FD-COMA 2013

2nd HiPEAC Workshop on Feedback-Directed Compiler Optimization for Multicore Architectures

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

8th International Conference on High-Performance and Embedded Architectures and Compilers

HiPEAC 2013

Berlin, Germany

January 21–23, 2013
Title image of Brandenburg Gate at dawn by Thomas Wolf, www.foto-tw.de.
Workshop Programme:

10:00 – 10:05  Welcome: Clemens Grelck

10:05 – 10:40  Key note: Marc Schrijver, Mike Creemers, Philips Healthcare: 
*Running Real-time and Best-effort Applications Concurrently on Common Off-the-shelf Hardware*

10:40 – 11:00  Workshop paper: Kenneth MacKenzie, Philip Holzenspies, Kevin Hammond, Raimund Kirner, Nga Nguyen Vu Thien, René te Boekhorst, Clemens Grelck, Raphael Poss and Merijn Verstraaten: 
*Statistical Performance Analysis of an Ant-Colony Optimisation Application in S-Net*

10:00 – 11:30  Coffee break

11:30 – 11:50  Workshop paper: Jaroslav Sykora and Sven-Bodo Scholz: 
*Towards Self-Adaptive Concurrent Software Guided by On-line Performance Modelling*

11:50 – 12:10  Workshop paper: Vu Thien Nga Nguyen and Raimund Kirner: 
*Influences on Throughput and Latency in Stream Programs*

12:10 – 12:30  Workshop paper: Merijn Verstraaten, Stefan Kok, Raphael Poss and Clemens Grelck: 
*Task Migration for S-Net/LPEL*

12:30 – 12:50  Workshop paper: Raimund Kirner, Vicent Sanz Marco, Michael Zolda and Frank Penczek: 
*Fault-tolerant Coordination of S-Net Stream-processing Networks*

12:50 – 13:00  Discussion and closing
Running real-time and best-effort applications concurrently on common-off-the-shelf hardware

The Philips Healthcare use case in the Advance project

Marc Schrijver
Philips Healthcare – BIU Interventional X-Ray
Best, The Netherlands
marc.schrijver@philips.com

Mike Creemers
Philips Healthcare – BIU Interventional X-Ray
Best, The Netherlands
mike.m.creemers@philips.com

Abstract
The Philips use case in the Advance project consists of running multiple (image processing) applications on a single PC. Each application has its own specific timing requirements, ranging from soft real-time to best-effort constraints. The challenge for the project is to devise a system that runs all applications concurrently, while balancing – at run-time – the constraints of each application.

Keywords Advance project; PC; best-effort; real-time.

1. Introduction

Figure 1 shows an X-ray system as used for interventional procedures like catheterization and balloon angioplasty. In a typical setup, the equipment is installed in two rooms as depicted in Figure 2: the exam room where the actual intervention takes place, and the control room where the procedure is monitored, patient administration is done, etc.

During interventional cases, X-ray fluoroscopy is used as an imaging technique to monitor and provide visual guidance for the diagnostic examination or therapeutic intervention. The resulting moving images are viewed immediately during the intervention on a monitor, while the clinical procedure is carried out. Hence, patient and medical staff is exposed to radiation during prolonged periods of time. To minimize such exposures, very low X-ray dose rates are used for the imaging process, which unfortunately result in considerable degradations of image quality through X-ray quantum noise. For the simultaneous delivery of low X-ray radiation dose while preserving the image quality, advanced de-noising and contrast enhancement techniques are used to maintain an acceptable Signal-to-Noise Ratio (see Figure 3). Because of the involved eye-hand coordination, this advanced image processing should be carried out in real-time with sufficiently low latency.

Next to such low-latency image processing, there are numerous image processing applications to analyse the image, and to support the physician in his procedure by e.g. high-lighting objects of interest, or measuring the dimensions of such objects. These applications do not have a strict low-latency requirement, but obviously have some timing constraints to make them useable by the end user.

2. The use case for the Advance project

In designing a high-end X-ray system with support for the image processing applications described above, a straightforward approach is to allocate dedicated resources to each of the (image processing) applications. In our case, we would have separate PC’s for the real-time low-latency image enhancement and for the image analysis applications, respectively.

However, in terms of total-costs-of-ownership, it is beneficial to integrate as many functions as possible on a single PC. And given the ever increasing performance of modern multi-core PC’s, such integration of functions will be feasible today or in the short future (from a raw performance point of view).

The specific use case consists of three different applications:
• The real-time low-latency image enhancement application;
• A best-effort application known as StentBoost;
• An arbitrary background application that consumes whatever computing resources are left.

The challenge is to assign as many resources to the background application as possible – at run-time – while still guaranteeing the performances of the first two applications. Knowing in advance that some of the applications will have dynamic resource requirements, we are after using the computing power when it comes available, even if it is available only for a limited time interval.

The IST research project ADVANCE (Asynchronous and Dynamic Virtualisation through performance ANalysis to support
Concurrency Engineering) addresses “the dynamic adaptation of concurrent applications based on statistical performance feedback” [1]. And as such, the Advance project develops methods to solve the aforementioned challenge.

2.1 The real-time application

The real-time application is characterized by:
- It is either active, or it is not. It is only active when the system acquires X-ray images; it is not active during the rest of the time (including the replaying of previously acquired images).
- If active, it receives a sequence of images at a regular rate of 15 or 30 images per second (with images of 1024 by 1024 pixels, with 16 bits per pixel). The result should be ready before the next image arrives at its input, and low jitter (i.e. variations in latency) is very important.
- If active, it requires a fixed amount of compute power that does not depend on the image data.
- If inactive, it does not consume any compute power.
- The switch between active and inactive (and vice versa) is a designated state change of the complete system, and can be anticipated very well. Switch times are not critical for this use case.

2.2 The best-effort application

The best-effort application, known as StentBoost, enhances the visibility of a so-called vascular stent (see Figure 4). Such a stent is a small mesh of wire that is inserted into a blocked vessel to keep it open after balloon angioplasty. Due to the small size of the struts of that mesh, such a stent is hardly visible in normal X-ray images.
The StentBoost application takes a sequence of stent images as input, and produces a single enhanced image as output (see Figure 5). It operates by finding the stent in each of the images (by feature tracking), and by exactly registering the stents in all the images. By then combining all the registered stent images, it is possible to annihilate the noise while enhancing the visibility of the stent itself.

The input to this application is the same X-ray image sequence as used by the real-time application. That is, the feature tracking starts as soon as the first image of the sequence comes in, and the aim is to keep up with the incoming image stream (just like the real-time application does). If it cannot, one or more input images are dropped (typically without affecting the quality of the output image as long as there are sufficient image left). The feature tracking process is not pre-emptive, in the sense that it will always finish the processing of an image once it started processing it.

Although the final output of StentBoost does not need to be ready within milliseconds, the result should be available within seconds after the last image of the X-ray image sequence has been acquired.

For the Advance project, we are especially interested in (the modelling of) the feature tracking part, as it is strongly data dependent. In some X-ray image sequences it will be rather easy to find the stent, whereas in others it will be much harder. This ‘complexity’ typically depends on the noise level, the existence of stent-like objects that can be mistaken as a stent, and the speed at which the stent moves across the image (e.g. due to the heartbeat of the patient).

The amount of compute power needed to track the stent can be predicted by predicting the size of the search area to consider: the compute power scales linearly with the size (total number of pixels). And the search area size can be predicted by knowing the stent locations in the previous images:

- If there is no previous location known, the search area equals the complete image;
- In other cases, the search area size is proportional to the speed at which the stent moves across the image during the sequence. This speed is determined by a moving average filter.

Note that this predictor of the computation complexity is just a predictor; others are very well perceivable, but beyond the scope of this project and use case.

2.3 The background application

The background application can be any application that would normally run without any user interaction. This includes system monitoring functions, automated applications that store images to a hospital archive (PACS), etc.

In view of the Advance experiments, we impose a number of constraints on this application:

- It can be started anytime; for simplicity we assume it always has input data available (if applicable);
- Its execution can be divided into ‘tasks’ that execute only for a small amount of time. In practice, we assume each task does not take more than 200ms (on a single core), a time equivalent to 3 to 4 image intervals of the real-time application.
- It does not act on the same X-ray image sequence as the real-time and best-effort applications do. It is independent of that image stream.

The deployment of this application (and the others) is depicted in Figure 6.

3. System Requirements

In view of the Advance project [1], a number of requirements specific to this use case are of interest:

Figure 5. Example input and output images of StentBoost and StentBoost Subtract.
Conformity to the reference architecture – The reference architecture entails some guidelines for the design of the resulting X-ray system. It prescribes the use of standard (common-off-the-shelf) PC’s where possible, Microsoft Windows as the operating system, etc. Specifically, this means that the hardware platform itself is far from real-time, and it will have only a limited number of physical compute cores (say up to 32 per PC).

Modelling of applications - To support high-end imaging applications at reasonable cost, we require that the image processing applications make efficient use of the available resources on a platform. For performance control of the applications, the tasks should be modelled and the required intrinsic complexity should be known.

Analysis of dynamic applications - Medical imaging applications tend to be less streaming oriented and increasingly perform analysis of the data and specific features contained within the data for further processing steps. The nature of analysis applications is more dynamic in its behaviour with respect to both computing and memory usage, as compared to streaming and regular image processing. Concluding the above, there is a need for modelling of the dynamic applications and concepts for integrating them in an overall system.

Architectural Mapping - Professional medical imaging platforms are currently being built with off-the-shelf homogeneous processor cores, as they offer high-performance computing and programming flexibility at a reasonable system cost. For future systems, we might be interested in investigating the mapping of a multitude of advanced medical imaging applications onto many-core processor platforms. Knowing in advance that some of the applications will have dynamic resource requirements we are after using the computing power when it comes available, even if it is available only for a limited time interval.

Cost-efficient solutions - Cost constraints may be imposed on the system. Hence, it may be very interesting to fit the total processing inside a predefined number of clusters of cores. This can be used to create scalability in platform costs, both upwards and downwards. If the amount of applications and their load is too high within a predefined system, there is no other way than to control the resource usage of applications. A resource-management system may be required to control the resource assignment to individual applications and their corresponding quality.

Resource management and Quality-of-Service (QoS) – We want to optimize the quality of individual applications, under the condition that a certain set of applications can still be executed concurrently and the overall system performance is optimal for the end user. There should be a trade-off between real-time image processing and ‘best-effort’ post-processing tasks.

4. Practical approach in Advance

During the project, some practical considerations were made.

4.1 Application emulation and implementation

An application simulator was created to mimic the challenging dynamic behaviour found in the original applications but in a simplified manner to ease implementation. This application simulator consists of two concurrent processes: a hard real-time image de-noising and contrast enhancement process that must run with sufficiently low latency, and a soft real-time process that tracks a moving region of interest.

Realistic image streams, obtained from test runs of the X-Ray fluoroscopy system, were used as an input to the system. In the X-Ray images, the position of the stent was marked manually. This was done in order to reduce the variability of the tracking process due to algorithmic details and for the sake of controlled performance evaluation. Variability of the feature-tracking task is thus solely dependent on the area of the region of interest.

The two pipelines have different timing characteristics. The algorithm used to mimic the real-time application consists of a straightforward application of a Gaussian blur. The time needed to run this algorithm on an image depends on the size of the image, the number of channels and the number of bits per channel. All these three factors are kept constant throughout the runtime of the application. As a result, the latency of the task enhancing the image's quality (disregarding task scheduling policies, task pre-emption, cache effects, bus arbitration, etc) is rather constant.

The feature tracking part of the application shows a greatly varying runtime per image. This variation is due to the stent’s location, which changes slightly between two subsequent images. At certain (generally unpredictable) moments the location of the object may change faster due to larger movements, i.e. changes in the position of the patient or changes in the focus of the X-Ray scanner.

The image processing operations are done using the OpenCV computer vision library [2]. Feature tracking is done by matching a template of the object against overlapped regions of the image, where the match score is calculated from the sum of the squared differences in pixel values. Template matching is essentially a convolution, which is implemented as multiplication of frequency-transformed images in the frequency domain. Image enhancement (noise reduction) is achieved by applying a $7 \times 7$ Gaussian convolution kernel (i.e. Gaussian blur).

4.2 Exemplary scheduling approach

To further detail the exact operation of the applications and their timing constraints, an exemplary scheduling scenario is defined. This scheduling scenario is by no means intended to be the outcome of the Advance project, but it fulfils the requirements put forth for this use case.

First of all, we make some assumptions, in order to simplify the problem:

- Only the case where both the real-time and best-effort applications are running is considered. The case where only the best-effort (and background) applications are running, is very similar to solve, and is therefore disregarded.
- The various applications are mapped to disjoint sets of cores, at all times.
The operating system is assumed not to take any resources.

Each ‘task’ is finished completely before the set of cores is assigned to a new task.

Now, define the number of cores available for the best-effort and background applications to be $P$. So, $P$ is the number of cores minus the number of *statically assigned* cores of the real-time application. The key aspect is that it is a fixed number (except when switching between active and inactive real-time system, which case is not considered here any further). So, the problem is reduced to mapping only two applications onto $P$ cores (the StentBoost and the background applications).

Also assume that none of the applications can be pre-empted or remapped to different cores during their execution, other than when they end their current ‘task’. For the real-time and best-effort applications, such a task is the processing of an incoming image. For the background application the task itself is undefined, but is known not to last longer than a certain amount of time, no matter what (see Section 2.3).

The next step is to predict the number of cores required to finish the feature tracking of the best effort application within the given timing constraint (see Section 2.2). This prediction may for example be derived from the estimated motion found in the sequence of images. Denote that estimation by $n$. So, for the pertaining image, the system should assign $n$ cores to the StentBoost application, leaving $m=P-n$ cores for the background application. Obviously this is too simplistic:

- If $n>P$, we can assign $P$ cores at best, knowing that it will take more time to track the feature than we have available before the next image arrives. So, we’ll have to accept that the next image will not be used for feature tracking (see Section 2.2).
- If the background application (which runs independently from the image sequence, see Section 2.3) still occupies more than $m$ cores, there are no $m$ cores available at that point in time (let alone $P$ cores). So, the mapping algorithm will have to do with what is available, and assign all available cores to the feature tracking application. And, as before, the next image will be skipped by the feature tracker. The system will recover from this situation as the background application is assumed to run for less than 200 ms (see Section 2.3) before it can be reallocated to another set of cores (potentially the empty set).

### 4.3 Potential issues

We foresee some problems in implementing certain behaviour, such as:

- X-ray image data fed to the feature tracking application should be discarded in cases where the tracking process is busy with the previous image. This has proven to be a requirement that is difficult to represent in S-Net, as it requires extra-functional properties (i.e., latency) of the computation to be taken on board the coordination layer.
- An S-Net has no notion of time. This makes it difficult to implement a periodic generation of images at a fixed frame rate. To solve this, we had to create an external process that sends in clock records at a fixed interval.
- The predictor for the search area size of the feature tracker does not predict based on the input data, but based on the previous output data. So, another feedback loop (with its own state) is needed.
- We need a way of knowing that an application is still working on an image (or whatever other task), and thereby is holding on to one or more cores.

### Acknowledgments

The authors would like to thank the Advance project team and in particular Robert de Groote of the University of Twente for their help in defining and developing this use case.

### References


Statistical Performance Analysis of an Ant-Colony Optimisation Application in S-NET

Kenneth MacKenzie
Philip K.F. Hölzenspies
Kevin Hammond
School of Computer Science
University of St Andrews, UK
kwxm@inf.ed.ac.uk,
{pkfh, kh}@st-andrews.ac.uk

Raimund Kirner
Nguyen Vu Thien Nga
Rene te Boekhorst
University of Hertfordshire
School of Computer Science
Hatfield, UK
{r.kirner,v.t.nguyen,
r.teboekhorst}@herts.ac.uk

Clemens Grelick
Raphael Poss
Merijn Verstraaten
University of Amsterdam
Institute of Informatics
Amsterdam, Netherlands
{c.grelick,r.poss,
m.e.verstraaten}@uva.nl

Abstract

We consider an ant-colony optimisation problem implemented on a multicore system as a collection of asynchronous stream-processing components under the control of the S-NET coordination language. Statistical analysis and visualisation techniques are used to study the behaviour of the application, and this enables us to discover and correct problems in both the application program and the run-time system underlying S-NET.

1. S-NET: Language and run-time system

S-NET is an asynchronous stream coordination language [5, 6]. It combines so-called boxes, which are stateless computational kernels written in any programming language, into networks that transform a single input stream into a single output stream. A stream is a potentially infinite sequence of non-overlapping, discrete data items, called records (or messages). Records are collections of named fields containing values in the box language, together with tags which contain integer values. Values from the box language are opaque to S-NET, but integer tags are visible to both the box language and S-NET. The type of a record is the set of all the names and tag-values therein.

Disjoint paths can be constructed by using combinators such as parallel composition, which splits the input stream into a number of streams that are fed to the operands of the parallel composition. The transformed output streams are merged to make the resulting network Single In, Single Out (SISO) again. Which record is fed to which operand is determined by the record’s type. Boxes have specified input and output types, i.e. sets of names and tag-values that are expected to be in the relevant records. Where a stream is split, records are routed to that path which has the strongest matching input type, i.e. all names and tag-values in the input type are in the record and there is no path with an input type with more names and tag-values that are all in the record.

Statefulness is introduced by synchrocells. A synchrocell is also a SISO stream transformer, defined by a list of record types. For each record type, there is a corresponding (initially free) ‘slot’ in the synchrocell. In every slot, one record of that type can be stored, at which point the slot is filled. When all slots are filled, the synchrocell syncs, i.e. the records stored in all the slots are combined into a single record. This combined record is produced on the output of the synchrocell. Every slot of a synchrocell can only be filled once, so after a sync, the synchrocell ‘dies’.

Finally, S-NET has combinators for feedback (where the output of a network is fed back to its input stream, if it does not match a specified type) and recursion (where the output of a network is fed to a new instance of that same network, if it does not match a specified type). The latter is used in the application discussed in this paper. It is referred to as the star-combinator. The operands of the star-combinator are often informally referred to as ‘starred networks’. For an extensive treatise we refer to [7].

S-NET programs are compiled into binary and executed by the S-NET runtime system [4], which in turn uses the Light-weight Parallel Execution Layer (LPEL, [10]) for scheduling, placement and low-level thread management. User boxes as well as components that implement S-NET’s coordination are all instantiated as tasks in LPEL. This instantiation is ad-hoc, in the sense that tasks are created for boxes when the network in which they occur is first reached by a record. When the run-time system determines that a box can no longer be reached by any more records, the corresponding tasks are automatically garbage collected [3].

An LPEL task is input-buffered, i.e. every task has a (bounded) input FIFO buffer into which other tasks can write and from which only the owner-task can read. A task is enabled when there are items in its input buffer, and it is blocked when trying to write output to a full receiving buffer. LPEL creates a worker for every available processor core (or a user-specified number of cores to be used). A worker is assigned tasks, and the worker’s enabled tasks are executed in a round-robin fashion. A running task is never preempted; only when a task finishes or when it blocks on trying either to read from an empty input buffer or to write to a full receiving buffer, can a worker execute a different task.

Each worker has its own scheduler, and these cooperate to select tasks which are ready for execution and to execute them on the relevant worker. In the current implementation, all system tasks (synchrocells and various system boxes performing administrative tasks such as merging records) are allocated over cores in a round-robin fashion. There are also separate round-robin schedules for the
and eventually produce a hopefully close-to-optimal solution to the problem.

3. Statistical Analysis and Visualisation

3.1 Structure of the data

We performed a number of runs of the ant-colony optimisation application on a 48-core server with 4 sockets each having 2 by 6 core AMD Opteron 6174 processors. We have log data for all combinations of the following parameters:

- Number of ants: 1, 10, 20, 30, 40, 50, 60
- Number of cores used for execution: 4, 8, 16, 32, 48
- Size of input problem: 200, 400, 600, 800
- Number of iterations (max_it): 100, 500, 1000, 2000

This gives a total of $7 \times 5 \times 4 \times 4 = 560$ datasets. In this section we will concentrate on the data with 48 cores, 100 iterations and input size 400, with the number of ants varying. We report on the data for a single run, but repeated runs with the same configuration give very similar results.

Each dataset consists of a directory containing a map file describing the assignment of tasks to workers (i.e., cores), together with one log file for each worker, listing the events that have occurred on that worker. For the 1-ant example, the data totals 448 kilobytes: the map file contains 1408 lines and the log files contain approximately 180 lines each. For the 60-ant example, the data totals 23 megabytes: the map file contains 36808 lines and the log files contain approximately 10500 lines each.

For analysis and visualisation, we use the R system [11]; the plots in this paper were produced using the R library ggplot2 [12]. The log files contain a great deal of data and can be very large, so we use a parser written in Haskell to extract information of interest and output it in a tabular form suitable for input to R.

3.2 Analysis

We are primarily interested in the statistical properties of the application with a view to predicting execution time based on input. However, the data turned out to have some puzzling features which led to the discovery of problematic issues with respect to both the S-Net/LPSEL system and the implementation of the ant-colony application.

Given the structure of the application, one would expect the solve boxes (i.e., the ants) to have the largest latency. Internally, each ant contains a loop with 100 iterations, whereas the other user boxes contain mostly straight-line code. Moreover, there should be little variation in the latencies of ants. Each ant is executing identical code; there is a stochastic component (each ant occasionally attempts to improve its current solution by performing a slight random perturbation) but this should average out over a single execution.

However, this is very definitely not what happens. Figure 2 shows a graph of box latencies plotted against start time for 40 ants (“latency” here refers to the time from first input to final output). Points are coloured according to the type of box whose latency is being plotted: in this case, green points denote ants (solve boxes).

As expected, the execution time of ants is much greater than that of other boxes (with the exception of synchrocells, denoted by red points: however, these spend most of their lifetime waiting for input and perform very little computation). Unexpectedly, however, there are wide variations in the latencies of boxes. Moreover, latencies are not smoothly distributed; instead, boxes with similar start times tend to occur in small clumps with similar latencies.

What happens if we reduce the number of ants? Figure 3 shows a similar plot of tasks for the 30-ant example. Here we see that...
Figure 2: Task execution times for 40 ants

Figure 3: Task execution times for 30 ants

again latencies have a large variance; however, in this case there is a very striking periodicity evident in the variation of latencies.

For 20 ants we obtain Figure 4. In this case the latencies are much more evenly distributed and the clumping effect is less evident. Note also that the vertical scale changes, and that the latency of ants increases as the number of ants executing in parallel grows.

Table 5 shows the mean latency and the variance (taken over the entire execution of the program) for varying numbers of ants. The mean latency for 50 and 60 ants is very large, but this is to be expected since the number of ants exceeds the number of available cores (48), so some ants will have to wait for a previous ant of the same generation to finish before they can start. What is surprising is that even with 40 ants the latency is significantly higher than one would expect. The latency for the 1-ant case (15.5ms on average) can be regarded as the “true” execution time of an ant. Since the ants are supposed to be operating independently in parallel, one would expect the mean latency in the 40-ant case to be similar (perhaps with a little overhead), but instead it is more than 68 times greater.

How can this be explained? We were able to make some progress by looking more closely at execution times for ants in a single generation. In Figures 6 and 7 we display a close-up view of the plot of latencies against start times for generations 22 and 23 of the 40-ant data (the horizontal and vertical scales are the same for both plots). We see that these fall into two distinct classes. In the first class, which we will call Type I generations, all the ant executions start at approximately the same time and have a high latency. In the second class (Type II generations), the ant tasks for a
Figure 4: Task execution times for 20 ants

<table>
<thead>
<tr>
<th>Ants per generation</th>
<th>Total number of ants</th>
<th>Mean latency (ms)</th>
<th>Standard deviation</th>
<th>Total execution time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>15.5</td>
<td>7.95</td>
<td>2.56</td>
</tr>
<tr>
<td>10</td>
<td>1000</td>
<td>126.92</td>
<td>18.20</td>
<td>18.73</td>
</tr>
<tr>
<td>20</td>
<td>2000</td>
<td>314.06</td>
<td>38.93</td>
<td>44.30</td>
</tr>
<tr>
<td>30</td>
<td>3000</td>
<td>684.21</td>
<td>230.75</td>
<td>94.15</td>
</tr>
<tr>
<td>40</td>
<td>4000</td>
<td>1059.77</td>
<td>552.81</td>
<td>154.70</td>
</tr>
<tr>
<td>50</td>
<td>5000</td>
<td>1458.28</td>
<td>862.37</td>
<td>225.82</td>
</tr>
<tr>
<td>60</td>
<td>6000</td>
<td>1441.133</td>
<td>849.78</td>
<td>315.85</td>
</tr>
</tbody>
</table>

Figure 5: Latency statistics for varying amounts of parallelism

Figure 6: Ant execution times for generation 22

generation are split into two batches: each ant in the first batch has a high latency, and each ant in the second batch begins much later, but has a lower latency. Furthermore, the ants in Type II generations all have lower latencies than the ones in Type I generations.

Examination of all 100 generations shows that similar patterns are repeated throughout, with Type II generations forming about one third of the total. Moreover, there is an approximate periodicity: the generations typically (but not always) form groups of three, with one Type II generation followed by two Type I generations.

A similar dichotomy appears in the 30-ant case. The first few generations are of Type II, but then there is a long period where the majority of generations are of Type I; about halfway through, the generations revert to Type II and then the pattern repeats (cf Figure 3). In the 20-ant case, the majority of generations are Type I, with only about 7 out of 100 being Type II. For 10 ants the dichotomy disappears: all generations are of Type I.

3.3 Message analysis

The phenomena described in the previous section are still difficult to explain, but a third type of plot is very helpful. Recall that data is transferred between boxes by means of records (also called messages). The plots in Figures 8 and 9 show the messages emitted by boxes and the executions which they trigger.

In these figures, each horizontal line represents one message and consists of two contiguous sections (some of the lines are very short, and may be difficult to make out). The left-hand section
represents the transmission time of a message $M$, from output to input: the colour corresponds to the type of task from which $M$ has been output. The right-hand section represents the lifetime of the task $T$ whose execution is triggered by $M$, from the time when $T$ inputs $M$ the time when $T$ produces its final output: the colour denotes what type of task $T$ is.

Figure 8 shows the messages for generation 22 (which is Type I), and corresponds to the plot in Figure 6. The lower section of the graph shows messages being output by the update task and received by the solve tasks. The string-like upper section shows the sequence of ⟨sync⟩–pick_best executions forming the fold operation which chooses the best result from the current execution, and the middle section shows the output messages of the solve tasks travelling to synchrocells. There is some waiting time here because the synchrocell may not be created until preceding entries have been processed in the fold.

Returning to the bottom section of Figure 8, we see that the start times of the ant tasks are slightly staggered: this is due to the ants waiting for their input records, which are output sequentially by the update task. Apart from this though, the ants all run concurrently (confirming Figure 6) and finish at roughly the same time.

On the other hand, Figure 9 (corresponding to Figure 7) depicts generation 23, of Type II. We again see the two distinct batches of ants which appeared in Figure 7, but now it is clear that none of the ants in the second batch start to execute until after most of the ants in the first batch have finished.

This suggests that in Type II generations some of the solve tasks are being blocked, and indeed this turns out to be the case. Recall
that the update task outputs 41 records, and that each of these forms the input to a different subsequent task: 40 of them go to new solve tasks (in numerical order), and the final one is supplied to a (sync) as the seed for the fold. This routing is performed by a system task called a splitter, which runs on a worker of its own. Close examination of the log files confirms that generations become split into two different batches when the splitter is scheduled on the same worker as one of the solve tasks. The splitter sends a number of inputs to new solve tasks and can then become blocked because some solve tasks have not yet been scheduled (or because the update task has not yet produced all of its outputs), and at this point the solve task on the splitter’s worker can start to execute if its input has been emitted by the splitter. If this happens then the splitter is unable to emit its remaining outputs until the solve task has completed, and thus the remaining solve tasks are in turn blocked. This behaviour is a consequence of the scheduling strategy described at the end of §1, which purposely allows a system task to be scheduled on the same worker as a user task.

3.4 Periodic phenomena

A closer examination of the placement also explains the very distinct cyclic behaviour seen in the 30-ant example (Figure 3) and the less evident period-3 cycles in the 40-ant example.

In the 30-ant example, in generation $n$ (counting from 1), the ants are scheduled on successive cores (counting from 0) starting at $30n + 20 \pmod{48}$. The splitter is situated on core $31n + 20 \pmod{48}$, so it follows that the splitter is $n$ cores after the first ant (modulo 48). Thus at the start of the execution the splitter is on the same core as a low-numbered ant, leading to a high probability of the splitter being blocked, with a consequent delay to later ants. Thus initially most generations will be Type II, with all the ants having relatively low latency. As the execution progresses, the splitter moves on to clash with higher-numbered ants (or to be situated on cores without ants), leading to (a) a lower probability of blockage, and (b) fewer ants becoming blocked. Thus later generations will mostly be Type I, with all ants having high latency. On the 48th generation the splitter returns to the same core as the first ant, and the pattern repeats. This explains the cyclic nature of Figure 3.

In the 40-ant case, the first ant of generation $n$ is situated on core $40n + 10$ and the splitter is situated on core $23n + 28$ (both modulo 48). Thus the relative position of the splitter is $23n + 28 - (40n + 10) = -17n + 18 \equiv 31n + 18 \pmod{48}$. The first few elements of this sequence are 1, 32, 15, 46, 29, 12, 43, 26, 9, 40, 23, 6, 37, 20, 3, 34, 17, 0, … We have bolded “low” positions (arbitrarily chosen as those less than 20) which are likely to lead to Type II generations, and we see that these form a sequence which is roughly periodic with period 3; this reflects the structure which we saw earlier for the Type I and II generations in the 40-ant case.

We see that the very neat situation in the 30-ant case is essentially due to a numerical coincidence involving the operation of the scheduling algorithm, and that things are much more irregular for 40 ants. These features are very dependent on the details of the application. For example, in the 30-ant case the splitter is situated on core $31n + 20$; the number 31 occurs because there happen to be $126 = 2 \times 48 + 30$ other system tasks scheduled between successive (split) tasks. A small change in the S-NET network could remove some of these system tasks or insert new ones, leading to a significant change in the way the splitter moves about the cores and thus to a corresponding change in the latency of the ants. In particular, if the splitter was always situated on the same core as the first ant then the splitter would become blocked when the first ant started, and we would always see (short, fast) Type II generations; on the other hand if the splitter was just one core before the ants then it would never become blocked and we would always have (long, slow) Type I generations. In more complex applications the structure of the S-NET network can evolve under program control since the number of serial and parallel replications can depend on tag values in the output of user boxes. This could lead to sudden and completely unpredictable scheduling clashes, with corresponding drastic changes in network throughput. This strongly suggests that we should strive to eliminate such behaviour; possible solutions will be discussed later in the paper.

3.5 Variation of ant latency

We have now managed to explain why the executions of the ants in a single generation can become split into two batches, but we have not yet explained the observation that average latency increases with the number of concurrent ants. This is evident in Figure 8, where the strip representing ant executions is considerably longer than in Figure 9. We can get a better idea of the dependence by partitioning the ant executions into groups which are executing concurrently and then plotting the average execution time of the

Figure 10: Mean latency of ant groups

Figure 11: Logarithmic mean latency of ant groups
ants in the group against the size of the group. This is shown in Figure 10 for the 40-ant data.

We see that mean execution time increases rapidly with the number of concurrent ants, and the logarithmic plot in Figure 11 is close to a straight line, showing that the growth is approximately exponential.

This suggests that there is some contention between concurrent ants, and examination of the source code for the boxes confirms that this is the case. One source of contention is that at the start of execution, each solve box allocates some memory for the storage of temporary data structures: this is done using the libc library function malloc, which (in some versions of libc at least) performs a locking operation to preserve the integrity of the heap during memory allocation. However, there is another problem which is more serious. We mentioned earlier that the ant-colony optimisation method contains a non-deterministic step. In the implementation which we have been studying this is achieved by calls to libc’s rand function:

```
for (k = 0; k < num_jobs-1; k++){
    ...  
    q = ((double)rand())/RAND_MAX;  
    if (q < const_q0) {  
        ...  
    }  
    else{  
        q = ((double)rand())/RAND_MAX;  
        ...  
    }  
    ...  
}
```

Here \(\text{const}\_q0\) is defined to be 0.9, and \(\text{num}\_jobs\) is the number of jobs in the input data. In the present case, \(\text{num}\_jobs\) is equal to 400, and it follows that the execution of each solve box calls rand about 440 times. However, rand is not a pure function: it contains some internal state which is preserved by a global mutex. When we have 40 ants running concurrently with each ant making over 400 calls to rand, this leads to a significant amount of contention, nullifying much of the supposed advantage of parallelism. This also explains why average ant latency increases as the number of ants increases: the more ants we have, the more contention, and hence the longer the ants take to finish. The use of rand is quite a serious error in the code (and indeed is as a violation of the S-NET “contract” which should be satisfied by box code in order to obtain valid S-NET applications), but is very easily overlooked.

### 3.6 Summary

Thus we have a putative explanation for the complex statistical behaviour of the ant-colony application. We have two interacting bugs, one relating to the behaviour of splitters in the LPEL implementation, and the other to the use of calls to library functions which perform locking operations. The first bug leads to ant generations sometimes becoming split into two subgroups which execute consecutively, and the second leads to an increase of ant latencies as the size of a group of concurrently-executing ants increases. As we have seen, this increase is exponential (Figure 10), which leads to a significant decrease in performance.

### 3.7 Amelioration

How can we overcome these problems? In one way or another we must re-engineer some aspects of the S-NET/LPEL runtime system. One approach could be to fuse splitters with the user boxes whose output they are distributing. Another tactic would be to give splitters a high priority, allowing them to preempt box tasks. This would also require introducing preemption into the so far collaborative task management layer of LPEL. A third approach could be to alter the scheduling algorithm so as to place splitters (and perhaps other system tasks) on a subset of cores which is disjoint from the ones running user boxes. We are in the process of experimenting with these approaches, but at the time of writing performance data for the ant-colony example was not quite available; however, we are hopeful that this will solve our problem.

The bugs in the actual implementation of the ant-colony example are more easily dealt with. We have modified the ant code by replacing malloc-allocated heap memory with stack-allocated arrays, and by replacing the calls to the libc rand function with calls to a random number generator which maintains its state locally. We have some data for this version of the application, and the performance has improved markedly: for example, the total execution time for the 40-ant case drops from 154.7 seconds to 68.9 seconds. However, it appears that there is still some contention.

Figure 12 plots average latency per group versus group size for the modified program with 40 ants, and corresponds to Figure 10 for the unmodified program (but note that the vertical scale is different). We see that the exponential behaviour shown by the original program has been replaced by linear behaviour, and that the average latency has decreased considerably. However, ideally we would expect that ant execution time should be independent of the number of ants executing concurrently, and should be close to the 15.5ms average seen in the case of a single ant. Here the latency of an ant can be of the order of 400ms, still 25 times larger than we would expect. We have as yet been unable to explain this. We believe that all calls to stateful library functions have now been eliminated from the solve boxes, and thus we should have removed any possibility of contention; however the graph indicates that contention remains. A possibility is that there is some contention in the logging system as individual cores write data to files (and in particular to the map file which records information for all cores). We intend to try some experiments to see if this might possibly be the case, for example by turning off logging for some subset of the cores and seeing if behaviour of the overall system improves.

### 4. Conclusion

We have used statistical and visualisation techniques to investigate the behaviour of a complex multicore application. The LPEL logging system produces a great deal of output, and it is very difficult to interpret the raw data; visualisation has helped us to gain a much better understanding of the behaviour of our application and to dis-
cover (and make progress towards correcting) not only bugs in the program we have been analysing, but also previously-unnoticed issues in the S-NET/LPEL platform itself. Thus we believe that techniques such as the ones discussed here can be helpful both to application programmers and to system developers.

Our initial motivation for these investigations was to gather statistical data relating to latency and throughput of S-NET applications with a view to providing guarantees to end-users that applications will perform within specified parameters. This has proven to be more difficult than anticipated due to the complex nature of the performance data, but we hope that our investigations will now lead to simplifications in the system which will in turn lead to better programs with more consistent behaviour, enabling us to meet our original goals.

In the short term, we wish to fully resolve the issues discussed earlier in this paper; in the longer term, we will investigate a number of other S-NET applications, and we hope that the methods developed and the experience gained in the research discussed above will enable us to make rapid progress in our analysis, and also help us to recognise and correct errors and inefficiencies in the applications which we study.

Acknowledgements

The work has been funded by the EU FP-7 project ADVANCE (Asynchronous and Dynamic Virtualisation through performance ANalysis to support Concurrency Engineering, project no. 248828).

References


Towards Self-Adaptive Concurrent Software
Guided by On-line Performance Modelling

The ADVANCE Approach

Jaroslav Sykora
Heriot-Watt University, Edinburgh, United Kingdom
J.Sykora@hw.ac.uk

Sven-Bodo Scholz
Heriot-Watt University, Edinburgh, United Kingdom
S.Scholz@hw.ac.uk

Abstract

We present preliminary results on dynamic self-adaptation of streaming networks of concurrent programs (called boxes) to the underlying hardware platform. The monitoring subsystem gathers on-line measurements on the performance of individual boxes and uses the measurements to construct (and improve) performance models of the boxes. Ideally, the models accurately predict the running time of boxes in their next invocations in stream processing. Hence, they can be used by a mapper to optimize the distribution of the limited system resources—processor cores—to the boxes.

Categories and Subject Descriptors C.4 [Performance of systems]: Modelling techniques


Keywords S-NET, SAC, LPEL, SVP, auto-parallelization, self-adaptation.

1. Introduction

Contemporary and future compute platforms can deliver massive raw performance, however, the problem is that each platform typically requires an application to be structured and optimized in a different way to take advantage of the hardware potential. Even if we constrain ourselves to just many/multi-core homogeneous general-purpose processor systems the prospect of fine-tuning a large concurrent application to a given hardware configuration (number of nodes, network topology, memory hierarchy) is daunting at the first thought, and impossible at the second when we realize that the effective hardware configuration may change dynamically at runtime due to frequency/voltage scaling of the cores (energy and heat management) and fault-tolerance (cores going temporarily or permanently off-line).

This paper is based on the work carried out during the EU-funded project ADVANCE (IST-248828) [5]. The goals are two-fold. The first is to create a tool-chain that integrates three otherwise independent technologies: SAC, S-NET, and SVP. These technologies are briefly described in Section 2.

The second goal of the project is to research and implement methods for dynamic on-line adaptation of concurrent programs (called ‘boxes’) to the underlying hardware platform. The idea is to use a light-weight monitoring system to gather performance and other data about the execution of programs in the streaming data-flow S-NET network. At the same time the application developer may supply performance annotations, or ‘hints’, to express various extra-functional properties of the concurrent programs. The measured data and the annotation would be analysed by a statistical framework and processed by a machine learning system to devise a complex model of the system. The model could be in turn used by a mapper to optimize the resource distribution.

2. Infrastructure

2.1 SAC – Single Assignment C

The Single Assignment C (SAC) [3] is a strict, purely functional programming language that combines the syntax of C with a high-level support for array programming. The language treats all variables as n-dimensional arrays, and provides overloaded operators and standard library functions to manipulate them. The compiler is geared towards high efficiency of the generated code. Essential SAC features such as call-by-value parameter passing for arrays, fully automatic memory management, architecture-agnostic programming and fully automatic parallelisation for different architectures set SAC apart from most other array-oriented languages.

2.2 S-NET

S-NET [4] is a declarative coordination language for describing streaming networks of asynchronous components. Streaming networks are defined using an expression language featuring four network combinators as operators: serial composition, parallel composition, serial replication and parallel replication. With the exception of serial composition, the combinators come in two flavours each: the deterministic versions preserve the order of data on streams, whereas non-deterministic variants trade this property for improved throughput. Two primitive components serve housekeeping and synchronisation purposes. Streams are associated with record types: collections of data where each item is uniquely identified by its name. Structural subtyping on records directs the flow of data through the streaming network.

2.3 SVP – System Virtualization Platform

The System Virtualization Platform (SVP) [1] is a hardware virtualization layer used in the ADVANCE project. The purpose is the separation of concerns between the expression of concurrency in the application source code, and the optimal concurrency granularity of the target platform. Without this separation the application developer would be forced to code using the granularity of the platform, making the code not portable to different target platforms.
2.4 CAL – Performance Annotations
The Constraint Aggregation Language (CAL) [6] allows us to write
a set of behaviour assertions for each system component, each of
which is guarded by some context condition. We use annotations at
the component level to reason about the behaviour of the system at
the coordination level.

S-Net components respond to a single input message with zero,
one or more output messages, with both the input and the output
using a single channel. Since an output message of one component
becomes an input message of another, implicative predicates can be
chained over connection primitives to aggregate functional con-
straints over the whole system.

2.5 LPEL – Lightweight Parallel Execution Layer
The Lightweight Parallel Execution Layer (LPEL) is a user-space
threading library for co-operative execution of tasks on workers.
The fundamental requirement is an efficient scheduling and execution
of a huge number of dynamically created tasks, often short-
lived, onto a fixed set of processor cores. The processors cores are
represented by workers that are implemented by standard POSIX
threads, pinned to individual processors in the system to improve
cache locality and ensure predictable execution.

The LPEL layer provides a dynamic stream primitive for task
communication. The advantage is that LPEL streams are exposed
to the task scheduler and the higher layers, therefore they can be
used to reason about task dependencies.

Finally, the LPEL layer provides cheap monitoring services for
gathering performance data of individual tasks and streams. The
performance measurements are available at run-time for statistical
analysis and dynamic adaptation services, and can also be stored in
a file for off-line analysis.

2.6 Automatic parallelization in SAC
The SAC compiler can generate multithreaded parallel code exe-
cutable on common shared-memory Unix-based systems [2]. The
compiler exploits massive data parallelism in the implicit array op-
erations of the language.

In SAC the basic building block (not only) for parallel execution
is a with-loop. A with-loop describes input/output arrays, an index
space, and a code block that is executed independently for each
point in the index space. The compiler optimization passes will first
merge (fuse) with-loops in the source code to enlarge the granular-
ity of the parallel block. Then the index space is partitioned, usu-
ally along the outer dimensions, into independent work-tasks.
At runtime the work-tasks are put into a pool (if dynamic scheduler
is used) and then they are executed concurrently by SAC bees. A
worker bee is an abstraction layer in the SAC runtime that repre-
sents a compute resource, i.e. a processor. In practice, worker bees
are implemented by POSIX threads when the program was com-
piled as a stand-alone application, or by LPEL tasks if the pro-
gram was compiled for the S-Net environment. There is always a
1:1 mapping of bees to the underlying execution entities, i.e. to the
POSIX threads or LPEL tasks.

An important feature is that worker bees are persistent across
the whole run-time of the SAC application. The user can choose the
number of bees at the program command line, hence allocating the
given number of processor cores of the system to the application.
Sequential parts of the application are executed by the queen bee
(the main program thread) and the other (slave) bees lay dormant.
Parallel parts are executed concurrently by all bees of the appli-
cation. The compiler ensures that there is no nesting of the parallel
mode.

3. Box performance modelling

In the S-Net port of the SAC runtime system the worker bees
are persistent across the life time of a single S-Net function box.
Each box has its own independent hive of bees as shown in Fig-
ure 1. The bees are implemented as LPEL tasks and hence they
are managed by the common S-Net mapper and scheduler.

3.1 Definition

Performance model of a box is function $\tau$:

$$\tau(p, f(x))$$ \hspace{1cm} (1)

where $\pi$ are model parameters, $p$ is the number of cores, $x$ is
an input record, and $f(x)$ is an optional user-supplied predictor
function, $f : X \rightarrow \mathbb{R}$, that evaluates the next input record and
predicts its compute complexity. The function $\tau$ estimates the box
running time (in seconds) for the given number of cores and the
input record.

3.2 Amdahl’s law
The inverse speed-up $S$ of a program code is limited by Amdahl’s
law:

$$S(\alpha, p) = \frac{\alpha}{p} + 1 - \alpha$$ \hspace{1cm} (2)

where $\alpha$ is the fraction of the program that is parallel, and $p$ is the
number of processor cores.
3.3 Basic constant workload model (record-independent)

The constant workload model is adequate for boxes that exhibit constant work-load in response to every input record. This basic model ignores record types as the \( f(x) \) is missing. The performance model \( T_{SEQ}(\alpha) \) is directly derived from Amdahl’s law:

\[
\tau(T_{SEQ},\alpha(p) = T_{SEQ} \cdot S(\alpha,p) = T_{SEQ} \cdot \left( \frac{\alpha}{p} + 1 - \alpha \right)
\]

where \( T_{SEQ} \) is the sequential time (in seconds) when only a single-core is used, and \( \alpha \) is the fraction of the program that is parallel.

3.4 Sequential time estimation

Using CAL annotations the application programmer may specify a predictor function \( f(x) \) that estimates the sequential run-time (i.e. the compute intensity) that a given record \( x \) will incur in the associated box. Thus, \( T_{SEQ} \) is re-defined as a function:

\[
T_{SEQ}(x) = f(x) \cdot \beta + \gamma
\]

The predictor function \( f(x) \) is meant to be specified using the big-O notation, hence the auxiliary parameters \( \beta, \gamma \) are needed for proper re-scaling. As an example consider a box with sequential run-time proportional to \( O(n^3) \). The predictor will be \( f(x) = n^3 \), where \( n \) is extracted from \( x \) in an application-specific way, and the estimator is \( T_{SEQ}(x) = \beta \cdot n^3 + \gamma \). The lower-order terms \( (n^2 \) and \( n) \) are ignored.

By substituting Eq. 4 into Eq. 3 we get an improved \( \pi = (\alpha, \beta, \gamma) \) performance model that takes into account the record-dependent estimation of the sequential running time:

\[
\tau(\pi,\beta,\gamma)(p, f(x)) = (f(x) \cdot \beta + \gamma) \cdot \left( \frac{\alpha}{p} + 1 - \alpha \right)
\]

3.5 Parameter extraction

Model parameters \( \pi \) are extracted from real-time measurements and observations in the running system. The parameters of the basic constant (record-independent) model in Eq. 3 can be estimated from two measurements:

\[
\{(p_1, t_1), \}
\]

\[
(p_2, t_2)\}
\]

and: \( p_1 \neq p_2 \)

where \( p_i \) is the number of cores (the independent variable), and \( t_i \) is the observed running time of the box in seconds. From these the \( \alpha \) and \( T_{SEQ} \) parameters are computed:

\[
\alpha = \frac{p_1 p_2 (1 - \frac{t_1}{t_2})}{t_1 p_1 (1 - p_2) - p_2 (1 - p_1)}
\]

\[
T_{SEQ} = \frac{t_1}{\alpha / p_1 + 1 - \alpha}
\]

In the improved record-dependent model in Eq. 5 the extraction of parameters is slightly more elaborate. Given three observations in the form \( (p, f(x), t) \):

\[
\{(1, f(x_1), t_1), \}
\]

\[
(1, f(x_2), t_2), \}
\]

\[
(p_3, f(x_3), t_3)\}
\]

and: \( f(x_1) \neq f(x_2) \wedge p_3 \neq 1 \)

we can directly compute \( \beta, \gamma, \alpha \):

\[
\beta = \frac{t_2 - t_1}{f(x_2) - f(x_1)}
\]

\[
\gamma = t_1 - \beta f(x_1)
\]

For an unsuitable set of observations (e.g. \( f(x_1) = f(x_2) \)) we may begin with the simpler constant workload model, and upgrade to the improved model only after acquiring a more diverse set of measurements.

3.6 Discretization and a zone-based estimator

A complementary approach to the model construction is a discretization of the continuous parameter space of the \( f(x) \) predictor(s) into a set of non-overlapping ranges or zones. An independent model could be constructed in each zone. This way different models could be combined to more accurately predict the performance of a complex box or even a sub-network. However, it is clear that automated construction of such combined models will require more advanced statistical techniques than presented in this paper, and it is left as a future work.

4. Network performance models

We consider here only a serial composition of boxes (a pipeline):

\[\text{A} \rightarrow \text{B} \rightarrow \text{C}\]

Throughput \( \phi \) of a simple (non-replicated) box is an inverse of its latency:

\[
\phi = \frac{1}{\tau}
\]

The total latency \( \tau_s \) [s] of a serial composition of boxes is the sum of individual latencies; the total throughput \( \phi_s \) [1/s] is equal to the smallest throughput in the pipeline:

\[
\tau_s = \sum \tau_i
\]

\[
\phi_s = \text{Min}\{\phi_i\}
\]

The pipeline can be optimized for minimal latency or maximal throughput. The global constrain is usually the total number of processor cores. In our analysis we assume the system is running steady-state.

5. Experiments

5.1 Set-up

Figure 3 shows the experimental set-up. The experimental system runs in a loop. Each iteration consists of the following tasks:

1. Getting the prediction \( f(x) \) (if available) from the next input record in the queue.
2. Computing the optimal mapping of boxes to processors using the box performance models and the \( f(x) \) prediction.
3. Running the boxes.
4. Reading and parsing the box measurements from log files.
5. Updating the box models based on the latest measurements.

In the centre of the figure there is a serial S-Net pipeline of three boxes (A, B, C). The boxes are artificial workloads implemented in the SAC language. Real SAC, S-Net and LPEL runtime systems are used as the infrastructure and for real-time measurements, however, the analysis and mapping subprograms described in this paper were implemented in an external Perl script.

The test-bench boxes A, B, and C in Figure 3 are artificial workload emulators written in the SAC language. Their source code is in...
Figure 3. Experimental setup.

Table 1. Box configurations in the experiment

<table>
<thead>
<tr>
<th>Iterations</th>
<th>Box A</th>
<th>Box B</th>
<th>Box C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(w)</td>
<td>(x)</td>
<td>(w)</td>
</tr>
<tr>
<td>1–4</td>
<td>100</td>
<td>0.0</td>
<td>100</td>
</tr>
<tr>
<td>5–9</td>
<td>200</td>
<td>0.0</td>
<td>300</td>
</tr>
<tr>
<td>10–13</td>
<td>300</td>
<td>0.0</td>
<td>200</td>
</tr>
</tbody>
</table>

Figure 4. The box-code entry point is in the \(\text{workload()}\) function. Its parameter \(\text{double}[.]\ V\) is an array of three numbers; the parameter is transported as a record via the S-Net streams. The parameter \(a\)-priory specifies the work-load the function will emulate at run-time: the total work-load \(\bar{w}\) and the fraction of it that is parallel \(\bar{x}\).

The total work-load is specified in an ad-hoc work-unit. The sequential part \((1 - \bar{\alpha})\) of the work-load is emulated in the \(\text{run_seq()}\) function by a for-loop. The parallel fraction \(\bar{\alpha}\) is emulated in \(\text{run_par()}\) by the parallel \(\text{with-loop}\). The SAC compiler will automatically distribute the \(\text{with-loop}\) to available cores.

The artificial workloads enable us to test the system under different conditions. Of course, the \(a\)-priory parameters of the emulated workloads are not used in the model construction. As the base work-load is specified in work-units the actual running time depends on the hardware configuration. Similarly, the specified parallel fraction \(\bar{\alpha}\) is usually not exactly the same as the observed one in the running system. The discrepancy reflects hidden overheads and/or optimizations in the SAC compiler and run-time. This is however not a problem because what we need is the models to accurately predict the observed behaviour of the boxes.

The hardware configuration of the experimental system was 8 core Intel Xeon CPU E5506 @ 2.13GHz, 4MB cache size, 12GB main memory, running 64-bit CentOS 6.3.

5.2 Experimental box configurations

Table 1 shows the hidden configuration of work-loads the boxes will emulate in run-time. The \(\bar{\alpha}\) parameter is constant for each box over all iterations. We have chosen the box A to be fully sequential \((\bar{\alpha} = 0)\), box B half-parallel \((\bar{\alpha} = 0.5)\), and box C to be fully-parallel \((\bar{\alpha} = 1.0)\). The base work-loads \(\bar{w}\) of the boxes are varied over time as shown in the table to simulates different compute complexities of records in real S-Net system.

5.3 Model construction

We construct the initial performance models of the boxes by handling the first two records (iterations) as a special case. We initially assume that all boxes can be represented by the constant-workload model (Eq. 3). Each box is given one core for the first record, and two cores for the second record. From these two initial measurements (not shown in the tables and plots) the base constant-workload models can be extracted using Eq. 9, 10.

The box work-load configuration values \(\bar{w}\) are used as the predictor function \(f(x)\). The analysis subprogram starts with the constant-workload models, and switches to the improved \((\alpha, \beta, \gamma)\) model when it sees a record with a different \(f(x)\).

Figure 5 shows the run-time data over all iterations. The three dashed vertical lines denote the three ‘phases’ from Table 1. The first four plots on the left hand side (a-d) show the evolution of the model parameters over the iteration steps. In the first phase (steps 1–5) the system uses the constant-workload model, hence the \(\beta\)
Figure 5. Experimental results. Colours: red = box A; green = box B; blue = box C.
and $\gamma$ parameters (plots c, d) are undefined. In the second and third phases the $T_{SEQ}$ parameter is undefined because the system uses the improved $(\alpha, \beta, \gamma)$ model.

Plot (e) on the right hand side of the figure shows the expected running time of the boxes as predicted by the models, and plot (f) shows the actual observed time in seconds. Prediction error of the models is in plot (g). The error is the worst in the fifth iteration because at the time the box configuration has been abruptly changed (cf. Table 1). The system adapts in step 6 by switching away from the constant-workload models to the improved models.

### 5.4 Mapping to cores

Mapping is a separate problem to the model construction. In our setting it means allocating processor cores to individual boxes. There are two basic strategies: to minimize box latency, or to maximize total throughput. As there are only 8 cores and 3 boxes in our experimental system, we simply generate all possible assignments of cores to boxes $(p_a, p_b, p_c)$ with the global constraint $\sum p_i = 8$, and compute the total latencies $\tau$ and throughputs $\phi$ using the Eq. 19. Table 2 shows the predicted optimal configurations in each iteration, optimized either for the minimal total latency or the maximal throughput. We have used the later in the experimental runs.

### 5.5 Results and discussion

The prediction error (Figure 5 (g)) is close to zero for all three boxes, except during the fifth iteration when we demonstrate the capability to adapt the models themselves in response to the availability of the prediction data $f(x)$.

An interesting data point is the iteration No. 10 when we have changed the emulated box workloads again (see Table 1). However, this time the models were able to accurately reflect new running times thanks to the prediction data $f(x)$.

One possible objection to our experimental set-up is that we assume that the parallel fraction $\alpha$ is constant in each box even when the work-load is varied. We make this simplifying assumption because the usual SAC applications (such as image processing) tend to be regular and highly data-parallel. Moreover, it is not clear if the same approach we have used in the $T_{SEQ}$ estimator in Eq. 4, i.e. the use of the big-O notation for the $f(x)$ predictor and the $\beta, \gamma$ parameters for its re-scaling, applied to an $\alpha$ estimator, would be transparent and intuitive for the system developer.

### 6. Conclusion

The presented preliminary results are encouraging. We have used two relatively straightforward analytical performance models based on Amdahl’s law to predict execution times of function boxes in S-NET streaming networks. Free parameters of the models are extracted at run-time from the on-line performance measurements. The models can be augmented by application-level ‘hints’ in the form of predictor functions that account for the varying compute complexities of individual data records flowing in the network.

Our approach is enabled by three key technologies: (a) the capacity of the SAC language to execute the data-parallel code in a run-time configurable number of cores; (b) the data-flow nature of the S-NET system that has a handle on the repetitive ‘firing’ of the boxes; (c) the SVP hardware abstraction layer along with the LPEL threading library that provides the monitoring and scheduling services.

### Acknowledgments

This work has been supported from EU-funded FP-7 project “Asynchronous and Dynamic Virtualization through performance ANalysis to support Concurrency Engineering (ADVANCE)” under contract no IST-248828.

### References


Influences on Throughput and Latency in Stream Programs

Vu Thien Nga Nguyen
University of Hertfordshire
v.t.nguyen@herts.ac.uk

Raimund Kirner
University of Hertfordshire
r.kirner@herts.ac.uk

Abstract
Stream programming is a promising approach to execute programs on parallel hardware such as multi-core systems. It allows to reuse sequential code at component level and to extend such code with concurrency-handling at the communication level. In this paper we investigate in the performance of stream programs in terms of throughput and latency. We identify factors that affect these performance metrics and propose an efficient scheduling approach to obtain the maximal performance.

1. Introduction
With the current trend towards increasing number of execution units running in parallel, stream programming has been emerged by its elegant way of exposing useful types of parallelising. This programming paradigm separates communication from computation. This separation relieves the programmer of managing concurrent communication and synchronisation at the same time.

Because of this advantage, several research projects have introduced stream programming frameworks such as StreamIt [14], Brook [2] and S-Net [7]. However, it is intricate to understand the performance of stream programming. Possessing properties similar to communication networks, stream programs are usually evaluated in terms of latency and throughput. These two performance metrics depend not only on internal factors such as scheduling policies but also on external factors such as the arrival rate of data.

In this paper, we study the influences on the performance of in stream programs in terms of throughput and latency. The paper i) shows that different ranges of the data arrival rates give different influence on the performance; ii) analyses in which way the scheduling policy affects the performance; and iii) proposes a scheduler aiming to achieve the optimal performance of stream programs in multicore systems. The proposed scheduler consists of a static strategy targeting at the maximum throughput; and a heuristic dynamic strategy which observes the runtime behaviours and automatically tunes the policy to obtain the optimal latency while keeping the maximal throughput.

The paper is organised as follows. Section 2 provides basic backgrounds including stream programming paradigm and the stream execution model. Section 3 discusses latency and throughput in stream programming. Section 4 presents different ranges of input rate with different affects on latency and throughput. Section 5 investigates on the scheduler’s influences on the latency and throughput. Based on these analysis, Section 6 proposes a scheduling strategy to achieve the optimal performance. Section 7 describes related work and Section 8 presents conclusions and planned future work.

2. Backgrounds

2.1 Stream Programming Paradigm
Stream programming is a paradigm that allows to express the parallelism by decoupling computations and communications [2, 7, 14]. In this model, a program is represented as a directed graph whose vertices are computation nodes and edges are communication channels called streams. For short, we from now use nodes instead of computation nodes.

Data is expressed as a flow of messages moving between nodes via streams. Streams connect nodes in different ways such as pipeline, parallel, feedback, etc.

A node receives messages from a set of streams, called input streams. The node then processes these messages to produce new messages and sends to a set of streams, called output streams. For one execution, a node may consume n messages and process to produce m messages. The value of n and m may change at the runtime, for example depending on the input data. The node can progress only when n required messages from its input streams are available.

Nodes that receives messages from outside of the stream programs are entry nodes. Similarly, nodes sending messages to outside of the stream programs are exit nodes. A stream program can have multiple entry nodes and multiple exit nodes. Messages coming from outside are called input messages and messages sending to outside are called output messages. Other messages (messages inside the stream program) are called intermediate message.

![Figure 1. An example stream program](image)

Figure 1 shows an example of a stream program. In this example, the computations are done on 5 nodes: A, B, C, D, E. A is an entry node and connected B in pipeline. B is followed by the parallel combination of C and D. This combination is then pipelined with an exit node E. A message produced by B can take either the route via C or D, i.e. can be processed either by C or D.

When a node consumes a message X (and possibly other messages) to produce a message Y (and possibly other messages), it is said that X derives to Y, or Y is derived from X. In this case, X is a predecessor of Y and Y is a successor of X. The predecessor - successor relation is inherited, i.e. if X is predecessor of Y and Y is predecessor of Z then X is also predecessor of Z.

Copyright is held by the author/owner(s).

FD-COMA 2013 2nd HiPEAC Workshop on Feedback-Directed Compiler Optimization for Multi-Core Architectures, part of the 8th International Conference on High-Performance and Embedded Architectures and Compilers, Berlin, Germany, January 21-23, 2013
ACM SIGPLAN conference style.
An input message $I_i$ when processed by a stream program derives to multiple intermediate messages $M_j$ before deriving to output messages $O_k$. Figure 2 shows an example of input messages deriving to output messages. In this example, input message $I_2$ derives to $M_3$, $M_1$ derives to $M_4$ and $M_1$ derives to output message $O_4$. $M_1$ (together with input message $I_3$) also derives to $M_5$ and then $M_5$ derives to $O_4$. An input is said to be completed when all of its successors no longer exist in the stream program. That is usually when all its output messages are produced.

2.2 Stream Execution Model

A typical framework for executing a stream program is shown in Figure 3. A stream program first is compiled into runtime objects managed by the Runtime System (RTS). These objects include streams and tasks. Each stream is represented as a FIFO buffer for storing messages. Each computation node is associated with a task. Typically each task receives messages from a set of $n$ streams, called input streams. The task then processes messages before sending them to a set of $m$ streams, called output streams.

3. Throughput and Latency in Stream Programs

Stream programs are similar to communication networks in the sense that they transfer messages from one end to another via interconnected nodes. For these reasons, the performance of stream programs should be evaluated with similar metrics of communication networks, i.e., throughput and latency. However, unlike in communication networks, nodes in stream programs contain computations. These computations need to be executed on a mutual platform of physical resources (e.g., CPUs). This section discusses the influences on throughput and latency in stream programs.

3.1 Throughput

Similar to communication networks, Throughput ($T_p$) of stream programs is measured as the number of input messages are completed per time unit. Each node in a communication network is a physical resource with its own throughput. Queuing theory therefore can be used to analyse the throughput with some assumptions [12].

However, queuing theory is not applicable for stream programs as nodes perform their computation on a shared platform of physical resources. The throughput of stream programs highly depends on the scheduling policy.

3.2 Latency

Generally, latency is the delay experienced in the system, i.e. the delay to transfer one message from one entry point to an exit point. In stream programming, latency of an input message is the time interval from when the input message arrives to when it is completed.

Latency in stream programming may vary for different input messages for two reasons. First, each node may take different amounts of time to process different messages. Second, a message can take different route inside the stream programs. In the example shown in Figure 1, a message produced by $B$ can pass either $C$ or $D$ which may take different time to process the message.

On a platform with limited physical resources, the latency of an input message also depends on the work load when the input message arrives and the scheduling policy which decides when the stream program starts to process the input message. If the scheduler decides to process the input message immediately, it may increase the latency of previous input messages as processing the new input will bind some physical resources. Otherwise, the input message will wait in the queue before getting into the stream program. Since the queuing time of an input message is affected by resource limitation and the previous input messages’ successors still to be processed, it is should be separated from the processing time of the
input message. We therefore propose different kinds of latency as follows.

- **Queuing Latency (QL)** of an input message is the time it waits before getting processed by the stream program. The queuing latency depends on when the scheduler decides to start processing a new input message; and how frequently new input messages arrive at the stream program.

- **Processing Latency (PL)** of an input message is the time interval from when it is taken into the program until it is completed. Processing the input message’s successors may not be continuous for two reasons. One reason is that one of its successors when reaching a node needs to wait for other messages so that the node has sufficient messages to make progress. Another reason is that the scheduler decides to halt processing the input message’s successors due to the lack of physical resources.

- **Overall Latency (L)** is the sum of queuing latency and processing latency. For the simplicity, from now on we use latency to refer to overall latency.

For communication networks in which each node has its own physical resource, the latency over the whole network can be reasoned from the latency over individual nodes. This is non-trivial for stream programs in which nodes share a mutual number of physical resources within the scheduler’s control.

4. **Input Rate affects on Throughput and Latency**

For most of stream programs, input messages arrive with a rate, called **Input Rate (IR)**. The IR along with the scheduling policy are key factors controlling on the program’s throughput and latency. In this section, we focus on the effects of the input rate alone on the throughput and latency.

With a specific platform of physical resources and a specific scheduling policy, Figure 4 shows a typical change in latency and throughput according to the input rate. There are two marks that divide the IR value into three ranges. Within each of these ranges, the IR affects the throughput and latency in different ways. For the simplicity, we assume that the stream program has exclusive usage of the physical resources, i.e., no other application is running at the same time on the platform.

![Figure 4. Latency and Throughput are affected by Input Rate](image)

4.1 **Idling Range**

As explained in the previous section, the latency of an input message depends on the system load, i.e., how busy all the resources are. At the beginning, when the first input message arrives all the resources are free and immediately invoked to process the input message and its successors. The smallest latency is therefore achieved for the input message.

When the IR value is small enough that all the physical resources are always free whenever an input message appears, all input messages have the smallest latency value. Thus the average latency is also smallest and this value is called **Trough Latency**. The throughput in this case equal to the IR.

There might be a case where the average latency is decreased when the IR increased. That is when a node requires successors from two input messages to make progress. That means the processing the former input message can not continue until the later input message arrives. In this case, the latency is decreased when the IR gets higher but this is a temporary improvement. When the IR reaches a value that the faster arrival of input messages do not help nodes make progress earlier, the latency stops decreasing and reaches the stable value. The trough latency in this case is this stable latency value. Throughput in this case is smaller than but still proportional to the IR.

The highest IR value at which the system still keep the trough latency is called **IR_{trough}**. Whenever the actual IR is smaller than this value, there are time intervals where the physical resources are idling and waiting for input messages. For this reason, the period from 0 to **IR_{trough}** is called **Idling Range**.

4.2 **Working Range**

When the IR exceeds the **IR_{trough}** value it can no longer be guaranteed that the system exclusively processes one message at a time. Instead the processing of messages tends to get overlapped. When an input message arrives, the physical resources are partly or completely busy for processing previous input messages’ successors. The latency of an input message gets higher either because the input message has to wait in the queue or because the resources are not fully used to process the input message and its successors but to contribute to process others. For this reason, the average latency is therefore higher than the trough latency.

While the latency gets worse, the throughput becomes better. The high input arrival rate gives the stream program the chance to consume input messages faster. More input messages are processed and therefore the throughput gets higher. However when the IR is high enough to saturate the system, the throughput stops increasing. The highest throughput value that the system can achieve is called **Peak Throughput**. The IR value at which this happens is called **IR_{peak}**.

We call the IR range from **IR_{trough}** to **IR_{peak}** as **Working Range** as with the IR in this range the platform does not have any moment idling but working constantly.

4.3 **Overload Range**

With an IR higher than **IR_{peak}**, the throughput does not exceed the peak value. The system cannot consume input messages as fast as input rate. The platform gets saturated after an initial period. After this initial period input messages have to wait before getting processed. The later the input message comes, the more input messages are already waiting in the queue before it. The queuing latency therefore rises up and eventually becomes infinity. As the result, the average latency is infinity.

In this circumstance, the platform is saturated and cannot keep up with the requested input rate. Hence the range from **IR_{peak}** up is called **Overload Range**.

5. **Scheduler affects on Throughput and Latency**

As presented in Section 2.2, the scheduler manages a set of workers and a set of tasks. One worker is associated to a physical resource of the platform and one task is associated to a computation node.
of the stream program. The scheduler decides when and how long a worker performs a task. This decision has a strong influence in throughput and latency of the stream program.

### 5.1 Throughput

Consider a stream program deployed on a platform of homogeneous physical resources. Let N be the number of workers. To process M input messages, it takes the stream program the time period P. The total time that N workers have spent over that period is:

\[ T = N \times P \]  

Within the period P, N workers spend time C on computations and time W on idling. The sum of computation time and the idling time must be equal to the total time that N workers have:

\[ T = C + W \]

From Equation 1 and Equation 2, we have:

\[ T = N \times P = C + W \]  

We have the throughput of the stream program is equal to the number of input messages over the time period:

\[ T_p = \frac{M}{P} \]  

From Equation 3, we have:

\[ T_p = \frac{M \times N}{C + W} \]  

As C is the required computation time for processing M input messages, it varies on the implementation and the underlying hardware. These factors are not under the sphere of control of the scheduler. We thus can assume C is constant in the scheduler’s context. Throughput then is inverse-proportional to the idling time.

There are two situations that a worker gets idling. One situation is that there is no ready task because the input rate is too low. As explained in Section 4, when input rate is in idling range the throughput is as low as the input rate. The other situation happens when the input rate falls in two other ranges. There are ready tasks and also free workers at the same time. This happens usually due to the mapping strategy of the scheduler.

The scheduling policy determines the idling time and therefore controls the throughput value. If the scheduling policy helps to achieve higher throughput during the working range, the stream program can consume more input messages and therefore can deal with higher input rate. As a result, it expands the working range by increasing \( IR_{peakTP} \) value, i.e. IR needs a higher value to saturate the system. The peak throughput also is increased.

### 5.2 Processing Latency

As described in Section 2.2, latency is the sum of queuing latency and processing latency. The queuing latency of an input message reflects the time the input message has to wait before getting processed. Therefore the queuing latency depends on the speed at which the stream program consumes input messages. That means the queuing latency depends on the throughput. In this section, we focus on the processing latency which is directly controlled by the scheduler.

The processing latency of an input message is the time it takes to make the input message completed, i.e. the period from when it is consumed until all its output messages are produced. Within this period, the input message itself and all its successors are processed. If these messages are processed continuously, the latency is minimal. This is decided by the RTS and the scheduler.

As explained in Section 2.2, to preserve the stream program’s semantics the RTS decides when a task is ready. The scheduler affects processing an input messages and its successors by deciding the order of executing ready tasks. In particular, the scheduler determines when a ready task is executed and for how long.

The scheduling policy affects the latency in two first ranges of the input rate as the latency rises up to infinity when the input rate falls in the overload range. Within these two ranges, while deciding the executing order of ready tasks, the scheduler effectively has an influence on the latency and also the trough latency. If the scheduler helps to reduce the trough latency, the stream programs can process input messages faster. That means the workers get idling more often in the idling range. Therefore the input rate needs a higher value to get in the working range, i.e., the \( IR_{through} \) is reduced.

### 6. A Proposed Scheduler for Maximising Performance

In this section, we propose a scheduler aiming to maximise the throughput and minimise the processing latency. We focus only on the processing latency because the queuing latency depends on both the input rate and the throughput. When the throughput is maximised, the queuing latency is also minimised as messages are taken to by the program with the highest rate. Hence while maximising the throughput, the proposed scheduler effectively reducing the queuing latency.

Generally, all ready tasks are stored in a central task queue as in Figure 5. Each task in the task queue has a priority which is calculated dynamically during the runtime. A worker requests for a task from the task queue when it gets idling. The task with the highest priority at that time will be chosen and assigned to the requesting worker. The worker executes the task until when either the task becomes not ready or it is asked to yield.

The following of this section provides more detail about the scheduling policy and explains how it steers the performance.

![Latency of an individual processing component](image.png)

### 6.1 Maximising Throughput

As shown in Section 5.1, the throughput is maximal when the idling time is minimal assuming that the individual processing times of messages are independent from the scheduling decisions. In the proposed scheduler, each worker requests for a task whenever it is idling. If there is a ready task in the central task queue, it will be assigned to the worker immediately. By doing this, each worker is kept busy as much as possible. A worker has to wait only when there is no ready task in the central task queue because input messages have not arrived. The input rate is not under the sphere of control of the scheduler.

Therefore by using a central task queue, the scheduler minimises the idling time. However this also raises a risk of overhead.
In particular, the central task queue can become a bottleneck when too many workers request at the same time.

For the implementation, an efficient data structure should be used to implement the central task queue. Also it is suggested to dedicate a worker as a conductor which controls the central task queue. In this case, a sufficient communication protocol between the conductor and the workers is necessary in order to reduce the delay from when a worker sends a task request till receiving one.

6.2 Minimising the processing latency

As presented in Section 5.2, how an input message and its successors are processed has a strong affect on the processing latency. As the RTS applies some constraints to preserve the stream program’s semantics, the scheduler has control only on ready tasks. When a worker becomes free, the scheduler can make two decisions: which ready task is executed and for how long. In general, a message can be routed to streams dynamically during the runtime as explained in Section 2.1. Thus it is difficult to make these decisions at the compile time.

Our proposed scheduler uses a heuristic to compute the task priority based on the state of input and output streams of the task during the runtime. Let \( I \) be the number of messages in the task’s input streams and \( O \) be the number of messages in the task’s output streams. The \( I \) value of entry tasks are always zero and the \( O \) value of exit tasks are always zero. The task’s priority is calculated as follows.

\[
P = \frac{I + 1}{(O + 1)(I + O + 1)}
\]

Equation 6 implies that a task’s priority has a positive correlation with its \( I \) value and a negative correlation with its \( O \) value. When a task consumes \( n \) messages and produces \( m \) messages, it lowers its \( I \) value by \( n \), raises its \( O \) value by \( m \) and also raises the succeeding task’s \( I \) value by \( m \). That means, the task’s priority is reduced while the following task’s priority is increased. This increases the chance of scheduling the succeeding task and therefore the message’s successor will be processed earlier. This therefore lowers the latency of the input messages that are the message’s predecessors.

After a task has processed one message, the priorities are reevaluated. The scheduler then pick the highest-priority task to assign to a worker. By doing this, the priority is finely updated and therefore the heuristic is more accurate. However, this also raises the overhead for switching tasks and calculating the priority. For the implementation, a task when scheduled should process a limited but large number of messages in order to compensate the cost of loading the task.

7. Related Work

Stream programming has seen a revival over the last years, due to the main-stream multi-core computing. Gordon et al. have identified three forms of parallelism to be exploited in stream programming: task parallelism, data parallelism, and pipeline parallelism [5]. The performance of a stream program can be optimised by task allocation, resource mapping, and scheduling policy.

One of the prominent stream programming approaches is StreamIt [13], which is conceptually based on synchronous data flow (SDF). In the initial version of the language, all flow rates have to be static, even though this can be avoided in later versions of the language.

There are many approaches of performance-optimised scheduling of StreamIt programs. All encountered approaches assume a static structure of the streaming graph, as StreamIt currently requires at compilation time fixed message routes, i.e., the connection of network nodes is determined at compile time. Karczmarek et al. introduced the concept of phase scheduling for StreamIt programs, exploiting the static nature of the streaming graph [10]. The goal of phase scheduling is to address the trade-off between code size and data size [1] without considering the actual performance of the program. The concept of phase scheduling is to construct the stream processing as a repetition of phases. Based on this construction, one strategy is to deploy each phase on one resource [15]. The phases are executed in parallel to take advantage of data parallelism. Another scheduling strategy is to execute on each resource a separate phase [6]. In this case relatively long waiting times can happen if synchronisation between different phases happens.

Farhad et al. recently used efficient approximative algorithms with bounded inaccuracy to schedule StreamIt programs [3]. In a different approach they built a model of communication costs to improve performance-guided optimisations [4].

Stream programming has been also introduced for hard real-time computing. Giotto is a coordination language where nodes are scheduled in a strict static time-triggered way [9]. Message processing is synchronous, i.e., a constant number of messages is consumed and produced per invocation by a node. With such a static language as Giotto is, one actually has to solve the inverse problem to that ours: with Giotto one starts with the performance values as specification, and has to produce a schedule in order to fulfill the required performance constraints.

With our discussion of performance of stream programming we include also more dynamic network structures of stream programs, as this, for example, is the case with the coordination language S-Net [8]. Streaming programs in S-Net can have a dynamic network structure, which requires the use of a dynamic scheduling algorithm. In S-Net there are no constraints over the flow rate of input. Thus, actual throughput and latency depend on the concrete message values and timings of the input stream. CAL (constraint aggregation language) has been developed to provide contracts of the behaviour of network nodes [11].

8. Conclusion & Future Work

In this paper we have discussed the performance measures throughput and latency for stream programming. We have contrasted them to the established calculation of throughput and latency in the domain of communication networks. Even though there are many similarities, the subtle but important difference is that communication networks typically have a separate resource for each network node, while in stream programming typically multiple nodes have to share each processing resource. Thus the behaviour of stream programs get additional dependency, the resource allocation policy, often described by the allocation, mapping, and scheduling of tasks on resources.

Most discussions of performance of stream programs have been based on the assumption of the special case of static network structures, synchronous communication, and fixed data rates. Important insights are the formation of schedule-dependent trough-latency and peak throughput. Within this paper we discuss more generally the influence of scheduling on throughput and latency of stream programs. In addition, we outline a scheduling approach for such dynamic streaming programs.


Task Migration for S-Net/LPEL

Merijn Verstraaten, Stefan Kok, Raphael Poss, Clemens Grelck
Informatics Institute, University of Amsterdam, Netherlands
m.e.verstraaten@uva.nl, stefan.kok@student.uva.nl, r.poss@uva.nl, c.grelck@uva.nl

Abstract
We propose an extension to S-Net’s light-weight parallel execution layer (LPEL): dynamic migration of tasks between cores for improved load balancing and higher throughput of S-Net streaming networks. We sketch out the necessary implementation steps and empirically analyse the impact of task migration on a variety of S-Net applications.

1. Introduction
S-Net is a dataflow coordination language and component technology [4, 6]. As a pure coordination language S-Net provides (almost) no means to describe computations of any kind, but it turns regular functions/procedures implemented in a conventional programming language into asynchronously executing, state-less components, named boxes. In principle, any conventional programming language can be used, but for the time being we provide interface implementations for the functional array language SAC [5] and for a subset of ANSI C.

S-Net components are connected by and solely communicate via uni-directional typed streams. Fig. 1 shows an intuitive example of an S-Net streaming network. Data objects enter the streaming network via a dedicated input component and then travel alongside the streams to compute components. Whenever a data object arrives at box, it triggers a computation as specified by the corresponding box language function (or procedure). During this computation a number of data items may be sent to the output stream to trigger further computations in subsequent boxes. Eventually, data objects reach the dedicated output box, which writes them to file or some other output medium.

S-Net streaming networks are not static, but evolve over time. In Fig. 1 this can be seen best with the box named C. This box is replicated in parallel meaning that data objects are routed to some instance of C as indicated by a named index in the data object itself. Hence, instances of C (which could also recursively be complete S-Net networks again) are instantiated as needed. The other dynamic network aspect is serial replication. In Fig. 1 this is indicated as a feedback loop around the parallel replication of box C, but in fact there is no feedback in S-Net, only feed forward (among others to rule out deadlock by construction). Effectively, the entire network within the “feedback loop” is dynamically replicated and the replicas are connected by streams one after the other. Data objects entering a serial replication network are routed through an a-priori unknown number of replicas. Before and in between any two such replicas a certain program-dependent condition is checked and the data either routed to the next instance of the replication or to the subsequent network (i.e. box D in the example of Fig. 1).

Serial and parallel replication can arbitrarily be nested, contributing much to the expressive power of S-Net. Consequently, the number of box instances in a running S-Net networking quickly grows and demands a smart mapping to compute resources, e.g. the various cores of contemporary server system or cluster node. While the deployment and operational execution of streaming networks is handled by the S-Net runtime system [3], the mapping of boxes to cores as well as the stream communication with suspension and activation of boxes is handled by the underlying Light-Weight Parallel Execution Layer (LPEL) [10].

Whenever the S-Net runtime system (due to replication) instantiates a new component, the LPEL layer maps it to some core for execution according to some heuristics. Once mapped a component remains tied to that core for the duration of program execution. This may lead to load imbalances where some cores have a pile of data objects to be processed while others remain idle. The highly dynamic nature of S-Net and the coordination approach that deliberately limits information exchange between compute and coordination layer (boxes are effectively black boxes) very much limit any form of static analysis and scheduling.

Hence, in the work presented in this paper we extend the LPEL threading layer by means for dynamic task migration. Firstly, we redefine the interface between LPEL and the S-Net runtime system box language interface to temporarily yield control to LPEL between any two data objects to be processed by some box. This gives LPEL a handle to change the mapping of components on this occasion. Secondly, we define an asynchronous scheduler task (a migration controller) that continuously observes the load balancing status of a running streaming network. According to selectable heuristics the migration controller may choose to asynchronously update the mapping of components to cores. The LPEL layer in turn implements the re-mapping, which becomes effective with the next data object to be processed.

The remainder of the paper is organized as follows. In Section 2 we provide additional background information on S-Net, its runtime system and the LPEL threading layer. Section 3 describes our technical contribution on task migration in greater detail, followed by an experimental analysis in Section 4. In Section 5 we draw conclusions and outline directions of future work.
2. S-NET: Design and Implementation

2.1 S-Net language

The basic building blocks of S-NET streaming networks are boxes. Each box is connected to the rest of the network by two typed streams: one for input and one for output. Following the data flow principle, a box is triggered by receiving a record on its input stream, upon which the box applies its box function to the incoming data object. As pointed out before, this box function is implemented in a box language selected for suitability in the relevant application domain. During execution the box may send records to its output stream. As soon as execution of the box function has finished, the box is ready to receive and process the next item on the input stream.

It is a distinguishing feature of S-NET that it neither introduces streams as explicit objects nor defines network connectivity by explicit wiring. Instead, S-NET uses algebraic formulae for describing streaming networks in a much more abstract way. The restriction of the boxes to single input streams and single output streams (named the SISO principle) 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, again is an SISO entity.

Let $A$ and $B$ denote two S-NET networks or boxes. Serial composition $(A \cdot B)$ 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. Parallel composition $(A \parallel B)$ constructs a network where incoming records are either routed to $A$ or to $B$; their output streams are merged to form the compound output stream. The system controls the flow of records. Serial replication $A^{\times type}$ constructs an infinite chain of replicas of box or network $A$ connected by serial combinators. The chain is tapped before every replica to extract records that match the type specified as the second operand. Last not least, parallel replication $A^{!<\text{tag}>}$ also replicates box or network $A$, but this time the replicas are connected in parallel. All incoming records must carry a property $<\text{tag}>$ whose integer value determines the replica to which the record is routed. These four orthogonal network construction principles are sufficient to define complex streaming networks.

For more detailed information on the S-NET language we refer the interested reader to [4, 7].

Each component, both internal split and merge components as well as user-level boxes, runs a simple event loop. First, a component checks the input stream for data. If the input stream is empty the component suspends. Otherwise, the first data item on the input stream is consumed and processed. If this processing requires sending a data item to an output stream, the component may suspend on a full output stream. If a component completes processing one item, it continues from scratch. Taking a data item out of a stream automatically wakes up components suspended on sending data to this stream. Likewise, adding a data item to some stream wakes up components suspended on reading from this stream.

2.2 S-Net runtime system

The S-NET runtime system [3] is responsible for deployment and operation of streaming networks. Thanks to the serial and parallel replication combinators networks evolve dynamically, and thus deployment and operation are not two distinct phases, but rather alternating, i.e. the operation of some network component may trigger another replication and, thus, the further deployment of network structures.

Furthermore, the S-NET runtime system turns implicit split in merge points in the construction of networks into active internal components that explicitly split an incoming stream into two (or more) outgoing streams by implementing the routing protocol or that merge two (or more) incoming streams into a single outgoing stream. As internal routing components these splitters and mergers do not comply to the SISO principle, but effectively implement the various routing protocols derived from the S-NET network combinators. Fig. 2 illustrates a partially deployed state of the example network introduced in Fig. 1. For illustration reasons, splitters and mergers are represented as (anonymous) triangles, but in fact each split and merge component does have a proper identity.

2.3 LPEL threading layer

The S-NET runtime system relies on basic threading mechanisms such as task creation, suspension, wake-up and termination. Such mechanisms are essentially provided by any multithreading library, including PThreads to name a specific one. However, even fairly simple S-NET streaming networks with nested replication combinators induce a large number of components to be instantiated at runtime. This motivates a two-layered approach where a small number of kernel threads essentially abstract the compute resources (cores) to be used while the tasks demanded by the S-NET runtime system are implemented by lightweight user-level thread contexts that are cooperatively scheduled among the kernel threads.

The Light-Weight Parallel Execution Layer (LPEL) [10] is such a two-level threading implementation tailored to the needs of the S-NET runtime system. On initialization LPEL creates a user-specified number of worker threads. These workers are kernel threads and, thus, preemptively scheduled by the operating system to the available cores. The general assumption is that the number of workers does not exceed the number of cores, and workers are bound to individual cores to effectively deactivate the operating system scheduler.

The instantiation of some S-NET component during a deployment phase incurs the creation of an LPEL task, or light-weight thread. This task is assigned to some worker based on some heuristic. Important for the subject of this paper: tasks are never reassigned (or migrated) from worker to another once created. Each worker has a priority queue of ready tasks and a queue of suspended tasks that wait for data on an empty stream or for space on a full stream. Reading from and writing to streams accordingly moves tasks between these queues not dissimilar to standard operating system procedures.

3. Task Migration

In this section we will discuss the task migration framework developed for S-NET and LPEL.

3.1 Challenges

Conceptually, S-NET boxes are nothing more than (pure) functions that are called on some incoming data item. As a result, migrating tasks between workers should be as simple as sending the input data to a different worker and having that worker perform the next function invocation. However, as already pointed out in Section 2.2 the S-NET runtime system implements boxes as long-lived tasks with an internal event loop triggered by receiving data on the input stream and by sending data to the output stream.

Migration of such long-lived tasks would involve halting the task, migrating the task’s current state (including state of the computation, such as the stack) and then resuming the task. This would be doable in a shared memory system, but with an eye on DISTRIBUTED S-NET [2] and NUMA architectures, we want to make the migration of state as explicit as possible to simplify any future work in these areas.
Another migration challenge is that any migration mechanism, and its associated heuristics, will introduce overhead. This overhead should be less than the performance gained by performing the migration, otherwise there is nothing to be gained from migrating tasks.

3.2 Respawning

As a first step towards task migration we modified the S-NET runtime system to expose more fine-grained concurrency: each task becomes a one-shot activation of an entity that handles a single input record. The simple implementation of this idea is to have every S-NET task spawn a new copy of itself upon termination. However, this would introduce a significant amount of overhead for the common case where a task does not migrate. This is due to LPEL performing expensive allocations upon LPEL thread creation, such as the task’s stack. These allocations can be reused when a task does not migrate to a different worker.

To solve this issue we implemented thread continuations in LPEL, this means that every thread has an optional continuation associated with it. Whenever a thread exits, this continuation is checked by LPEL. If the continuation is set, LPEL execute the continuation is the current thread context. The S-NET implementation of spawning a new task for the next activation can then be achieved by setting the continuation to the executing function.

As a result, control flow is returned to LPEL at the end of each activation of an S-NET entity. At this point, LPEL can check whether the task should be migrated to another worker or not. If the task has to migrate, LPEL will simply spawn a new thread on a different worker and execute the continuation in that thread.

3.3 Synchronous vs asynchronous migration

Now that LPEL is capable of migrating tasks between workers, we need a way to decide when to migrate tasks. An approach that immediately comes to mind is to define a placement oracle which is consulted on each continuation. This would be simple to implement, but would likewise introduce a significant amount of overhead if the oracle requires a non-trivial amount of computation. This follows from the every worker having to do a blocking invocation of the placement oracle upon each continuation of an S-NET task.

Rather than following the above synchronous approach, we decided to make placement decisions asynchronously. For this we extend the LPEL thread control structure with a next-worker field that indicates the worker on which the next continuation should run. This means that LPEL checks, on each continuation, whether the current and next worker are the same. If they are, the continuation is invoked. If, however, the next worker is different from the current, LPEL spawns the continuation on the new worker, effectively migrating the task.

3.4 Placement scheduler

The open question is still where, when and how the next-worker field is updated. As a starting point we introduce the notion of a placement scheduler. This is a conceptual task in the LPEL system that periodically inspects tasks and determines whether they should migrate on their next invocation. The placement scheduler is set up so that it can use any arbitrary oracle to decide the new placement.

As a small starting experiment to test the migration code and placement scheduler we implemented two very simple strategies for placement. To accommodate these strategies we added optional hooks to each scheduling event. These hooks update any strategy specific state that is used by the placement scheduler to determine placements.

3.5 Placement strategies

The first implemented strategy is random migration. After every invocation a task is marked for migration with probability $p$. The placement scheduler updates the next worker field of selected tasks with a random worker. This strategy can then be used as a baseline to see whether placement has any effect (positive or negative) at all, in terms of performance gain or overhead introduced.

The second strategy does placement based on the waiting times of tasks. That is, the time that a task is runnable, but not running. The waiting time $T_{\text{ready}}$ is the sliding window average of the past $n$ run-suspend cycles. For every worker we maintain the average $\mu_{T_{\text{ready}}}$ of the $T_{\text{ready}}$ of each task on that worker. A task is selected for migration if its $T_{\text{ready}}$ is larger than the $\mu_{T_{\text{ready}}}$ of its worker.

The goal of this strategy is to minimize the time a ready task spends waiting to run, aiming at increasing the average utilization of workers and balancing their loads.

4. Analysis

In this section we investigate the performance impact of task migration. Our first implementation of task migration did not yet use scheduling hooks to update the migration state of tasks. With this we did experiments on a dual 6-core Intel(R) L5640 2.27 GHz Xeon(R) system with 24 GiB memory. Our first benchmark was an S-NET adaptation of raytracing following a standard domain decomposition approach [9].

The results of our experiments are shown in Figure 3 and in Figure 4. It quickly became apparent that task migration, regardless of the concrete placement strategy, had an adverse effect on runtimes and scalability of the S-NET raytracer. The overhead introduced by the placement scheduler clearly outweighs any potential benefits of task migration.

After these disappointing results we redesigned the implementation of the placement scheduler to avoid unnecessary locking and use atomic operations where synchronization between threads could not be avoided. This new implementation reduced the overhead created by the placement scheduler to an insignificant amount in the range of timing accuracy.

However, we still did not observe any benefits of task migration. To find an explanation for this at first glance counter-intuitive behaviour, we thoroughly analyzed the S-NET raytracer application. It turned out that the raytracer implements a very regular concurrent execution pattern: a splitter box divides the image to be computed into a given number of slices. Each slice is then routed to one specific solver box for the actual raytracing. There is exactly one
Comparison between different implementations

random, thr = 0.5
ready-time, thr = 0.9
old
pthread

Figure 3. Raytracing runtimes for different placement strategies

Speedup

Figure 4. Raytracer scaling for different placement strategies

Rather than unfolding a high degree of concurrency and trusting on S-NET to efficiently and effectively map this down to the available compute resources, this (and many other) S-NET application actually manages the application-specific concurrency explicitly. This makes dynamic task migration largely obsolete as the static task distribution of the LPEL layer is silently anticipated by the design of the application and shows optimal results. To conclude the raytracer is not a good candidate to evaluate dynamic task migration, other than providing a quantification of potential negative performance impacts due to continuous observation of program execution.

Consequently, we proceeded to explore several other example S-NET applications with various thresholds to examine the performance impact of placement on less regular workloads. Two of the applications have been used as benchmarks in previous research, an ant colony optimization program[1] and an acoustic target tracker using the MTI-STAP algorithm[8]. In addition to these we used an example network generated by our automatic benchmark generator. All these experiments were done on a 48-core system with four 12-core AMD Opteron(tm) 6172 2.1GHz system with 128 GiB memory.
Figure 5. Ant colony results.

Figure 6. MTI-STAP results.
The results of these benchmarks are shown in Figure 5, Figure 6 and Figure 7. These graphs illustrate that placement is a very application specific problem. For the ant colony optimization there is no significant difference between no placement, smart placement and random placement. The MTI-STAP application, on the other hand, shows fairly big differences between execution times with placement and execution times without placement. With increasing worker counts random and smart placement both outperform the statically placed version. Lastly, our generated benchmark shows two different distributions of execution times. No placement and smart placement end up performing and scaling identically. The executions using random placement are all identical in performance and scaling too, but are slightly slower than those with ready time placement or without dynamic task migration.

While our graphs, unfortunately, do not show clear performance improvements through dynamic task migration, they do show that placement can positively affect the runtimes of applications, as illustrated by the results of the MTI-STAP application. It would be interesting to determine in what ways the MTI-STAP application differs from the ant colony and the generated benchmark. This would help us to realize which scenarios are amenable to dynamic placement scheduling and would allow us to target our placement strategies to these.

For example, one notable difference is that the MTI-STAP application does not use the parallel replication combinator. This combinator is usually used to give the application some control over the amount of concurrency to use, but could also function as throttle, limiting the amount of possible concurrency. However, drawing any in-depth conclusions on how placement relates to different applications requires much more exhaustive testing and analysis than we can give here, it is thus left as future work.

5. Conclusion

We presented the design and implementation of automatic task migration for S-NET through concerted extensions of the S-NET runtime system and the underlying LPEL threading layer. An asynchronous placement scheduler task continuously monitors workload distribution between cores. Depending on a migration oracle the placement scheduler, asynchronously to all other runtime activities, decides to migrate a task from its current worker/core to another presumably less loaded worker/core or not. This decision only becomes effective as soon as an S-NET streaming component processes its next input, upon which it will be respawned on a different worker/core if the placement scheduler made this decision in the mean time.

We experimented with two concrete placement scheduler implementations. One simply migrates tasks with a certain probability and does not actually monitor any dynamic behaviour. The other, more elaborate placement scheduler implementation monitors average waiting times per worker and migrates tasks from workers with a long average waiting time to workers with shorter average waiting time.

Extensive experiments with a range of applications show mixed results. We did manage to reduce the overhead inflicted by the placement scheduler, in particular, for monitoring the dynamic behaviour of an S-NET application after adverse experiences with an initial implementation. So, we can conclude that task migration does not adversely affect the runtime performance of S-NET programs.

However, none of the S-NET applications available to us for experimentation demonstrated task migration as a killer feature either. Our explanation for this unexpected and somewhat surprising observation is that all existing S-NET applications were written with the knowledge that task migration or similar dynamic load balancing and adaptation mechanisms were missing. Quite typi-
cally, S-NET applications rather manage concurrency themselves. Prominent examples are essentially data parallel problems such as raytracing. Such problems are approached with explicit domain decomposition done by some box and explicit work distribution through a parallel replication combinator. We can observe this pattern also in the ant colony optimization application.

An application that deviates from domain decomposition is the MTI-STAP application, which implements a classical signal processing pipeline. Here, we can indeed observe a positive impact of our ready-queue placement scheduler and task migration. Unfortunately, this application does not scale well due to other limitations in the combination of S-NET runtime system and LPEL threading layer.

We are, thus, confident that automatic task migration will, in the long run, prove useful by making S-NET application more robust against particularly unfortunate scheduling decisions.

Acknowledgements
The work has been funded by the EU FP-7 project ADVANCE (Asynchronous and Dynamic Virtualisation through performance ANalysis to support Concurrency Engineering, project no. 248828).

References
Fault-tolerant Coordination of S-Net
Stream-processing Networks ∗

A Feasibility Study

Raimund Kirner, Vicent Sanz-Marco, Michael Zolda, Frank Penczek
University of Hertfordshire, School of Computer Science, Hatfield, United Kingdom
{r.kirner, v.san-marco, m.zolda, f.penczek}@herts.ac.uk

Abstract
Fault tolerance and robustness are system properties of increasing importance, both in the domain of embedded computing as well as in the domain of high-performance computing.

In this paper we study the applicability of fault-tolerance mechanisms in the context of stream-processing networks, in particular based on the coordination-language S-Net. We identify three basic fault tolerance mechanisms and discuss the technical solutions for them within S-Net. The applicability of these mechanisms depends on the requirement of the concrete application domain. The contribution of this paper is a feasibility study of tool-supported fault tolerance mechanisms in a flexible coordination language allowing for asynchronous execution. As part of this feasibility study we discuss potential extensions of the S-Net language and runtime system in order to implement the identified solutions.

General Terms concurrent computing methodologies, parallel computing methodologies, reliability

Keywords dependability, fault-tolerance, robustness, software engineering, coordination languages, stream processing, data-flow, S-Net

1. Introduction
Computing systems in the embedded domain as well as in the high-performance domain share some common challenges. Arguably the most important one of them is fault tolerance, either to protect human lives, or to improve the performability of a system. At the same time, stream-processing computing is a popular pattern in both domains for its benefits in multi-core programming as streams can be used as a form of implicit synchronisation. Implicit synchronisation is a powerful tool to cope with the inherent complexity of concurrent programming.

Fault tolerance has been studied thoroughly in the field of dependable real-time systems [12]. A classical way to achieve fault tolerance is the use of triple modular redundancy, which is based on voting over redundant computations. With the increasing complexity and fault likelihood of computer systems, additional fault-tolerance techniques are useful to extend the performability of the system. For example, reconfiguration can be used to increase the dependability of core services, which is especially useful in mixed-criticality systems [10].

Fault tolerance in stream-processing systems is an important issue, as stream-processing applications is a frequent application pattern that is well-suited for programming parallel systems. In stream-processing systems with an explicit coordination layer the compiler can use structural information about the application to generate code to support fault tolerance.

For example, Balazinska et al. provide fault tolerance for distributed stream processing, using checkpoint/restore [5] approaches [2]. This approach is based on redundant computation for fault-tolerance with support for reintegration after a fault, but not reconfiguration.

In this paper we study how different fault-tolerance strategies are applicable to coordination-based stream processing in a flexible execution model, namely that of S-Net [8], which allows for asynchronous message processing. The language S-Net is introduced in Section 2. In Section 3 we describe how to realise different fault-tolerance techniques based on three basic mechanisms. Section 4 discusses related work. Section 5 concludes this document.

2. Key-Concepts of S-Net
The approach taken by S-Net is targeted at stream processing. S-Net is a very compact, powerful and declarative coordination language for describing streaming networks of asynchronous components. It reflects the modern notions of subtyping, encapsulation and inheritance, while completely separating all communication and concurrency concerns from the application code.

Streaming networks are defined using an expression language featuring four network combinators as operators: serial composition, parallel composition, serial replication and parallel replication. With the exception of serial composition, the combinators come in two flavours each: the deterministic versions preserve the order of data on streams, whereas non-deterministic variants trade this property for improved throughput. Two primitive components serve housekeeping and synchronisation purposes. Streams are associated with record types: collections of data where each item is uniquely identified by its name. Structural subtyping on records directs the flow of data through the streaming network.

Instead, the language is built upon data-flow principles: an application is a collection of computational components, also called boxes, that are put in relation to each other only by their data dependencies. To express such dependencies, the language provides means to construct a hierarchical data-flow graph. In this graph the

---

Footnote:

∗ This work has received funding from FP7 STREP project no. 248828 (ADVANCE) and from ARTEMIS-JU project no. 295371 (CRAFTERS).
boxes are forming the set of nodes, the data dependencies between boxes are captured by the set of edges. A data dependency between two boxes is expressed by placing an edge between them. An intuitive interpretation of the data-flow graph is that of a communication infrastructure between computational components: The edges of the graph are directed channels through which components communicate with each other when they need to exchange data. The program defines a streaming network in which data items are received and processed by a box before the result is sent on to the next adjacent box. In an S-Net program, the implementation of a box is not accessible, i.e., boxes are opaque components that may be implemented in a range of conventional programming languages, as for example C.

Within an S-Net program we have no means to inspect the data that a box receives and produces; this is an important consequence of the freedom of choice of a computational language for the implementation of boxes. The data types of the computational language are never exposed to the outside. Yet, we need a handle on data from the box domain in order to receive results from boxes, route them through the data flow network and supply them as arguments to other boxes. For this, S-Net employs its own, record-based notion of types. Data that is received from a box is handled as a set of label-value pairs, i.e., a record, and it is assigned a record type based on the type signature that is supplied with a box. The organisation of data as records allows us to refer to data symbolically by a label while keeping the value, i.e., the data of the box domain, opaque. Even more importantly, types play a fundamental role in the way records are routed through the network. At split points the type of a record determines the choice for the branch the record takes, i.e., at runtime the communication between different parts of the network is established through type-directed routing.

As an illustration, consider a generic fork-join pattern that applies a computation in parallel to an input. An S-Net program for this example may look similar to what is shown in Figure 1. In this representation, boxes that require results from other boxes are located to the right of the component they depend on. Boxes that can execute independently of each other are vertically aligned. In this program, a `split` box that decomposes an input into several smaller chunks is the first component to execute. This is followed by several `compute` boxes that may execute in parallel, implementing the actual computations that we are interested in. After the execution of the computations finishes and the result is output, the results are collected by a sub-network `fold`, to collect and output the overall result.

The core language of S-Net that allows us to express coordination programs like the one above is very compact. The atomic building blocks of programs, as mentioned earlier, are boxes. These represent computations without revealing their actual implementations. Only an abstract type signature, similar to a function prototype in a C header file, is exposed to the S-Net program that describes the data flow network within which the boxes are instantiated. As S-Net programs are based on data-flow graph modelling, the essential ingredients of the language are combinators for graph construction. Only four combinators together with one primitive for explicit synchronisation suffice to express elaborate coordination programs. Two combinators allow for static sequential and parallel composition. Two further combinators define dynamically unfolding sequences and dynamically expanding parallel compositions. By allowing hierarchical graphs, i.e., graphs in which nodes are graphs again, even complex coordination programs can be expressed in a structured and intelligible way. The design of the language does not rely on a specific execution model, but is built on the assumption that all boxes within a graph may execute asynchronously as soon as input becomes available. Boxes strictly consume a single data item per invocation, but are allowed to output any number of data items as long as boxes return from their computations eventually, i.e., boxes may not implement infinite producers. Furthermore, boxes are not allowed to maintain an internal state across invocations. The latter requirement enables us to very cheaply re-instantiate and duplicate boxes, i.e., computational tasks, on different resources: S-Net allows for explicit task-to-resource mappings on the level of boxes and networks using a `@R` notation where `R` is a resource identifier. Depending on the underlying executing machinery, the resource identifier may, for example, refer to a specific node in a cluster, a processor in a multi-processor system or a specific core in a SoC.

From a software-engineering perspective it is worth noting that S-Net fulfils the six principles of modular software construction [4]:

- **Information Hiding:** The user doesn’t need to know the internal mechanism of the boxes to use it. In the example of Figure 2, the user doesn’t know the procedure inside the boxes. Remember, the implementation of a box is not accessible.
- **Invariant Behaviour:** The same box may be used at different locations in the network. These boxes have the same behaviour regardless of their location in the network.
- **Data Generality:** The interface to a box is capable of passing any messages an application may require.
- **Secure Arguments:** The boxes do not exert side-effects on their inputs.
- **Recursive Construction:** Any S-Net network constructed with boxes, can be used as a subnetwork within an S-Net application.
- **System Resource Management:** Resource allocation is performed by the runtime system and not by boxes.

Figure 2 shows an example of an S-Net program [7]. In the example, we can find four different kinds of boxes: `leq`, `if`, `dec` and `mult`. They reflect the basic building blocks of the functional implementation of factorial: The box `leq` computes the termination condition; the result is stored in field `p`. The box `if` makes the boolean value of the field `p` visible to S-Net by choosing it either into a tag `<T>` or a tag `<F>`. Last but not least, the boxes `dec` and `mult` do the required arithmetic. Also, in the example, there are two more boxes: `filter` and `sync`.

The primitive `filter` box is devoted to all kinds of housekeeping operation. Effectively, any operation that does not require knowledge of field values can be expressed by this versatile primitive box in a simpler and more elegant way than using an atomic box and a box language implementation.

The synchronisation cell, or synchronocell for short, is the only stateful box in S-Net. Syntactically, a pattern merely is a record type. The principle idea behind the synchronocell is that it keeps incoming records which match one of the patterns until all patterns have been matched. Only then the records are merged into a single
one that is released to the output stream. Matching here means that type of the record is a subtype of the pattern. The pattern acts as an input type specification of the synchronecell: a synchronecell only accepts records that match at least one of the patterns.

This example shows how concepts of functional programming (e.g., nested function definitions, function applications) can be expressed in the framework of S-Net in a systematic way. The example in particular serves as blueprint for expressing linear recursive functions in S-Net. In the factorial example the box language code is extremely simple, one atomic instruction each. However, without changing the principles of the S-Net we could replace the box inscriptions by complex computations with record fields referring to large data structures. As long as the algorithmic pattern remains the same, we can easily turn a toy example like factorial into a real application. Leaving the concrete example behind, our example sketches out a methodology to convert functional programs into S-Net in order to express and to exploit concurrency.

3. Fault-Tolerance of Stream-Processing Networks

It is the property that enables a system to continue operating adequately to a hardware or software failure. The fault tolerance is very important in systems that must work all the time. Given a failure, another component or a special procedure can take control to remedy or mitigate the effects of the error.

A fault-tolerant system may be able to tolerate one or more fault-types including:

1. hardware faults,
2. software errors,
3. externally induced upsets or physical damage.

An extensive methodology has been developed in this field over the past thirty years, and a number of fault-tolerant machines have been developed. A large amount of supporting research has been reported. Fault tolerance covers a wide spectrum of applications, for example, ranging across embedded real-time systems, commercial transaction systems, transportation systems, and military/space systems. These areas often involve widely diverse core expertise ranging from formal logic, mathematics of stochastic modeling, graph theory, hardware design and software engineering.

It’s important don’t confuse fault tolerance with system maintenance. Because the system maintenance requires an external agent to work when there is any problem. On the other hand, we speak of fault tolerance when an error occurs in the system and the system can handle that error without any exterior help.

Fault tolerance for stream-processing networks in principle is not different from generic fault-tolerance precautions. The challenge still is to provide an acceptable level of service even in the presence of faults. However, in our aim to cover embedded computing as well as high-performance computing domains, we identify the following properties that influence the design criteria of adequate fault tolerance mechanisms:

1. Fault tolerance is transparent to the box level, i.e., fault tolerance is dealt with at the level of the coordination language S-Net and its runtime system
2. Mixed Criticality: The fault tolerance precautions do not necessarily have to be applied to the whole S-Net program, i.e., the developer has fine-grained control over the operators for which fault tolerance is needed
3. We aim to handle both, permanent and transient faults [1]
4. We provide multiple fault-tolerance precautions to provide adequate techniques for the high-performance domain, i.e., average execution time is the main metric, as well as for the embedded domain, i.e., worst-case execution time is the main metric for the real-time constraints
5. Fault tolerance should be available for faults in the time domain (includes fail silent faults) and in the value domain (for the computation of messages as well as for their communication)

In previous work we already investigated how a compiler can simplify the programming of safety-critical applications by keeping the fault-tolerance mechanisms orthogonal to the application programming [11]. However, the analysis in [11] was focusing on a compiler for a functional programming language. While functional programming languages already provide benefits, like the ability to achieve a significant reduction of state size for checkpointing [3], we see further potential when applying them to a coordination language like S-Net. First of all, the boxes in S-Net are free from persistent state, like a functional program would be. But further, a streaming connection between boxes provides a natural choice for fault-tolerance mechanisms like checkpointing, as the state size is relatively small. In addition to the fault tolerance requirements listed above, we aim to support the following software behaviour imposed by the S-Net language:

6. S-Net boxes are functional, i.e., they always produce the same output for the same input. This feature simplifies fault tolerance as re-computation for an input will still produce the same value as it would have in the first attempt.
7. S-Net networks and subnetworks can be constructed in such a way, that they have the MIMO property (multiple input stream, multiple output stream). Based on this property we can use a logical time for messages ordering instead of having to use real-time with the overhead of clock synchronisation.
8. An S-Net box may have a multiplicity larger than one, i.e., it may produces more than one output message from one input message. This raises consistency issues in the context of fault tolerance, as boxes may get faulty after only part of the output has been written.
9. Asynchrony: Except for the special case of where order is enforced, messages in S-Net may be processed out of order.
10. Nondeterminism: Routing of messages can be nondeterministic, i.e., the routing is not defined at S-Net language level. This behaviour is meant for fairness-based routing, but causes additional consistency challenges for fault-tolerance mechanisms.

There are several fault-tolerance techniques we aim to support. However, these techniques will be based on one or more basic mechanisms. We have identified the following basic mechanisms to be added to S-Net to support fault-tolerance:

- Checkpointing/restoring of program state
- Dynamic Reconfiguration
- Redundant Computation

These mechanisms are building blocks for the realisation of different fault tolerance techniques as listed by Avizienis et al. [1]. The meaning of these mechanisms and their contribution to fault tolerance is discussed in the following subsections.

3.1 Checkpointing/Restoring of Program State

A very basic mechanism for fault tolerance is to checkpoint the system state regularly and in case a fault has been detected just reload the state of the latest checkpoint and retry it again. The main overhead in checkpointing in modern systems stems from storing the associated state [5]. Techniques like uncoordinated checkpointing help to reduce this overhead [6]. In case of a fault, doing a restore
Within the context of S-Net, the major challenge for fault tolerance mechanisms is Property 8 (multiplicity). Because if a box produces just one out of two messages and then fails, one cannot just restart the box with the original input on another resource. Only in special cases where messages are always meant to be state messages such a simple restart would be tolerable. In general, there is no guaranteed boundary for the impact of the first produced message when the restart takes places. Thus the only way to cope with this problem is to remember the messages already been sent and to filter out those messages on a restart.

To explain the realisation of such a message filter for fault tolerance with checkpoint/restore, we assume that checkpoint/restore regions (FT-CR) are specified over subnetworks as shown in Figure 3. The input stream can be seen as a simple FIFO (first in, first out) queue. But the output having been passed through the covered subnetwork has to be temporarily remembered in a two-dimensional buffer. For each input message \( i_k \) the S-Net subnetwork may produce \( m_{k,i} \) output messages: \( o_{k,0} \ldots o_{k,m_{k,i}-1} \). The FT-CR region will forward any output message \( o_{k,j} \), but will keep a local flag indicating which messages have been seen. For each input \( i_k \), the passed-through flags of \( o_{k,j} \) have to be stored until all \( m_{k,i} \) output messages have been passed through. When this has happened, it is assured that the input \( i_k \) has been completely processed and any output has been passed over to another FT-CR region, so that there will be no need to ever restart the execution of the subnetwork with input \( i_k \). At this point all pass-through flags of the messages \( o_{k,i} \) can be discarded.

The indication that all output messages \( o_{k,i} \) have been passed will require the knowledge of the multiplicity \( m_{k,i} \), which can be calculated at runtime through control records.

The passed-through flag buffer will store the unique id of messages, where the computation of the id of an output message solely depends on the id of the input message. This two-dimensional buffer is typically implemented as a vector of linked lists, as the multiplicity for each box can be different, resulting in a sparse occupation of the buffer.

3.2 Dynamic Reconfiguration

Dynamic reconfiguration in general allows to change the configuration of a stream-processing subnetwork or even change the subnetwork itself.

The reasons to reconfigure a network may be manifold: A box has to be replaced, because an improved version of an algorithm a box is currently running became available. A network suffers from poor throughput, because of imbalanced execution times and bottlenecks, or a box has crashed and does not process any records anymore. Reconfiguration is key to maintain, improve or restore system stability and performance, as well as it is a means to implement fault-tolerance mechanisms in general.

Providing separate mechanisms to address any of such issues in particular is possible, but in the context of S-Net we are developing a more general approach that is directly integrated into the language. Following the design-principles of the language, a combinator-based approach is proposed. In fact, a total of just two combinators prove to be versatile enough to natively support the implementation of various fault-tolerance mechanisms. This two combinators are replacement combinator and feed-back combinator.

Replacement combinator.

As with the other combinators, the replacement combinator may be applied to any network. By applying it to a network, the network is made replaceable. The combinator may be applied to entire sub-networks such that it is not the network that replaces itself but it is the combinator that replaces its operand. The surrounding replacement construct, i.e., the combinator, remains as a static component within the surrounding network. New operands are sent to replacement combinators using the existing communication infrastructure that networks provides for stan-
dard data processing. For this to be integrated seamlessly, we promote networks to first-class citizens of the language: A network may be part of a message, just like other data. This way the deployment of new networks follows the same routing principles as plain data. Messages that contain new network operands are routed on the same basis as before, i.e., routing is determined by the name and the type of the network operand, again, just as is the case with other data. When a message arrives at a replacement combinator, the message may be treated in two different ways. If the messages contains a network, i.e. a potential operand for the combinator, this network will be inspected. If it is compatible to the currently deployed operand, the replacement combinator reads and removes the network from the message. The combinator replaces its current operand by the new network and discards its old operand. If the message contains an incompatible network or no network at all, the message is sent to the operand of the replacement combinator. In this case, the message is processed as usual.

Feed-back combinator.

The replacement combinator allows for reconfiguration to be triggered by a network component that lies upstream of it or from the global input of the network, i.e., triggered from the outside. However, it is often desirable to monitor the results of a network and then make a decision on if or which kind of reconfiguration is required. In order to allow for the back-propagation of collected information, a feed-back combinator complements the replacement mechanism. The feed-back combinator introduces a back-edge in the streaming network and allows for a communication of a component with another component that lies further upstream.

The combination of these two combinators, i.e., a reconfiguration combinator as the operand to a feed-back combinator, is very versatile as it allows for the implementation of self-adaptation: By collecting runtime information of the operand of a replacement combinator and then acting on this information once a result exits the combinator’s operand network, we may make a decision on how the network should be reconfigured. A reconfigured operand may then be sent back through the feed-back edge to the replacement combinator that is under observation. Dynamic Reconfiguration can be used handle different kinds of faults:

**Hardware Faults:** When some resource doesn’t work correctly, then it needs to be changed. One possible strategy is to use resource reallocation, and change the bad resource for another good resource. To do this, Dynamic Reconfiguration knows about all the resource in the system and must choose a spare resource to replace a broken one. Its possible to see one example in Figure 4. In the figure, such as dynamic reconfiguration observed the network. When a problem occurs, inform the relevant software (runtime monitor) and the software acts accordingly. In the example, resource $R1$ is broken. Then dynamic reconfiguration does not catch the error. The error is recognized by the runtime monitor and dynamic reconfiguration is used to remove the fault.

**Software Faults:** The system is affected by a software fault. The author of this software can find the problem and change this implementation. The basic building blocks of an S-Nets are boxes, and we only need to replace the box with the error with another box with alternative implementation. Dynamic Reconfiguration is a method for replacing a bad box with another box. (Figure 5)

**Timing Faults:** When there is a significant delay in the execution time and the program needs to finish before a determinate time

![Figure 4. Example hardware fault.](image-url)
Figure 5. Example software fault.

Figure 6. T3 cannot complete within deadline because resource is busy.

Figure 7. Dynamic resource allocation with timing fault.

Figure 8. Example channel fault.

Figure 9. Timestamps on data messages can be used to trigger automatic reconfiguration of a system.

decide if the processing time lies within acceptable boundaries. If this isn’t the case, the box outputs a new resource mapping that is sent back to the beginning of the processing network through a feed-back stream (the back-edge that is seen on top of Figure 9; this is achieved by applying the feed-back combinator to the processing network).

If the measured processing time lies within acceptable bounds the trigger box strips the time-stamps from the result before forwarding it, without producing a new resource mapping.

Using this and similar techniques, other reconfiguration and adaptation strategies may be implemented such as automatic restarting of services using heartbeats, dynamic recompilation and optimisations of box code and re-balancing load on computational clusters, and also fault tolerance by remapping resources for critical tasks in a mixed-criticality system in case of partial loss of resources due to a fault.

3.3 Redundant Computation

Besides re-computation, another fundamental pattern of fault tolerance is redundant computation. In many applications, redundancy is extensively used strategy to avoid misunderstandings or decoding errors. Descriptively, redundancy is strategic communication factor consists in intensifying and repeat the information contained in the result, even without a fault occurring. Redundancy can be used in several different ways for fault tolerance:

Fault detection uses redundancy in order to obtain inconsistencies of the produced results in case of a fault.
 Fault correction means fault tolerance without the need of re-
computation after a fault has been detected. Fault correction has to be used in real-time systems if re-computation would not be possible without risking a deadline violation.

Assuming a deterministic system, to tolerate the case where a subsystem shows one fault failure (incomplete but no wrong output produced) [1], redundant computation with only two instantiations of the computation is sufficient to correct this silent failure of the subsystem.

In case the subsystem fails with erratic failures [1], a majority voting over several instantiations of the computation has to be used to tolerate this fault without the need for restore and re-computation. For example, to tolerate one failure either in the communication system or in the computational units, triple-modular redundancy (TMR) can be used [12].

Redundancy in S-Net

Regardless of using redundancy for fault detection or fault correction in S-Net, the results of the different outputs within the redundancy region have to be held back and buffered until a conclusion can be drawn.

Figure 10 shows an S-Net subnetwork that is surrounded by a redundancy region FT-RED(n) meaning that n instances of the subnetwork will be created. Again, the input stream can be seen as a simple FIFO (first in, first out) queue. But in contrast to the buffer of checkpoint/restore as described in Section 3.1, the output messages of the covered subnetwork have to be temporarily stored and held back in a three-dimensional buffer. For each input message \( i_k \) each of the \( n \) redundant S-Net subnetworks may produce \( m_k \) output messages: \( o_0 \ldots o_{m_k-1} \), stored separately for each of the \( n \) redundancy instances. The FT-RED(n) region will output a message only when enough of redundant outputs have been produced. "Enough" in this context is dependent on the fault-tolerance technique to be realised with the redundancy, e.g., a) for fault detection with two instances of the subnetwork, all two messages have to arrive before it can be forwarded, and b) for TMR at least two of the redundant outputs have to be produced (in the absence of faults) before it can be forwarded. Note that the holding back and storing of output is only for redundancy purposes, i.e., the \( m_k \) individual messages resulting from multiplicity if the subnetwork are not needed to be held together. The multiplicity is either ignored (e.g., for fault-masking with TMR) or is handled in the same way as described in Section 3.1 for checkpoint/restore (e.g., when combining fault-detection based on redundancy with checkpoint/restore).

We finally have to note that above discussion is slightly simplified, as we only want to show principal feasibility of fault-tolerance with S-Net based on redundancy. In practical realisations it is slightly more complicated. For example, when using TMR regions, the output of each redundant subnetwork instance of a FT-RED(n) region has to be forwarded to each of \( n \) subnetwork instances of the succeeding FT-RED(n) region [12]. Without such an individual connection between the subnetworks of two adjacent FT-RED(n) regions the voting would be a single point of failure.

4. Related Work

Software-controlled fault tolerance is a broad field of research. For example, Elnozahy et al. give a good overview of different rollback-recovery protocols in message-passing systems [5]. The authors also conclude that the main overhead of checkpointing in modern systems stems from storing the associated state. Choi et al. discuss compiler support for automatic checkpointing in general [3]. The frequency of checkpointing with general programming languages is limited due to this high overhead.

Therefore it is beneficial to minimise the state that is to be stored. Rich type systems as they are commonly found in functional programming languages can be of great value in this context. By exploiting structural information on heap objects derived from their types, the size of the checkpoint data can easily be reduced by up to 70% [3]. Functional programming languages here have the key advantage that due to there call-by-value semantics. The call-by-value semantics ensures the arguments to functions are immutable. Further, side effects would need to be explicitly modelled. This allows an effective containment of function state in a functional setting. Only the arguments to a function are required to restart its computation and the function itself does not modify any global state. Therefore, support for reversing and repeating a computation in a compiler for functional languages is simple to implement compared to the imperative setting [9]. Kirner et al. discussed several techniques for software-controlled fault tolerance in a functional setting [11].

S-Net is not a strictly functional language, as there is state on the network, e.g., in buffers of streams or in case of explicit synchronisation with so-called Synchro-cells. But boxes in S-Net have a functional setting, i.e., there is no persistent state inside the boxes. Thus, it is expected to be able to take advantage of this property for S-Net, similar to what has been shown for functional languages.

5. Conclusion

Development of fault-tolerant systems has its strong need in the domain of embedded computing as well as high-performance computing, although these two domains have apparently different application requirements. Within this paper we have studied similarities of fault tolerance for these two application domains.

The study has been based on S-Net, a coordination language for stream-processing systems. As a result we have shown that fault-tolerance mechanisms from the safety-critical embedded domain can be supported by a tool chain for S-Net. This is remarkable, as S-Net, which has been originally developed for the high-performance domain, has a rather flexible execution model that supports asynchronous computation and processing of partial output of its components.

As future work we will develop the control protocol needed to trigger the fault-tolerance actions and implement the approach.

Acknowledgement

The research leading to these results has received funding from the IST FP7 research project “Asynchronous and Dynamic Virtualization through performance ANalysis to support Concurrency Engineering (ADVANCE)”, and the ARTEMIS-JU research project “ConstRaInt and Application driven Framework for Tailoring Embedded Real-time Systems (CRAFTERS)".

Figure 10. Three-dimensional message buffer for Redundant Computation.
References


