source-to-source transformations and runtime tuning and optimization [6]. Our proposed methods are aligned with this integrated approach of autotuning, since the enhancements to the feedback mechanism can be utilized for both offline and online search.

The issue of model-guided tuning has been approached from several different angles. Most notable among these is the use of compiler-based analytical models in limiting the search space [12, 31, 21, 35, 28]. Chen et al. showed the analytical models can significantly cut down the search space for a set of transformations including tiling, loop interchange and unroll-and-jam [11]. Qasem and Kennedy have used models for pruning the combined search space of loop fusion and tiling [28]. The Active Harmony project focuses on runtime optimizations and use analytical models to establish an ordering of transformations rather than reducing the size of the search space [18]. The OSE compiler uses static heuristics for generating a pruned search space for optimization sequences [31]. Kulkarni et al. use techniques such as detecting redundant sequences and identifying equivalent code to cut down the number of program evaluations [21].

Apart from compiler models, machine learning techniques have been applied to tune unroll factors [29] and also for selecting the best optimization set (without reordering) [3]. There has been some work in using statistical models to explore the search space of optimization parameters. Vuduc et al. establish an early stopping criteria to eliminate less promising search space regions on-the-fly [32]. Pinkers et al. use a statistical method based on orthogonal arrays to choose the optimal sequence of transformations [26]. More recently, Greck et al. have used statistical analysis to isolate performance bottlenecks in application components. This information is then utilized to eliminate the bottlenecks dynamically [17].

The proposed method of enhanced feedback driven optimization can be used in conjunction with existing pruning strate-

Figure 3: Feedback Synthesis for Performance Diagnosis


gies to speed up the overall tuning process. Moreover, these methods can also be used to validate or verify static estimators, and statistical or analytical models.

4. CONCLUSIONS AND FUTURE WORK

This paper describes an FDO framework that focuses on the feedback module and aims to improve the diagnostic capabilities of the collected performance measurements. Although we focus on just the memory subsystem, the framework can be extended to cover other aspects of performance including stalled cycles and inter-thread communication. Our future plans include a complete implementation of the system to include these other features.

5. REFERENCES


Jose, CA, USA, March 2005.


ABSTRACT
In this paper we present the implementation of a concurrent ant colony optimisation based solver for the combinatorial Single Machine Total Weighted Tardiness Problem (ACO-SMTWTP). We introduce S-Net, a coordination language based on dataflow principles, report on the performance of the implementation and compare it against a sequential and a parallel implementation of the same algorithm in C. As the workload of the optimisation algorithm is highly irregular we consider this application to be an important use-case for runtime measurement directed optimisations of the co-ordination program as much as for guiding optimisations of numerical code.

1. INTRODUCTION
ACO is a meta-heuristic inspired by the foraging behaviour of ants [5]. A number of artificial ants iteratively construct solutions to a given combinatorial optimisation problem. Ants are thereby guided by so called pheromone information that previous ants, which have found good solutions, have disposed to mark their decisions in the solution construction process. Since the artificial ants construct their solutions independently and since the core of the algorithm consists of iteratively repeated instructions, ACO is very attractive for parallel execution on multi processor architectures (for an overview see [18]). The large majority of parallel ACO implementations were presented during the last ten years. These implementations mainly follow two distinct parallelisation approaches. In the master-slave approach, a master processes global information (like updating pheromone information, or determining globally best solution) and a number number of slaves (or workers) execute subordinate tasks (like constructing solution, or calculating fitness). The other main group follows a multi-colony approach where a number of colonies search for good solutions using their own pheromone matrices while exchanging information in certain time intervals.

In this publication, we discuss dataflow-oriented variants of parallel ACO algorithms targeting multi-core systems. A new approach to modeling such algorithms as a stream-
can be changed from time to time to suit different optimisation agendas. Here we have the combination of a software engineering challenge: keeping the building blocks of the application suitably encapsulated and abstracted, and a distributed computing challenge: supplying the necessary resources to those blocks and properly connecting them with sufficiently low-latency, high throughput data streams. This is where the coordination technology introduced with S-Net comes into play. S-Net [10] was designed for keeping the two concerns separate while at the same time enabling the application designer to focus on the distribution aspect of the code. Being a coordination language as well as a component technology it allows different distribution solutions to be expressed abstractly, in network form, as well as properly instrumenting the components and profiling the application run-time behaviour.

The main form of glue characteristic of S-Net is a single connecting stream. The local state of computation is embodied in a record that floats down that stream from one component to another. The receiving component has the ability to process a part of the data contained in the record while not caring and having no knowledge of any other parts, which is a property called "flow inheritance" in S-Net lingo. An output record from a component will retain a copy of such parts (hence "inheritance"), which is explained in Section 2. In contrast to OOP, this form of inheritance allows for subsequent stages of a cascaded processing scheme to access the information not affected by earlier stages locally thus avoiding the linkage between hierarchical and physical remoteness, characteristic of object-based solutions.

The ability of S-Net to float the local computation state, while preserving inheritance, also makes processing components virtually stateless, since any state they may require is appended to the state record without being exposed to irrelevant components (that is taken care of by the type system, which checks that only those items that are declared by a component’s type signature is ever affected by the component). Thus the state record, which can be circulated around a component, turns a temporal sequence of state transitions as it is observed normally within statefull objects, into a "spatial" stream of state records.

What is the advantage of such an approach? Indeed at first glance there are only disadvantages: a seemingly higher storage demand, the need to keep inherited baggage local to its consumer, while at the same time avoiding excessive copying, etc. However, these problems are well understood and satisfactory solutions are available [8]. The biggest gain of S-Net technology is its ability to place work at zero cost on a distributed system. This comes from the fact that the S-Net components are stateless and hence can be replicated at zero cost and placed anywhere, including at multiple sites in a distributed system. The real cost is, of course, storage, not primarily the storage for records, which is highly optimised by reference-counting and intelligent caching, but synchronisation storage, i.e. memory for records that need to be joined with other records yet to be produced. Again, the unique advantage of S-Net is the fact that synchronisation storage is componentised, and that those components are devoid of domain-specific semantics, being fully defined by the coordination language. It is therefore quite possible to manage, instrument and profile synchro-storage in a manner independent of the evolving code of processing components (called "boxes" in S-Net lingo).

This paper will demonstrate how separation of box and coordination concerns, inheritance-based component hierarchies and encapsulated synchro-storage help to design a more expressive, better manageable and more easily profileable ACO code. The remainder of the paper is organised as follows: Sections 2 and 3 elaborate on the S-Net coordination technology and the corresponding toolchain. Sections 4 and 5 introduce the concept of ant colony optimisation in greater detail and explain our S-Net implementation. We report on extensive experiments with this S-Net implementation of ant colony optimisation on a 48-core server system in Section 6. Section 7 sketches out some related work before we draw conclusions in Section 8.

### 2. S-Net in a Nutshell

S-Net is a high-level, declarative coordination language based on the concept of stream processing. As such S-Net promotes functions implemented in a standard programming language into asynchronously executed stream-processing components, coined boxes. Both imperative and declarative programming languages qualify as box implementation languages for S-Net, but we require any box implementation to be free of state on the coordination level. More precisely, a box must not carry over any information between two consecutive activations on the streaming layer.

Each box is connected to the rest of the network by two typed streams: one for input and one for output. Messages on these typed streams are organised as non-recursive records, i.e. sets of label-value pairs. The labels are subdivided into fields and tags. The fields are associated with values from the box language domain; they are entirely opaque to S-Net. Tags are associated with integer numbers, which are accessible both on the coordination and on the box level. Tag labels are distinguished from field labels by angular brackets.

Operationally, a box is triggered by receiving a record on its input stream. As soon as that happened, the box applies its box function to the record. In the course of function execution the box may communicate records on its output stream. Once the execution of the box function has terminated, the box is ready to receive and to process the next record on the input stream.

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

\[
\text{box foo } ((a,<b>) \rightarrow (c) | (c,d,<e>));
\]

declarates a box foo that expects records with a field labelled a and a tag labelled b. The box responds with an unspecified number of records that either have just 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 \text{void*} to qualify for any opaque data; the second argument is of type \text{int} as the joint interpretation of tag values by the coordination and the box/application layer.

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.
Consequently, this step turns tuples of labels into sets of labels. Hence, the type signature of box foo is
\[ \{a, <b>\} \rightarrow \{c\} | \{c, d, <e>\} . \]

We call the left hand side of this type mapping the input type and the right hand side the output type. We use curly brackets instead of round brackets to emphasise the set nature of types.

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 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 input types of boxes raises the question what happens to the excess fields and tags. S-Net supports the concept of flow inheritance whereby excess fields and tags from incoming records are not just ignored in the input record of a network entity, but are also attached to any outgoing record produced by it in response to that record. Subtyping and flow inheritance prove to be indispensable features when it comes to make boxes that were designed in isolation collaborate in a streaming network.

It is a distinguishing feature of S-Net that it neither introduces streams as explicit objects nor that it defines network connectivity through explicit wiring. Instead, it uses algebraic formulae to describe streaming networks. The restriction of boxes to a single input and a single output stream (SISO) is essential for this. S-Net provides four 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: any network, regardless of its complexity, is an SISO entity in its own right.

Let \( A \) and \( B \) denote two S-Net networks or boxes. Serial combination \((A \cdot B)\) constructs a new network where the output stream of \( A \) becomes 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.

Parallel combination \((A|B)\) constructs a network where 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. The type system controls the flow of records. Each operand network is associated with a type signature inferred by the compiler. Any incoming record is directed towards the operand network whose input type is better matched by the type of the record. If both operand network’s input types are matched equally well, one alternative is selected non-deterministically. Parallel composition can be used to route different kinds of records through different branches of the network (like branches in imperative languages) or, in the presence of subtyping, to create generic and specific alternatives triggered by the presence or the absence of certain fields or tags.

The parallel and serial composition combinators have their infinite counterparts: serial and parallel replication combinators for a single operand network. The serial replication

\[
\text{A*type constructs an infinite chain of replicas of A connected by serial combinators. The chain is tapped before every replica to extract records that match the type specified as the second operand. More precisely, the type acts as a so-called type pattern: pattern matching is defined via the same subtype relationship as defined above. Hence, a record leaves a serial replication context as soon as its type is a subtype of the type specified in the type pattern. The parallel replication combinator A|<tag> also replicates network A infinitely, but this time the replicas are connected in parallel. All incoming records must carry the tag <<tag>>. This tag’s value determines the network replica to which a record is sent.}
\]

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: 

\[
\text{pattern} \rightarrow \text{record}; \text{record}; \ldots \text{record};
\]

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 filter

\[
\{(a, b, <c>) \rightarrow \{a, z=a, <t>\}; (b, a=b, <c=c+1>)\}
\]

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 \(<t>\) set to zero. The second record has fields \(b\) with the original value, \(a\) with the same value as \(b\) and the tag \(<c>\), whose value is incremented by \(1\):

While any box or filter can split a record into parts, we so far lack means to express the complementary operation: merging two records into one. For this purpose, S-Net features dedicated synchrocells. A synchrocell has the syntactic form \([\text{type}, \text{type}]\). 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.

![Figure 1: Illustration of the four S-Net network combinators](image-url)
by a runtime system. The extremely simplified behaviour of
synchrocells captures the essential notion of synchronisation in
the context of streaming networks. More complex synchron-
isation behaviours, e.g., continuous synchronisation of
matching pairs in the input stream, can easily be achieved
using synchrocells and network combinators. Details can be
found in [7].

To summarise S-Net is an abstract notation for streaming
networks of asynchronous components. It is a notation that
allows programmers to express concurrency in an abstract
and intuitive way without the need to reason about the typ-
ical annoyances of machine-level concurrent programming.
Readers are referred to [8, 10] for a more thorough presenta-
tion of S-Net and to [11, 19] for other case studies on
application programming with S-Net.

3. S-NET TOOL CHAIN

Fig. 2 illustrates the multi-layered architecture of the S-Net
toolchain. The S-Net compiler snect reads S-Net source
code and checks it for lexicographic and syntactic correct-
ness. The core of the compiler is the type inference system that
associates any combinator subexpression with an S-Net
type that defines routing of records through the network.
The S-Net compiler emits fairly high-level C code with calls
to S-Net library functions, the common runtime interface.
In fact, the C code still very much resembles the original
S-Net source code with individual library functions imple-
menting boxes, filters, synchrocells and the network combi-
nators, properly parameterised for the concrete application.

The common runtime interface features several implemen-
tations; in this paper we focus on the multithreaded runtime
system [9]. It consists of two layers that are mutually depen-
dent: the deployment layer sets up a system of asynchronous
components communicating via bounded buffers in shared
memory. At this level, combinators are resolved into split
and merge style components. The component layer imple-
ments the dynamic behaviour of the various S-Net compo-
nents. Both layers depend on each other as networks with
replication combinators dynamically evolve.

The component layer of the runtime system is based on a
number of separate auxiliary modules that manage records,
implement type pattern matching, realise streams as buffers
and, last not least, control the interfacing of S-Net with the
outside world on the global begin and end of the stream-
ing network and towards the box languages used to im-
plement the boxes. The most important of these modules is
the threading layer that controls low-level thread man-
agement. We currently have two implementations of the
threading layer: one maps each S-Net component to its
own Posix thread and, thus, leaves the scheduling of S-Net
components to the operating system. The more elaborate
threading layer implementation, named LPEL [23], actively
manages S-Net components as non-preemptive tasks with
low-overhead scheduling of tasks to a fixed small number
of Posix threads for effective utilisation of multi-core pro-
cessors. This threading layer is used in the experiments
reported in Section 6; it provides ample opportunities for
runtime profiling.

4. ACO ALGORITHM

This section briefly introduces the structure and operation
of a typical sequential ACO algorithm from a generic per-
spective for static combinatorial optimisation problems (for
a more detailed introduction to ACO refer to e.g. [5]). The
description conforms with the conventional (non-streaming
oriented) programming style. As shown in Algorithm 1, Line
1, the algorithm begins by initialising problem-dependent
parameters (e.g., calculating evaluation parameters, setting
initial pheromone values etc.). This is followed by an inter-
active body, where a number of m ants repeatedly construct
solutions (Line 4) by making a sequence of local decisions,
e.g. successive selections of items in the solution vector.
Every decision is made randomly according to a probability
distribution over the so far unchosen items in selection set
S and depending on pheromone information and heuristic
information. Pheromone information is encoded in an
$n \times n$ pheromone matrix $[\tau_{ij}]$. Pheromone value $\tau_{ij}$ expresses
the desirability to assign an item $j$ to place $i$ in the solution vec-
tor. Ant decisions are further supported by problem-specific
heuristic information $\eta_{ij}$.

Assuming the ant is positioned in row $i$ of the pheromone
matrix (assigning the $i$-th item in the solution vector), with
probability $q_0$ the ant makes a deterministic decision and
with probability $1 - q_0$ it makes a random decision: De-
terministic decision: Item $j \in S$ is selected which max-
imises $\tau_{ij} \cdot \eta_{ij}^\alpha$. Random decision: With probability $p_{ij}$ item
$j \in S$ is selected with $p_{ij} = \frac{\tau_{ij}^\alpha \eta_{ij}^\beta}{\sum_{k \in S} \tau_{ik}^\alpha \eta_{ik}^\beta}$ and $p_{ij} = 0$ if
$j \notin S$. Parameters $\alpha$ and $\beta$ determine the relative influence
of pheromone values and heuristic values.

At the end of an iteration, when $m$ solutions have been gen-
erated, the best solution $\pi^*$ of all iterations (global-best solu-
tion) is determined (Line 6) which is used to update the
pheromone matrix (Line 7): $\tau_{ij} := (1 - \rho) \cdot \tau_{ij} + \rho \cdot \Delta_{ij}$. Com-
monly, increment $\Delta_{ij}$ enforces pheromones along the trail of
the best solution, i.e. $\Delta_{ij} > 0$ if $\pi^*(i) = j$ and $\Delta_{ij} = 0$
otherwise. Parameter $0 < \rho \leq 1$ models the pheromone
evaporation rate. The ACO algorithm executes a number of
iterations until a specified stopping criterion has been met,
5. STREAMS-BASED ACO

This section describes various approaches to transferring sequential ACO algorithms into parallel streaming-based variants. ACO algorithms have been chosen as examples because they are well-suited for parallelisation and have been used in a variety of applications.

5.1 ACO in S-Net

Modelling the discussed ACO algorithm in S-Net is accomplished with minimal effort. The stages of Algorithm 1 are implemented as boxes; the control-flow of the application is straight-forwardly transformed into a data-flow representation as the resulting streaming network shown in Figure 3 illustrates. The implementation for each box is in large parts provided by existing C code as S-Net provides interfaces for external programming languages (cf. [8] for a more detailed interface description) such that for existing code (e.g. written in C) only small wrapper functions need to be supplied.

![ACO algorithm modeled in S-Net](image)

Figure 3: ACO algorithm modeled in S-Net

Assuming that the individual ant decisions are independent (which is commonly true) our S-Net implementation follows the most intuitive approach to parallelise the ACO algorithm by expressing the constructSolution procedures (virtual ants) as concurrent components (an outlook on further ACO streaming variants is given in Section 5.2). This is achieved by using the indexed parallel replication operator using <ant_id> as identification tag. The actual number of parallel ants is determined by the range of <ant_id> tags emitted by the initialize component which in turn is connected to the remaining network via as serial combinator.

The box initialize reads input data, initialises the optimisation problem and sends a stream of ants according to <ant_id> to different instances of the constructSolution box. The streams carry records of various parameters, such as constants C, iteratively updated best results A and iteration related variables R.

After the parallel instances of constructSolution finish processing the results are accumulated to identify the current best solution. This is achieved by employing a merge construction consisting of a synchro-cell and an accumulator box: The synchro-cell keeps an accumulator record of type A and joins this up with a result emitted from one of the constructSolution instances. The pickBest box adds this result to the accumulator. Through the star combinator (indicated by two stars and an exit pattern in Fig. 3) a serial chain of synchro-cells and pickBest instances is established. The accumulator is sent down to the next stage of the chain where it is joined up with a result of one of the remaining constructSolution instances by a synchro-cell. Through the <num_ants> tag the pickBest box determines when the last result has been seen; in this case the merging process ends and a record is send on to the box update.

The box update has two choices: it can update the pheromone matrix and start a new iteration, or stop the stream and output answers if the stopping criterion is met, i.e. box update may end the conceptually infinite pipeline with producing a record containing the tag <done>.

5.2 Further Streaming Variants – An Outlook

The above-mentioned implementation in S-Net represents the probably most straight-forward idea of a streaming-oriented ACO algorithm. In principle, this implementation considers a set of m stationary ants processing a continuous stream of pheromone matrices (and problem instances) where each ant is assumed to construct and evaluate a complete solution per call to constructSolution. This is followed by a synchronisation before starting the comparison and update procedures. Further research in streaming-oriented ACO may respect the following variations: i) streaming objects – to consider the pheromone matrix as stationary unit which is traversed by a continuous stream of ants, ii) partitioning – to sub-divide pheromone matrices or to split ants into groups of consecutive ant decisions, iii) evaluation – to also compute fitness evaluations in parallel with solution construction, iv) update – to also execute pheromone updates in parallel with solution construction without prior synchronisation (which may lead to a non-generational update concept). For each variation, the fundamental instructions remain unchanged, to a large extend existing code can be re-used. The main challenge is to model the problem partitioning and the coordination of dataflow and synchronisation. These tasks shall be largely supported by the expressiveness of S-Net and through the de-coupling of concurrency and algorithm engineering.

6. PERFORMANCE EVALUATION

In this section, the setup and the results of the experimental performance evaluation of ACO for the Single Machine Total Weighted Tardiness Problem (SMTWTP) are presented. For a detailed introduction we refer to e.g. [4].

6.1 ACO Instantiation for SMTWTP

The algorithm described here instantiates the generic ACO algorithm (see Section 4) for the Single Machine Total Weighted Tardiness Problem, where n jobs (items) need to be scheduled on a single machine. Associated with each job j are its processing time p_j, weight w_j and due date d_j. The goal of SMTWTP is to find a job sequence π with a vector represented by a permutation of job numbers 1, ..., n which minimises the total weighted tardiness TW = ∑_{i=1}^{n} w_j \cdot T_j, where T_j = max{0, C_j - d_j} denotes the tardiness and C_j defines the completion time of job j = π(i). The complexity
of SMTWTWP was shown to be \( \mathcal{NP} \)-hard [14]. For large problem instances \((n > 50)\) exact algorithms often fail to calculate the optimum in acceptable computation time [1, 3]. Therefore alternate approaches try to find good, near-optimal, solutions by applying different heuristics, amongst them ACO algorithms belong to the best performing meta-heuristics [15]. For this paper, the Apparent Urgency (AU) heuristic [22] was chosen to derive domain-specific guidance \( \eta_{ij} := 1/au_j \). The AU-heuristics sorts the jobs in non-decreasing order of its apparent urgency \( au_j = (w_j/p_j) \cdot \exp(-\max\{d_j - C_j, 0\}/kp) \) with \( p \) expressing the average processing time of the unscheduled jobs and \( k \) a parameter chosen as suggested in [22]. AU exhibited a competitive performance in prior evaluations [4]. For the sake of brevity, additional local search routines are disregarded in this paper. During the update process pheromone increments are calculated as \( \Delta_{ij} = n/TW^* \) with \( TW^* \) denoting the total weighted tardiness of the best schedule \( \pi^* \).

### 6.2 Experimental Setup and Evaluation

The common starting point for the comparative study was a sequential ACO algorithm for SMTWTP written in C. This code was de-composed to derive a concurrent streaming-oriented variant using S-Net as outlined in Section 5. As counterpart for the experimental evaluation, the sequential C code was manually parallelised (without streaming) using PThreads such that each ant executed in a separate thread. Both algorithms perform the same calculations such that their optimisation behaviour is identical and the performance evaluation can be restricted to a pure runtime comparison. The runtime measurements presented in this section were collected on a 48-core computation server, comprising 4 sockets with 2 by 6 core AMD Opteron 6174 processors. The machine is equipped with 256GB of main memory and runs Linux kernel version 2.6.35. The experiments have been repeated three times with stable measurements (average deviation of runtime < 1%).

A first set of experiments has been conducted to assess the overall performance of the implementation. In Fig. 4 the absolute runtimes on several problem sizes are presented. Each of the graphs shows the recorded runtime of the S-Net and C/PThreads implementation. For small problem sizes where each solving step per ant only takes a fraction of a second the results clearly show the overheads that the S-Net implementation comes with. For larger problem sizes, and accordingly longer runtimes, these overheads are less significant and the performance is closer to that of the hand-parallelised code. These overheads stem mostly from the differences in memory allocation strategies; where the C code operates on global arrays that are allocated only once in the beginning, the S-Net implementation reallocates temporary arrays in each iteration. This is owed to the state-freeness of boxes that are not allowed to share state through global variables. In addition to this, the merging phase of the S-Net implementation that iteratively collects all sub-results in an accumulator incurs higher costs in comparison to the barrier synchronisation that is used in the C implementation.

A second class of experiments was run to further quantify the efficiency of the implementations. In these experiments each implementation was given 40 inputs. From the total runtime measures, the runtime per record was computed. Due to the streaming nature of S-Nets, the S-Net implementation
An early model of stream-processing have been Kahn Process Networks [13]. A Kahn Process Network consists of processes that read from one or more FIFO input channels and write to one or more FIFO output channels. A process gets fired (i.e., becomes ready) when data are available on all its input channels. Kahn Process Networks are deterministic in the sense that the same input data always produce the same output data. This is based on the assumption that each process behaves in a deterministic way, i.e., the input data are read in a deterministic order and the process blocks while there are no data available on the input. Though the original specification of Kahn Process Networks defines the capacity of any FIFO channel as infinite, i.e., a write to the channel cannot block. However, real implementations with bounded channel capacity introduce artificial deadlocks.

Stream processing has evolved into a complete programming paradigm, resulting into various approaches combining stream processing with software engineering methods. For example, stream processing has become quite popular for embedded computing with the arising of synchronous strictly time-triggered approaches, like Giotto [12], Scade [6], or StreamIt [25].

Reo is a coordination language for stream processing that deploys software engineering methods to structure streaming networks hierarchically [2]. Connectors are explicit entities in Reo. The atomic connector is called channel and exposes properties like capacity, lossy communication, or allowing only synchronous read and write. Connectors can be constructed as networks of channels and may be reconfigured at runtime. WaveScript is an example of a stream-based programming language that does not follow the concept of a coordination language, as stream-based communication and logic programming are interweaved [17].

S-Net positions itself as a stream-based coordination language that supports asynchronous and non-deterministic computation. Further, S-Net facilitates typed messages and provides language interfaces for different process implementation languages.

8. CONCLUSION

The paper focuses on a particular form of combinatorial technique called Ant Colony Optimisation, which is known to be useful for solving various graph-based problems of practical significance. Using an application of ACO as an example of a highly irregular, distributed problem, we have demonstrated that the data-flow-style, stream-processing component technology S-Net facilitates both the development and distribution/parallelisation of the code with high-level and easy to use language constructs, while keeping the performance in the same league as the hand-coded solutions utilised by industry.

The performance evaluation of the ACO implementation shows that the performance of S-Net lies within the same range as that of hand-parallelised C code. Still, as the maximal measured speed-up on a 48-core machine is slightly below 24 further investigations are required to identify ways to increase the efficiency of the S-Net implementation. A finer granularity of the decomposition of the application is expected to be essential in achieving a higher utilisation of the machine. Experiments with additional problem sizes, different scheduling techniques and dynamic load balancing will provide further insight and guidance towards find-
ing current inefficiencies. Also, a measurement framework that allows for capturing several properties such as execution time, waiting time, throughput and latency of single tasks and networks is currently under development. Even in its current prototypical state it already helps to pinpoint hot-spots in the application and the S-Net runtime system. Work in this area is expected to substantially contribute to future developments of the S-Net tool chain. Ultimately, we plan to use runtime observations to apply dynamic network optimisations on the coordination level as well as to provide the compilation process of the box language with collected information in order to generate more specialised and optimised code.

On the algorithmic side, future steps include producing a variety of stream processing schemes for ACO, coded in S-Net, to investigate their relative efficacies.

Acknowledgements
This work is partially supported by the European Union through the STREP project Advance (Asynchronous and Automatic Virtualisation through Performance Analysis to Support Concurrency Engineering, project no. FP7 248828).

9. REFERENCES
ABSTRACT
In this paper the ADVANCE approach for engineering concurrent software systems with well-balanced hardware efficiency is addressed 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-
developed 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.

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.

Figure 1: General image processing pipeline for quality inspection

Figure 2: Abstract blob analysis tool for statistical performance analysis
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 synchrocels, 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 characterized 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 \( \sigma_{xx}, \sigma_{yy}, \sigma_{xy} \) are calculated using the standard deviation. Finally, the compactness \( K \) in Equation 8 defines the roundness of a region, where \( K = 1 \) denotes a circle and \( K > 1 \) denotes a line. To calculate the compactness \( K, P \) and \( A \) have to be known.

\[
P = \sum_{(x,y) \in C} 1
\]

(1)
\begin{equation}
A = \sum_{(x,y) \in R} 1
\end{equation}
\begin{equation}
x_s = \frac{1}{A} \sum_{(x,y) \in R} x
\end{equation}
\begin{equation}
y_s = \frac{1}{A} \sum_{(x,y) \in R} y
\end{equation}
\begin{equation}
\sigma_{xx} = \frac{1}{A} \sum_{(x,y) \in R} (x-x_s)^2
\end{equation}
\begin{equation}
\sigma_{yy} = \frac{1}{A} \sum_{(x,y) \in R} (y-y_s)^2
\end{equation}
\begin{equation}
\sigma_{xy} = \frac{1}{A} \sum_{(x,y) \in R} (y-y_s) \cdot (x-x_s)
\end{equation}
\begin{equation}
K = \frac{\pi^2}{4 \cdot \pi \cdot A}
\end{equation}

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 combined 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 provides 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 implementatation. 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 usual 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 Eclipse Modeling Framework (EMF)\(^2\), this effectively allows the use of XTExT together with other EMF frameworks like for instance the Graphical Modeling Framework (GMF)\(^3\).

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\(^\text{TM}\) PCG-81112M with an Intel\(^\text{TM}\) Core\(^\text{TM}\) 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], SAG-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

\(^1\)XTex: http://www.eclipse.org/xtex/
\(^2\)EMF: http://www.eclipse.org/modeling/emf/
\(^3\)GMF: http://www.eclipse.org/modeling/gmp/
Source code of blob analysis tool using S-NET

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 in parallel to their dependencies to the results of other modules. Each of those boxes is independent and 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 listing).

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 output.

Source code of area computation S-NET box

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 C4SNetGetData must be used (line 4), in case the pointer to a CBlob. The area of this box 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) have been used, whereas the latter is imple-

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>
<thead>
<tr>
<th>Table 1: Throughput (T) of seq. exec. [recs/s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perimeter</td>
</tr>
<tr>
<td>Area</td>
</tr>
<tr>
<td>Compactness</td>
</tr>
<tr>
<td>Centroid</td>
</tr>
<tr>
<td>Moments</td>
</tr>
<tr>
<td>Total</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2: Latency (L) of seq. exec. [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perimeter</td>
</tr>
<tr>
<td>Area</td>
</tr>
<tr>
<td>Compactness</td>
</tr>
<tr>
<td>Centroid</td>
</tr>
<tr>
<td>Moments</td>
</tr>
<tr>
<td>Total</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 3: Jitter (J) of seq. exec. [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perimeter</td>
</tr>
<tr>
<td>Area</td>
</tr>
<tr>
<td>Compactness</td>
</tr>
<tr>
<td>Centroid</td>
</tr>
<tr>
<td>Moments</td>
</tr>
<tr>
<td>Total</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Figure:blobs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perimeter</td>
</tr>
<tr>
<td>Area</td>
</tr>
<tr>
<td>Compactness</td>
</tr>
<tr>
<td>Centroid</td>
</tr>
<tr>
<td>Moments</td>
</tr>
<tr>
<td>Total</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Figure:blobs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perimeter</td>
</tr>
<tr>
<td>Area</td>
</tr>
<tr>
<td>Compactness</td>
</tr>
<tr>
<td>Centroid</td>
</tr>
<tr>
<td>Moments</td>
</tr>
<tr>
<td>Total</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Figure:blobs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perimeter</td>
</tr>
<tr>
<td>Area</td>
</tr>
<tr>
<td>Compactness</td>
</tr>
<tr>
<td>Centroid</td>
</tr>
<tr>
<td>Moments</td>
</tr>
<tr>
<td>Total</td>
</tr>
</tbody>
</table>

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.

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 multicore CPUs or many-core GPUs. As exposed, this support is provided by the stream processing language S-Net. Furthermore, we introduced the Light-weight Parallel Execution Layer 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 runtime 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 Graphical Modeling Framework.
8. ACKNOWLEDGMENTS
The work has been funded by the EU FP7-project AD-VANCE.

9. REFERENCES
Monitoring Framework for Stream-processing Networks

Vu Thien Nga Nguyen
School of Computer Science
University of Hertfordshire
Hatfield, United Kingdom
v.t.nguyen@herts.ac.uk

Raimund Kirner
School of Computer Science
University of Hertfordshire
Hatfield, United Kingdom
r.kirner@herts.ac.uk

Frank Penczek
School of Computer Science
University of Hertfordshire
Hatfield, United Kingdom
f.penczek@herts.ac.uk

Abstract—In this paper we present a monitoring framework that exploits special characteristics of stream-processing networks in order to reason the performance. The novelty of the framework is to trace the non-deterministic execution which is reflected in i) the dynamic mapping and scheduling of network components at the operating system level and ii) the dynamic message routing across the network at runtime. We evaluate the efficiency with an implementation for the coordination language S-Net, showing negligible overhead in most cases.

I. INTRODUCTION

The ongoing trend towards increasing numbers of execution units running in parallel raises challenges for the field of software engineering. For example, mastering concurrency issues while still exploiting a high fraction of the potential computing power of a parallel platform challenges the programmer.

Stream-processing networks [10], [2] are networks of components that communicate via streams. They have been proposed as a paradigm to reduce the complexity of parallel programming, since the communication with streams also provides a form of implicit synchronisation at the same time. However, as for other parallel programming approaches, achieving high utilisation of the individual resources is still of high concern for stream-processing networks. Thus, the availability of monitoring frameworks that provide insights into the internal temporal behaviour is important.

There exist numerous monitoring frameworks for performance debugging, with the commercial ones typically covering generic parallel programming approaches. However, stream-processing networks do provide special properties like synchronisations through communication channels, for which dedicated monitoring support would be beneficial. Monitoring the temporal behaviour would also require to extract information about the operation of the underlying operating system, as the resource management of the operating system typically has a significant influence on the performance of the parallel program.

In this paper we present a software-implemented monitoring framework for performance monitoring of stream-processing networks that operates both at the runtime system level and also at the operating-system level. This allows to provide detailed information with low overhead. The monitoring framework focuses on the non-deterministic execution behaviour of the application at both levels. In contrast to other approaches for monitoring stream-processing applications, we do not need to instrument the application code. The approach also does not rely on any special automatic code instrumentation, as we embedded the required annotations into the application-independent side, namely the runtime system of the streaming layer and the operating system.

We present in Section II the assumptions about the kind of stream-processing networks for which we want to support monitoring. In Section III we list the requirements about what information has to be monitored and present the conceptional architecture of the proposed monitoring framework. Section IV describes the concrete stream-processing environment for which we implemented the monitoring framework. This is based on the coordination language S-Net [7], [8], running on the user-mode operating system LPEL [13]. In Section V we study the overhead of the monitoring framework. Section VI sketches how the monitoring information can be used for performance measurement and analysis. In Section VII we discuss related work and Section VIII concludes the paper.

II. ASSUMPTIONS & DEFINITIONS

A. Terminology

Scheduling vs. Mapping. The scheduling of resources involves decisions for the space and time domain. By convention, within this paper we refer to the space scheduling as mapping and to the time scheduling simply as scheduling.

B. Stream-processing Networks

A stream-processing network [10], [2] consists of a set of processing components connected by directed communication channels, called streams. We assume that each stream has a single reader and single writer. While a component is a static description, its instances to be executed at runtime are called tasks.

C. Execution Framework

A stream-processing network is compiled into a program, to be executed on an execution framework. In the following we describe the generic form of execution framework for which the monitoring framework should be applicable.
An execution framework consists of a runtime system and an operating system as in Figure 1. The runtime system maps the compiled stream-processing program into a set of tasks and streams, as shown in the upper part of Figure 1. Each task loosely represents for a network component. Tasks communicate with each other by sending and receiving messages via streams. A stream again has single reader task and single writer task.

The operating system provides the resource management, including the mapper to assign tasks to computational elements of the underlying platform and a scheduler to define the temporal order of task executions within the computational elements.

![Assumed Execution Framework](image)

Fig. 1. Assumed Execution Framework

The operation of the scheduler has to take into account the current waiting relation between tasks to decide which task to execute next.

The waiting relation of a task A waiting for a task B is denoted as a predicate $W(A, B)$. $W(A, B)$ is TRUE if task A is waiting for task B and FALSE otherwise. To express the predicate formally, we denote the unique reader task and writer task of a stream $S$ as $\text{reader}(S)$ and $\text{writer}(S)$, respectively. Further $\text{state}(S) \in \{\text{full, empty, available}\}$ describes the filling level of a stream $S$’s waiting queue, and $\text{state}(A) \in \{\text{blocked-by-input, blocked-by-output, ...}\}$ describes the state of a tasks $A$. Based on these definitions the predicate $W(A, B)$ can be formally described as

$$W(A, B) \iff \exists S \in \text{STREAMS}.$$  

$$\begin{align*}
( \text{state}(S) & = \text{empty} \land A = \text{reader}(S) \land B = \text{writer}(S) \\
& \lor \text{state}(A) = \text{blocked-by-input} ) \\
& \lor \\
( \text{state}(S) & = \text{full} \land A = \text{writer}(S) \land B = \text{reader}(S) \\
& \lor \text{state}(A) = \text{blocked-by-output} )
\end{align*}$$

### III. MONITORING FRAMEWORK

In the following we present our requirements for a monitoring framework and also the conceptual realisation of the monitoring framework based on the system assumptions in Section II.

The main purpose of the monitoring framework is to analyse the temporal behaviour of the stream-processing system. We are basically interested in two main use cases. First, we want to determine the performance of the stream-processing system in terms of throughput, latency and jitter.

**Latency**, is the delay between the receipt of data by a processing node and the release of the processing results into the output channel(s). For the high-performance system the average latency is important, while for real-time computing the maximum latency is significant.

**Jitter**, describes the variability of the latency. For high-performance computing the jitter can be a useful metrics to guide the dimensioning of internal implementation-specific mechanisms needed to store data. In real-time computing the jitter is also important for control applications to reason about the quality of control.

**Throughput**, how much data (either how many messages or data volume) is passed through a node per time unit.

In the absence of concurrency, the throughput is related to latency but not to jitter. In the presence of concurrency, throughput, latency and jitter are independent.

Second, we want to support performance analysis, i.e., to provide information about the detailed timing behaviour of the individual computational elements in order to tune the performance.

#### A. Monitoring Requirements

1) **Determining Performance Metrics:** While throughput of an application can be obtained without knowing the internal operations on input messages, latency calculation is performed for each input message and requires the full execution trace of the input message. The execution trace of an input message is formed from two kinds of information: i) the descendant messages generated during the execution of the input message; ii) the execution time of tasks on the input message itself and all its descendants. To provide the first kind of information, all messages should be distinguishable and the parent-child relations between messages should be provided. The second kind of information can be obtained through task events including message-read and message-written.

2) **Performance Reasoning:** The application performance are affected by the resource management which is controlled by mapping and scheduling. Therefore, it is necessary to monitor the mapping and scheduling activities to identify performance problems.

Although it is possible to have static mappings for static stream-processing networks, it is not the case for dynamic networks. To support general stream-processing networks, the monitoring framework must capture the mapping activities. Scheduling activities are controlled by the scheduling policy on the current task states. Basically, there are two types of task states counted on the scheduling activities: ready and blocked. Ready tasks are considered to be scheduled while blocked ones are not. As the task states are changing during the runtime, scheduling activities are dynamic. The monitoring framework therefore must capture these activities and task states. However, this is not enough to reason about the application performance since it does not provide the causes of blocked tasks. These causes are in fact reflected by the waiting relations between tasks. Thus, this information should also be included.
B. Concepts of the Monitoring Framework

In this section, we present concepts of the monitoring framework aiming to collect the information described in Section III-A. The required information includes: mapping activities, scheduling activities, waiting relations between tasks, and execution traces of input messages. The overview of our monitoring framework is shown in Figure 2 which consists of the Message Identification Generator (MIG), Stream Monitoring Object (STMO), Task Monitoring Object (TMO), Mapper Monitoring Object (MMO) and Scheduler Monitoring Object (SCMO).

To capture the mapping activities, the MMO is employed, using event driven techniques. For each event in which the mapper sends a task to computational element, the MMO records the identifications of the task and the computational element. Using the same technique, the SCMO is used to monitor scheduling activities by capturing scheduling events such as task-created events, task-blocked events, and task-destroyed events, etc. For each of these, SCMO records the time, the event type and the task identification.

As presented in Section II-C, waiting relations are inferred from the task states, stream states, stream reader and stream writer. The task states are actually supplied by the scheduling events. The rest is provided by a STMO which is equipped for each stream.

As discussed in Section III-A1, to obtain the execution traces of input messages, first it is necessary to distinguish messages. In the runtime system, a MIG is responsible for this by generating unique identification and attaching to each message. Second, the parent-child relations can be obtained easily by providing each message with its parents’ identifications. Finally, task events are captured by TMOs since the SCMO does not have control on the internal operations of tasks.

IV. IMPLEMENTATION OF THE MONITORING FRAMEWORK

In the following we present a concrete instantiation of the monitoring framework, based on the execution framework composing of the S-Net runtime system and the LPEL operating system.

A. Stream-processing with S-Net

S-Net [7], [8] is a declarative coordination language that aims to separate computations from concurrency management aspects. The computational logic is meant to be capsuled inside the individual computational components, also called boxes, while S-Net focuses on how to connect the communication between these components via streams.

These boxes of a network are reentrant programs (written in a conventional programming language) that transform a data element from a typed, single input stream into a (possibly empty) sequence of data records on a typed, single output stream without any persistent state, i.e., the value of none of the internal variables is preserved from one execution to the next.

In order to construct a streaming network from boxes, S-Net provides four network combinators. On the one side these are static combinators, called serial composition (denoted as . .) and parallel composition (denoted as |), to construct pipelines and branches. They are static in the sense that only one instance for each of their operands are created. On the other side there are dynamic combinators that create replicas of their operands on demand by means of serial replication and parallel replication. Serial replication (called ?-combinator in S-Net) allows to instantiate execution pipelines of dynamic lengths. Data records in this pipeline are processed and forwarded to the next stage until an exit condition is met. Parallel replication (called !-combinator) creates a dynamic number of instances of its operand and combines these in parallel. Data records are processed by one of these instance; the concrete instance is determined by a tag that a data record is expected to carry. Note that above combinators, except for the serial composition, do not preserve message order. But they have a special order-preserving variant as well (denoted as | | ., **, and ! !)

The routing decision in parallel combinator is non-deterministic if a message type matches equally well to both branches, i.e., it is left to the concrete implementation to decide the routing. Our monitoring framework is especially targeted to capture such non-deterministic behaviour.

Constructed networks are, just as boxes, SISO entities. Therefore, if a box or network requires data from several records as input, these records have to be synchronised, i.e. merged, first. S-Net provides a primitive for this, called synchro-cell. A synchro-cell is parameterised over the type of records that it is supposed to merge. As soon as it receives records of all matching types, it releases a single combination of these records.

B. LPEL - A User-mode Microkernel for the Coordination Language S-Net

The Light-Weight Parallel Execution Layer (LPEL) [13] is an execution layer designed for S-Net to give control on mapping and scheduling. LPEL adopts a user-level threading scheme providing the necessary threading and communication mechanisms in the user-space. It builds upon the services provided by the operating system or virtual hardware, such as
kernel-level threading, context switching in user-space, atomic
instructions and timestamping.

On LPEL, each S-Net runtime component is mapped onto
a user-level thread, called a task. Tasks communicate with
each other via streams. Each stream is a uni-directional
communication channel between two tasks and modelled as
a bounded FIFO buffer.

Tasks are distributed on workers, each of which represents
for a computation element. Task distribution happens upon
task creation according to the task allocation strategy. Each
worker manages its own set of assigned tasks and facilitates a
worker-local scheduling policy. The scheduling policy deter-
mines a task with a ready state to be dispatched next. The state
of a task changes according to the availability of the input and
output streams. Reading from an empty stream or writing to
a full stream causes the task to be blocked. Likewise, reading
from a full stream or writing to an empty stream can unblock
the task on the other side of the stream.

C. Implementation of the Monitoring Framework for LPEL
and S-Net

1) Mapping and Scheduling Activities: Since LPEL has its
own mapper, it allows to instrument the mapper to obtain
information. The MMO is implemented as a hook inside the
LPEL mapper and is activated when a task is assigned to a
worker. The MMO records the task identification, the task
name and the worker it is assigned for. The task identification
is a sequential number while the task name describes the
runtime position of the task in the network. This feature
helps to correlate tasks to the network components. Similarly,
SCMO is attached to the LPEL scheduler to catch scheduling
events. In LPEL, there are five scheduling events: task-created,
task-dispatched, task-blocked-by-input, task-blocked-by-output
and task-destroyed. For each of these events, the SCMO
records the event type, task identification and event timestamp.

2) Waiting Relations: Each task in S-Net is equipped with a
TMO to catch task events such as message-read and message-
written events. When each of these events happens, the TMO
records the time and the identification of the processed mes-
sage. Similarly, each stream is instrumented by an SMO to
memorise the reader task and writer task of the stream. The
stream state is used to determine the waiting relation but is
necessary only when either the reader task or the writer task
is blocked. The SMO therefore does not continuously look up
the stream state but only when the reader task or writer task
is blocked. For the performance measurement in the future,
the SMO also records the number of messages read from or
written to the stream during each period of task execution.

3) Message Execution Traces: The MIG is implemented
in the message manager of the S-Net runtime system. The
MIG generates message identifications, each of which has
two parts: the identification of the computational element
where the message is produced; and a message index within
the computational element. The first part is different for
each computational element. The second part is a sequential
and unique number for each message produced within a
computational element. The combination of these two parts
forms the uniqueness of message identifications. As presented
in Section III-B, the parent-child relations can be determined
by the presence of parents’ identifications in each message.
However, it is not necessary for LPEL and S-Net. An S-Net
component is compiled into one LPEL task whose execution
contains multiple execution of the S-Net component (as shown
in Figure 3). Since S-Net components do not have persistent
state, during the task execution, all the S-Net component
executions are independent and separated from each other.
This characteristics of S-Net and LPEL helps to determine the
parent-child relations between messages. Within a execution
of an S-Net component, all the input messages are the parents
of all the output messages. In the case where no input message
is required, output messages will have no parents. Messages
coming from external sources do not have parents, too.

4) Extra Information: For each worker’s execution, the
monitoring framework recorded sequential executions of tasks
allocated to it. Execution periods of each task are marked
between either task-created or task-dispatched and either task-
blocked-by-input or task-blocked-by-output or task-destroyed
events. Theoretically, unmarked time periods are the wait-
ing time of the worker. However, in practice these periods
might also include the time that the worker is busy with
non-functional tasks such as scheduling and timestamping.
To distinguish these kinds of time periods, each worker is
equipped with a Worker Monitoring Object (WMO) to capture
the worker events. There are four worker events: worker-
started, worker-waited, worker-resumed and worker-ended.
Based on these events, the waiting periods of a worker are
marked between worker-waited and worker-resumed events.
Non-functional tasks are therefore executed during unmarked
intervals, also called unaccounted time.

5) Information Storage: All the monitoring information is
written to files. Since all tasks in one worker have to be
executed sequentially, a worker has one log file shared by its
WMO and the TMOs of all its tasks. To reduce the overhead,
TMOs write only entries for three events: task-blocked-by-
input, task-blocked-by-output, and task-destroyed. The task-

Fig. 3. S-Net component executions in a LPEL task execution
created event is combined within the entry of the task-destroyed event. task-dispatched, message-read and message-written events are combined with either task-blocked-by-input or task-blocked-by-output or task-destroyed events depending on which one happens next. Similarly, the WMOs write the time for every worker event except for worker-waited events. Instead WMOs write the waiting time for each the worker-resumed event.

6) Operation Modes: For the flexibility, the implementation supports two modes: monitoring and low_overhead. In the former mode, all monitoring information are collected and printed to log files. In the later one, no information is collected and this allows the program to be executed with the lowest overhead.

V. EVALUATION OF MONITORING OVERHEAD

The monitoring framework instruments LPEL and the S-Net runtime system by placing some controls to collect monitoring information. This causes some overhead compared to the original LPEL even no information is collected. We do some experiments to measure the overhead in terms of the execution time and the size of log files. In our experiments, we measure the maximum overhead by using in the monitoring mode, and the minimum overhead with the low_overhead one. The minimum overhead is caused by monitoring controls without collecting any information while the maximum one includes the execution of monitoring controls, information collecting and information writing.

The overhead is caused by the WMOs, MMO and SCMO, MIG, TMOs, STMOs. Basically, the overhead of WMOs, SCMO and MMO are affected by the number of mapping and scheduling events while STMOs’ overhead correlates to the number of streams. MIG and TMOs deal with messages and therefore their overhead is affected by the amount of messages.

<table>
<thead>
<tr>
<th>Application</th>
<th>#MSE</th>
<th>#Message</th>
<th>#Stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANT</td>
<td>2.6 \cdot 10^6</td>
<td>1.5 \cdot 10^6</td>
<td>10 \cdot 10^3</td>
</tr>
<tr>
<td>DES</td>
<td>2.1 \cdot 10^6</td>
<td>470 \cdot 10^3</td>
<td>62</td>
</tr>
<tr>
<td>MC</td>
<td>-38 \cdot 10^6</td>
<td>19.3 \cdot 10^6</td>
<td>5 \cdot 10^6</td>
</tr>
<tr>
<td>RT</td>
<td>22 \cdot 10^3</td>
<td>23 \cdot 10^3</td>
<td>100</td>
</tr>
</tbody>
</table>

**TABLE I**

APPLICATION PROPERTIES

<table>
<thead>
<tr>
<th>Application</th>
<th>LPEL_mon0 time overhead [%]</th>
<th>LPEL_mon1 time overhead [%]</th>
<th>log size [MB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANT</td>
<td>0.05</td>
<td>3.45</td>
<td>132</td>
</tr>
<tr>
<td>DES</td>
<td>0.54</td>
<td>3.90</td>
<td>88</td>
</tr>
<tr>
<td>MC</td>
<td>-0.44</td>
<td>23.14</td>
<td>1800</td>
</tr>
<tr>
<td>RT</td>
<td>-1.35</td>
<td>-0.17</td>
<td>1</td>
</tr>
</tbody>
</table>

**TABLE II**

THE MONITORING OVERHEAD

All the applications are run on a 48-core AMD machine with 800 MHz, 512 MB cache for each core and 256 GB of memory. The time and space overhead is shown in Table II. The space overhead correlates to the number of monitored events while the time overhead does not. This can be explained by the fact that scheduling events and task events does not happen equally on workers, i.e. monitored events on one worker can be far more than on another. The overall time overhead is the maximum of the unequal overhead on each worker, and therefore does not scaled on the number of monitored events.

When the overhead is very low, there appears negative value in case the MC and RT applications. This timing anomaly can explained by the fact that with the performance increase of components the schedule can change, which then can reduce overall performance, similar to timing anomalies inside a processor [16].

The maximum time overhead does not correlate to the size of the log files because the file I/O is performed asynchronously supported by the hardware. For this reason, the maximum time overhead is quite small for relatively long running application. For the option pricing application, the overhead is quite large (23.14%) because the network is unfolded proportional to the input value which is very large in this case (10^9). While unfolding the network, the runtime system create numerous new tasks which correspond to a large number of monitored events. That increases significantly the amount of information needed to write to files. However, the execution time of the application is small compared to the amount of data and therefore it cannot take the advantage of the asynchronous I/O operations.
VI. APPLICATIONS OF THE MONITORING FRAMEWORK
A. Visualisation of Resource Utilisation

In this section, we present an approach to visualise the CPU utilisation of the application. Basically, the CPU utilisation on each computational element is obtained by extracting monitoring information from tasks mapped on the corresponding worker. We created a tool which reads the log files and generates an image to exhibit the resource utilisation of the application.

![Visualisation of CPU utilisation of the MC application on a 4-core machine](Fig. 4)

The CPU utilisation of the application is shown in two aspects: space and time (as in Figure 4). The space view is displayed horizontally: each worker’s execution are drawn in one column. The operations inside of a worker are drawn vertically scaling to the execution time. The worker’s operations are composed by multiple task execution periods. Each task execution period is determined by either a task-created or task-dispatched event and either the task-blocked-by-input or task-blocked-by-output or task-destroyed event following. Each task execution period is annotated with the name of the task, the execution time and the task state by the end of the execution (task-blocked-by-input, task-blocked-by-output or task-ended). Thanks to the task events, processed messages are also displayed during each task execution period. This way of displaying shows an overview of the resource utilisation of each worker execution, each task execution and each message processing.

During the worker’s execution, there is some unaccounted time in which the worker is performing non-functional tasks such as scheduling and timestamping. The unaccounted time is shown with the white colour while the waiting time is drawn with grey colour. For convenience, different kinds of tasks are also displayed with different colours.

B. Performance Metrics Calculation

This section shows the formulae to calculate the performance metrics in term of throughput, latency and jitter from the monitoring information.

**Throughput.** The throughput of an application equals to the number of input messages over the execution time of the application. The formula to calculate the throughput is shown in Equation 1, with $N_{\text{input message}}$ the number of input messages and $\text{MAX}_{\text{worker}}(\text{Execution Time})$ the maximum execution time of workers in the application. The execution time of a worker is measured from the worker-started event to the worker-ended event. The number of input messages can be provided by the user. In case of unspecified, the number of input messages is obtained by counting the number of messages which are not produced by any task.

$$ T = \frac{N_{\text{input message}}}{\text{MAX}_{\text{worker}}(\text{Execution Time})} \quad (1) $$

**Latency** is calculated for each input message $I$ as sum of tasks’ execution time on itself and its descendants $\text{Descendant}(I)$ (as shown in Equation 2). The descendants of an input message can be derived in the similar way of deriving parent messages (presented in Section IV-C).

$$ L(I) = \sum_{D_i \in \text{Descendant}(I)} \text{Execution Time}(D_i) \quad (2) $$

**Jitter** is easily obtained by calculating the standard deviation of the latency of all input messages $L$ as in Equation 3.

$$ J = \sigma(L) \quad (3) $$

C. Profile-based Performance Optimisation

An Optimisation Experiment.

![The Monte Carlo Option Pricing network with default mapping](Fig. 5)

We do a simple experiment to demonstrate the possibility to deduce performance problems from the monitoring information. The goal is to predict the mapping problem by the number of waiting events and total waiting time of each worker. In this experiment, the MC application (described in Section V) is run with 50 price paths on a 4-core machine in the monitoring mode. From the collected monitoring information, we have the number of waiting events (WE) and total of waiting time (WT) for each worker shown in Table III. The WE numbers are quite large in sense of only 50 price paths. Also, these
In this approach, the monitoring framework is used to collect the information about the non-deterministic behaviour of the application and the execution framework. This information then is analyse to detect performance problems. We are currently working an automatic tool to analyse the monitoring information and adapt the application to obtain better performance.

In this paper

VII. RELATED WORK

To deal with performance measurement and analysis in parallel programming, there is a significant number of prior work with three main approaches.

First, most of the work uses code instrumentation, for example Paradyn [11] and Pablo [14], to provide performance metrics focus at the application level, for example blocking time, message rates, I/O rates or number of active processors of the application.

The second approach is hardware instrumentation, which provides performance measurement in terms of hardware properties such as catch misses and TBL misses. Typical frameworks of this approach include VTune from Intel [9] and CodeAnaylist [1] from AMD.

The third approach is to operate on the operating system level, as also done in ours. One example of this approach is KernInst [15] which allows dynamic instrumentation in the kernel’s code space to measure performance metrics of a specified function. KernInst supports two metrics: the number of procedure calls made by a specified function and the number of kernel threads executed within the specified function. VTune does support operating system instrumentation focusing on the process events such as semaphores or mutexes. Nevertheless, to our best knowledge, none of these framework capture the mapping events, scheduling events and waiting relations which deeply affect the performance. Another factor that differentiates our work from others is that we focus on stream-processing in which the communication and synchronisation are reflected on streams.

The work on debugging for the stream language SPADE [6] focuses on stream processing and also uses instrumentations of the operating-system. This framework also provides stream-related performance metrics such as throughput and latency. Compared to ours, it also does not focus on the resource utilisation which has strong effects on the performance.

VIII. CONCLUSION

The support of monitoring is essential for achieving high system utilisation of parallel execution platforms. In this paper we presented a monitoring framework that is geared towards performance monitoring of stream-processing networks. This monitoring framework extracts information from both, the runtime system of the stream-processing network, for which we used the coordination language S-Net, as well as from the underlying operating system. The extracted information provides the trace of non-deterministic behaviours of the application at both levels. We have shown how the monitoring information can be used to determine performance metrics like
throughput, latency, or jitter and how to use it for performance analysis.

The monitoring approach is purely software-based, but avoids the need for instrumenting the application code. Though we experienced one measurement with a monitoring overhead of about 23%, the experimentation with several applications shows that the monitoring overhead is typically quite low, in the range of a few percent or less.

For the future work we plan to automate the performance optimisation approach presented in Section VI-C.

REFERENCES