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 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 components. Tasks communicates with each other by sending and receiving *messages* via *streams*. A stream again has single reader task and singe 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.

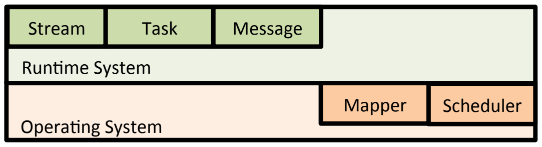

Fig. 1. Assumed Execution Framework

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

$$\begin{aligned} W(A,B) \Leftrightarrow\ & \exists S \in STREAMS. \\ & \quad (\ state(S) = \texttt{empty}\ \wedge\ A = reader(S)\ \wedge\ B = writer(S) \\ & \quad\ \ \wedge\ state(A) = \texttt{blocked-by-input}\ ) \\ & \quad \vee \\ & \quad (\ state(S) = \texttt{full} \wedge\ A = writer(S))\ \wedge\ B = reader(S) \\ & \quad\ \ \wedge\ state(A) = \texttt{blocked-by-output}\ ) \end{aligned}$$

III. Monitoring Framework

In the following we present our requirements for a monitoring framework and also the conceptional 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) Determing 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).

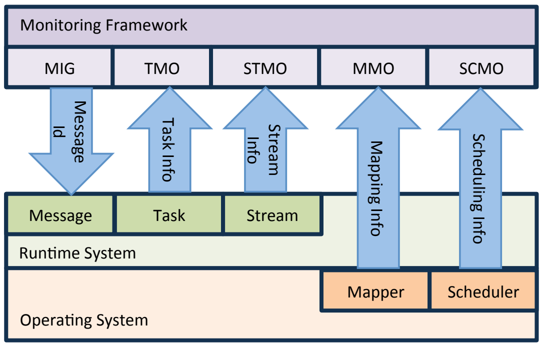

Fig. 2. Monitoring Framework

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 $\star$-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 `||`, $\star\star$, 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 determines 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 message. Similarly, each stream is instrumented by an SMO to memorises 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.

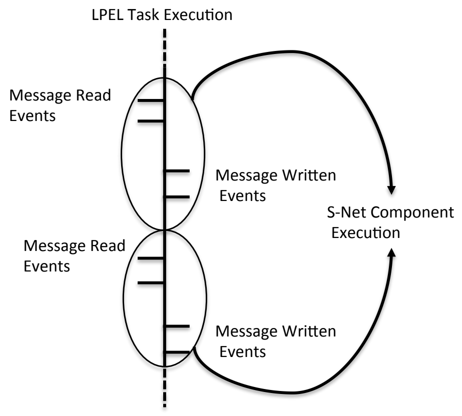

Fig. 3. S-Net component executions in a LPEL task execution

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 waiting 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-*

Application	*#MSE*	*#Message*	*#Stream*
ANT	$2.6 \cdot 10^6$	$1.15 \cdot 10^6$	$10 \cdot 10^3$
DES	$2.1 \cdot 10^6$	$470 \cdot 10^3$	62
MC	$38 \cdot 10^6$	$19.3 \cdot 10^6$	$5 \cdot 10^6$
RT	$22 \cdot 10^3$	$23 \cdot 10^3$	100

TABLE I
APPLICATION PROPERTIES

Application	LPEL$_{mon0}$	LPEL$_{mon1}$	
	time overhead [%]	time overhead [%]	log size [MB]
ANT	0.05	3.45	132
DES	0.54	3.90	88
MC	-0.44	23.14	1800
RT	-1.35	-0.17	1

LPEL$_{mon0}$ ...monitoring built-in, no output
LPEL$_{mon1}$ ...monitoring built-in, with monitoring output

TABLE II
THE MONITORING OVERHEAD

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.

Based on the default mapping and scheduling policies of LPEL [13], we do experiments on different applications with varied number of mapping and scheduling events (MSE), number of messages, and number of streams as in Table I. The experiment is performed on four applications as follows.

- **ANT** is a solver for combinatorial optimisation problems based on the behaviour of ants [5]. Several ants iteratively construct solutions to a given problem and leave a pheromone trail behind. Following ants use these trails as guide and base their decisions on it, refining previously found good solutions. Concurrency may be exploited in space, by means of parallel solver instances, and in time, by overlapping the execution of multiple stages of the pipeline. In this experiment, the application computes a schedule for 1000 jobs using 45 parallel ants that repeat the solver step 1000 times.
- **MC** is an application to calculate option prices using Monte Carlo method [3]. The experiment is performed with 1 million price paths.
- **DES** is an implements the DES cipher [4]. In the experiment, 500KB of data is encrypted.
- **RT** implements a distributed solver for ray-tracing (see [12] for detail). In the experiment, the 100 scenes are dynamically allocated to one of the four solvers with automatically balancing the load.

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 (10^6). 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.

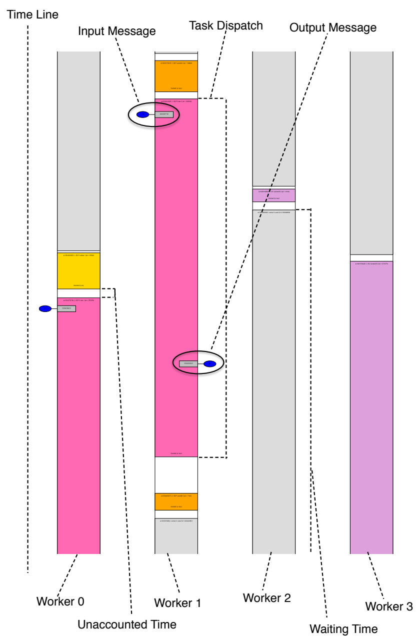

Fig. 4. Screenshot: Visualisation of CPU utilisation of the MC application on a 4-core machine

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 $N_{Input\_Message}$ is the number of input messages and $MAX_{worker}(Execution\_Time)$ is 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.

$$T = \frac{N_{Input\_Message}}{MAX_{worker}(Execution\_Time)} \tag{1}$$

Latency is calculated for each input message I as sum of tasks' execution time on itself and its descendants $Descendant(I)$ (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).

$$L(I) = \sum_{D_i \in Descendant(I)} Execution\_Time(D_i) \tag{2}$$

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

$$J = \sigma(\bar{L}) \tag{3}$$

C. *Profile-based Performance Optimisation*

An Optimisation Experiment.

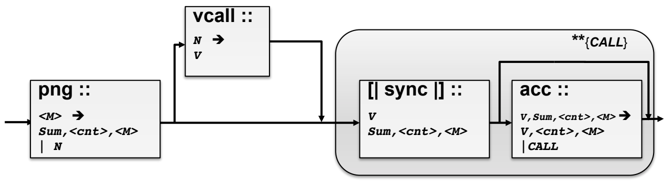

Fig. 5. The Monte Carlo Option Pricing network with default mapping

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

Worker	0	1	2	3
WE	45	128	105	152
WT [ms]	10.9	22.0	26.4	22.6

TABLE III
WORKER WAITING EVENTS AND WAITING TIME OF THE MC APPLICATION

numbers of worker 1, 2 and 3 are extremely high compared to those of work 0. Thus, it is likely that dependent tasks are spread widely on workers. The mapping log is used to confirm this prediction. From the original implementation shown in Figure 5, we see that all the tasks inside the serial replication are dependent on each other but are mapped in the manner of Round-Robin, i.e. spreading equally on 4 workers. Also, the log information shows that the execution time of all these tasks is quite small. That means the communication between workers is likely higher than the profit of the parallelism.

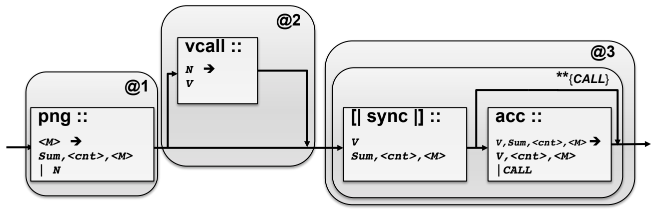

Fig. 6. The Monte Carlo Option Pricing network with explicit mapping

One way to improve the performance is to map all the tasks inside the serial replication to one worker as in Figure 6. Since LPEL supports manual mapping, we can measure to verify our reasoning. The result is shown in Figure 7. The first column is the execution time on original LPEL without monitoring framework. The second is the execution time with our monitoring framework and default mapping while the third shows the execution time with optimised mapping. The optimised mapping improves the execution time by 188%. Even compared with the original LPEL, the application is speed up to 183%.

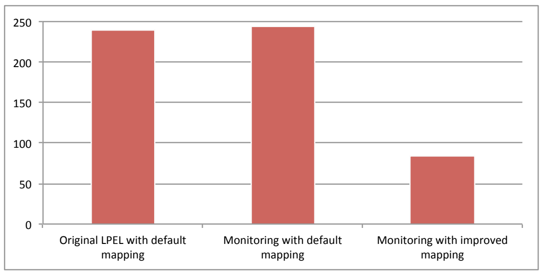

Fig. 7. Execution time (in seconds) of the MC application with different mappings

Profiled-based Optimisation Approach. With the improvement of 188%, the above experiment leads us to a promising profiled-based performance optimisation approach. 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.

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 CodeAnalysist [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

[1] AMD. AMD CodeAnalyst Performance Analyzer for Linux. http://developer.amd.com/tools/CodeAnalyst.

[2] E. A. Ashcroft and W. W. Wadge. Lucid, a nonprocedural language with iteration. *Commun. ACM*, 20:519–526, 1977.

[3] F. Black and M. Scholes. The Pricing of Options and Corporate Liabilities. *The Journal of Political Economy*, 81(3):637–654, 1973.

[4] DES. Data Encryption Standard. In *FIPS PUB 46-3, Federal Information Processing Standards Publication*, 1977.

[5] M. Dorigo and T. Stützle. *Ant Colony Optimization*. Bradford Book, 2004.

[6] B. Gedik, H. Andrade, A. Frenkiel, W. De Pauw, M. Pfeifer, P. Allen, N. Cohen, and K.-L. Wu. Tools and strategies for debugging distributed stream processing applications. *Softw. Pract. Exper.*, 39:1347–1376, November 2009.

[7] C. Grelck, S. Scholz, and A. Shafarenko. Asynchronous Stream Processing with S-Net. *International Journal of Parallel Programming*, 38(1):38–67, 2010.

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

[9] Intel. VTune Amplifier XE. http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/.

[10] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Proc. IFIP Congress on Information Processing*, Stockholm, Sweden, Aug. 1974.

[11] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. *Computer*, 28:37–46, November 1995.

[12] F. Penczek, S. Herhut, S.-B. Scholz, A. Shafarenko, J. Yang, C.-Y. Chen, N. Bagherzadeh, and C. Grelck. Message Driven Programming with S-Net: Methodology and Performance. *Parallel Processing Workshops, International Conference on*, 0:405–412, 2010.

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

[14] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera. Scalable performance analysis: The pablo performance analysis environment. In *Proc. the Scalable parallel libraries conference*, pages 104–113. IEEE Computer Society, 1993.

[15] A. Tamches and B. P. Miller. Using dynamic kernel instrumentation for kernel and application tuning. *Int. J. High Perform. Comput. Appl.*, 13:263–276, August 1999.

[16] I. Wenzel, R. Kirner, P. Puschner, and B. Rieder. Principles of timing anomalies in superscalar processors. In *Proc. 5th International Conference of Quality Software*, Melbourne, Australia, Sep. 2005.