Project no. 248828

ADVANCE

Strategic Research Partnership (STREP)
ASYNCHRONOUS AND **D**YNAMIC **V**IRTUALISATION THROUGH PERFORMANCE
ANALYSIS TO SUPPORT **C**ONCURRENCY **E**NGINEERING

Observation of resource utilisation
D17

Due date of deliverable: Nov. 30, 2011
Actual submission date: Jan 25, 2012

Start date of project: February 1st, 2010

Type: Deliverable
WP number: WP6
Task number: WP6c

Responsible institution: TWENTE
Editor & and editor's address: Jan Kuper
University of Twente
Department of EEMCS
7500 AN Enschede, The Netherlands

Version 1.0 / Last edited by Robert de Groote / Jan. 24, 2012

Project co-funded by the European Commission within the Seventh Framework Programme		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Revision history:

Version	Date	Authors	Institution	Section affected, comments
0.0	17/10/2011	Robert de Groote	TWENTE	Initial version
1.0	24/01/2012	Robert de Groote	TWENTE	Final version

Reviewers:
Jan Kuper

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
WP6c	Observation, analysis and evaluation	TWENTE*, UvA, USTAN

°This task list may not be equivalent to the list of partners contributing as authors to the deliverable
*Task leader

Executive Summary

This document describes methods to observe and measure performance data of a chosen placement of an application's implementation. These performance observations serve as input for the development of statistical resource usage models in work package 4.

Performance data are gathered using the monitoring function of the execution layer of S-Net. An example application (one of the use-cases described in work package 7) is used to illustrate how different data may be obtained from the monitoring function.

Contents

Chapter 1

Introduction

Resource optimisation of an application is a continuous process. At compile-time, an initial placement of an application may be chosen by following the approaches mentioned in [5], [9], [4] or [8]. After the initial placement, the application's resource may be further optimised by identifying key bottlenecks. In order to identity these bottlenecks, it is important to know performance data of the current implementation and placement of the application.

Finding performance bottlenecks in the system requires relating the performance metrics of a (group of) component to the performance of the system the component is part of. This involves aggregation and composition of statistical properties. Classical analysis techniques used in the realm of real-time systems fall short in providing the tools necessary to perform this aggregation. One of the goals of the ADVANCE project therefore is to present new analysis techniques that can cope with statistical properties.

At the base of these novel techniques are the statistical resource usage models, which are in developed in work package 4. In order to build statistical models of resource usage, performance metrics need to be observed. This report describes which performance metrics need to be observed, and how they will be observed. These observations are performed by the *resource management* component in the *Advance vision* depicted in figure 1.1.

Performance data include many different metrics such as execution time, la-

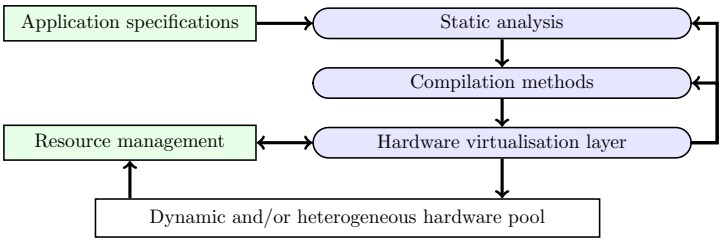

Figure 1.1: The Advance vision

tency, jitter, throughput, congestion, buffer loads, load balance, power consumption, etc. Some of these metrics may be derived (i.e. estimaed or calculated) from other, *basic* metrics. Jitter, for example, is not a basic metric as it is the variance of the distribution of latency, which is a basic metric. We distinguish between these *basic* metrics and *performance* metrics. The latter may be used as terms in a cost function that steers the software-to-hardware mapping optimisation process.

Throughout this report, an application derived from a use-case is used as a running example. We use this application to show how basic and performance metrics may be derived. We define 3 key performance metrics, all of which are based on the single basic *latency* metric. The application is illustrated in figure 1.2 and described below.

1.1 Example application

Figure 1.2 shows the S-Net model of the X-ray image processing application that implements one of the use-cases from work package 7. A more detailed description of the reference application can be found in [1]. The application involves two image processing pipelines that run in parallel. The first of these is an *image enhancement process*, where noise is removed from the image by applying a gaussian blur with a convolution kernel of size 9. The runtime of this process depends only on the size of the input image. As this size is kept constant (an image consists of 1000×1000 pixels with 8 bits per pixel), runtime of the image enhancement process is totally independent on any features of the data.

The second image processing pipeline (i.e., the *object tracking* box) is responsible for tracking a feature in the image. In reality, this feature may be a surgical device such as a *stent*, which may move due to the patient changing position. The feature tracking task remembers the last known location of the object and uses this location as the starting point for the search. Search is limited to a small area around the last-known location of the object, which is expanded in case the object was not found. Because most of the time the object moves only slightly, search will yield the new location of the object quite fast. From time to time, however, the object makes a larger "jump", causing the object tracking box to temporarily require more time to execute the search. In contrast with the image enhancement process, the execution time of the object tracking task *is* data-dependent.

1.2 Outline

The outline of this report is as follows: Chapter 2 will give an inventory of performance metrics and select a subset that is necessary to characterise the application's performance under the current mapping. Chapter 3 will discuss how the chosen metrics may be observed. The report concludes with chapter 4, which will give directions for future work.

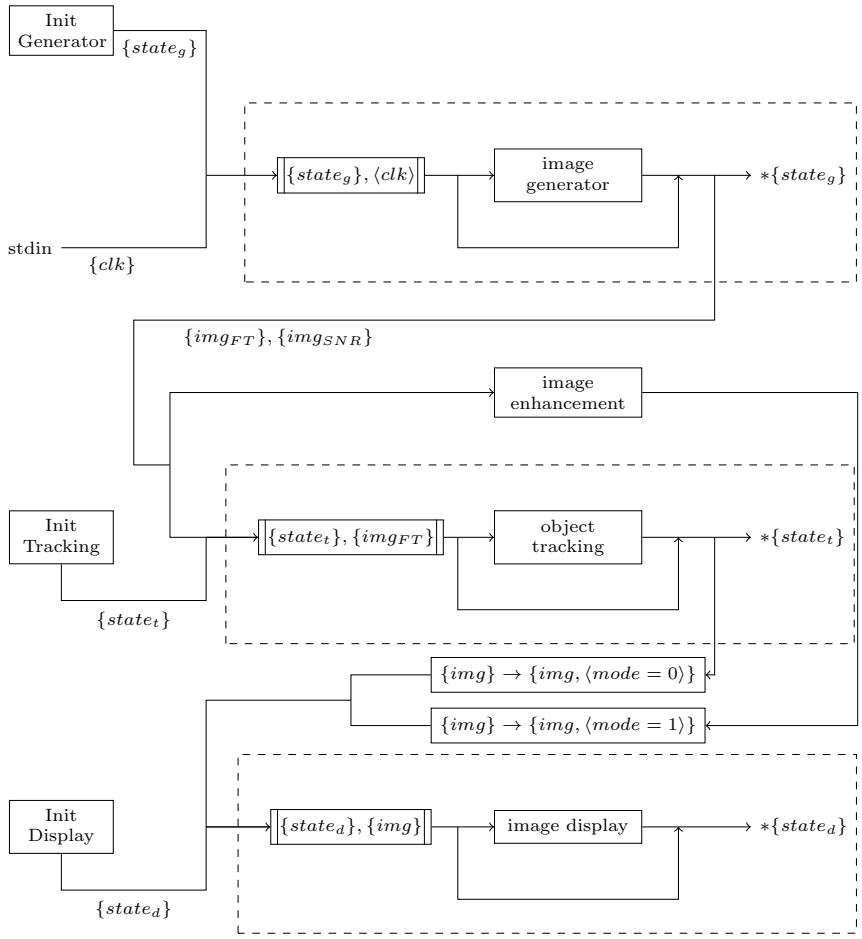

Figure 1.2: S-Net of the Philips X-ray image processing prototype application

Chapter 2

Performance Metrics

The performance of an application under a given placement can be expressed in many different ways. Quality-of-service constraints are less restrictive than the performance constraints found in traditional real-time critical systems. The major goal within Advance is to optimise the *utilisation of available resources offered by the hardware platform*. We capture this goal into a cost function that takes utilisation and communication overhead as primary factors.

2.1 Resource utilisation

The main motivation for using resource utilisation as a cost function to steer an optimising process towards well-balanced hardware usage is rooted in literature. In [4] and [9], the authors use *fragmentation* and *communication distance* as terms in a cost function for finding an optimal mapping of a streaming application to a heterogeneous *multiprocessor on chip* (MPSoC) system. As this report considers shared-memory multicore machines, these two terms need to be translated somehow. Fragmentation in an MPSoC captures the fact that processing elements may become unavailable for future tasks due to communication channels being cut off. Fragmentation leads to parts of the system not being used and decreases the maximum utilisation that may be obtained. As *physical* fragmentation does not occur in a shared-memory multicore system, we choose to replace fragmentation with the well-known metric of *utilisation* (see [2] for details). Utilisation expresses the extent to which computing resources (i.e. individual cores in a multicore system) are utilised in a number between 0 and 1. Badly chosen mappings (task-to-core assignment) may, like fragmentation in MPSoCs, lead to a waste of computational resources.

Another cost-metric used in [4] and [9] is *communication distance*, which is simply the number of hops of a series of communication links in an MPSoC. As in a shared-memory multicore system communication is implemented by writing and reading messages to and from memory, the metric is not directly adoptable for our setting. We therefore substitute communication *distance* with *time* needed by

individual messages to *travel* from the producing task to its consuming task.

The following sections provide further details on the performance metrics. Throughout the remainder of this document, we use $\mathcal{E}$ to denote the set of processing elements (i.e. cores in a multicore system), $\mathcal{M}$ to denote the set of all messages sent during the runtime of the application and $\mathcal{S}$ to denote the set of all synchronising tasks (i.e. tasks representing synchrocells in an S-Net).

2.1.1 Utilisation

Processing elements execute an application's tasks. Depending on the structure of the application's task graph and the assignment of tasks to processing elements, some processing elements may be busier than others. This may be due to the fact that a processing element has run out of tasks to execute, or may arise when all tasks assigned to a processing element are waiting for input to be produced by tasks assigned to other processing elements.

In case work is not evenly distributed over processing elements, these processing resources are not fully utilised. Depending on the objective, a resource manager may choose to lower the frequency at which tasks are executed for the under-utilised processing elements, or may change the schedule and mapping to achieve a higher, better balanced, utilisation.

Regardless the actions to take when a hardware platform is under-utilised, a clear measure for system utilisation is needed. We use the measure of *utilisation* [2]. For this, we define the total waiting time of a processing element as the total time spent waiting for input. Note that input may be either data to be consumed by assigned tasks, or tasks needing to be executed. If we define $T_{\text{wait}}(e)$ to be a processing element e's total time spent waiting for input and $T_{\text{active}}(e)$ the active time of processing element e, then the application's *load balance* or *utilisation* is defined as:

$$U = \frac{\sum_{e \in \mathcal{E}} \left(T_{\text{active}}(e) - T_{\text{wait}}(e) \right)}{\sum_{e \in \mathcal{E}} T_{\text{active}}(e)}$$

In other words, the system's utilisation is a weighted average of the utilisations $U(e)$ of individual elements e, with $U(e)$ defined as:

$$U(e) = 1 - \frac{T_{\text{wait}}}{T_{\text{active}}}$$

2.1.2 Communication overhead

Ideally, the time between the production of data by a given task and the consumption of that data by a subsequent task is close to zero. In reality, data needs to travel (e.g. through an on-chip or off-chip network) or is temporarily stored in a buffer. In an optimal setting, this communiation overhead is kept to a minimum and as such represents a component of the cost function of a software-to-hardware mapping.

Communication overhead can not be measured without taking computation into account: two computing and communicating tasks may spend most of their

time computing rather than communicating, in which case communication has little effect on the system's utilisation. For two communicating tasks that spend most of the time communicating rather than computing, minimising communication overhead has much more impact on the overall, system-level utilisation.

We therefore define the *communication overhead* C during the running time of the application as the ratio of the sum of the travelling times of all communicated messages, and the total computation time of a system, with:

$$C = \frac{\sum_{m \in \mathcal{M}} (t_I(m) - t_O(m))}{\sum_{e \in \mathcal{E}} (T_{\text{active}}(e) \cdot U(e))}$$

Here t_O and t_I are functions that associate a message with the times it was produced (*output*) and consumed (*input*), respectively.

2.1.3 Communication overhead due to synchronisation

A synchrocell merges multiple records into a single one. Since some records may be available earlier (i.e. they are produced earlier) than others, synchronisation may cause additional overhead. This overhead can be measured by examining the timestamps of messages consumed and produced by synchrocells. If $\mathcal{M}_{\text{in}}(s)$ is the set of messages consumed by a synchrocell task s, and $m_{\text{out}}(s)$ is the synchrocell's output message, and t is a function that assigns timestamps (i.e. time of creation) to messages, then a synchrocell's contribution to synchronisation overhead is:

$$C(s) = \max_{m \in \mathcal{M}_{\text{in}}(s)} t_O(m_{\text{out}}(s)) - t_I(m)$$

and the system's communication overhead due to synchronisation, denoted C_{sync}, is obtained by summing over all synchrocells:

$$C_{\text{sync}} = \sum_{s \in \mathcal{S}} C(s)$$

2.2 Basic metrics

A mapped application may use resources in an efficient manner. For example, it may use too much processing power to complete a computation, the result of which is not required to be available yet by consuming tasks. Buffers may be too small for intense data communication and cause congestion. Power consumption may be too high due to tasks being executed at a too high frequency, whereas a lower frequency would not impact the system's throughput. Workload may be unbalanced: one processing element is continuously executing tasks whereas other processing elements are idle or blocked on input from or output to the busy tasks. Tasks may be blocked on output to a busy communication link (congestion).

The primary metrics of interest in the Advance project are:

Latency the average delay between the receipt of data by a processing node and and the release of the processing results in the output channel(s).

Jitter the mean-squared variance of the delay time. This is a measure of variability of the processing time whose average value is the latency.

Throughput the amount of data passed through a node per time unit.

Note that in the absence of concurrency, throughput and latency are related. In the presence of concurrency, however, throughput and latency are independent.

2.2.1 Latency

A record needs time to travel through an S-Net. As the record travels through the network, it may be merged with other records in synchrocells, change its contents due to filter boxes or transform into other, multiple records as it enters and leaves a box. This dynamic life of a records makes recording the travelling time of a record difficult.

Latency is the time between the occurrence of two *corresponding* events due to a record. Events are typically the consumption or production of a record due to a box starting or completing its computation. As a single record may lead to the creation of multiple records, a single path in the network may yield different latencies.

Latency of a message through a (sub)network may be measured in different ways. Note that a single input message may lead to the creation of multiple output messages by a network. In that case the network is said to have a *positive gain*. We may distinguish between *earliest response* latency and *late response* latency. Earliest response latency is the time between the entry of a message in the (sub)network and the first exit of a corresponding message from the network. Similarly, late response latency is the time between entry of the message and exit of the last corresponding message.

Measuring latency involves keeping track of correspondence: messages produced by a box due to the consumption of a message correspond. In case of networks with a high *gain*, tracking correspondence may become a resource consuming task.

2.2.2 Jitter

Jitter is the variance of latency. Maintaining low jitter in the presence of varying execution times is a difficult task. As jitter essentially translates to unpredictability of resource usage, a fully utilised system may be highly sensitive to variations in component execution times or communication overhead. Jitter may be decreased by inserting buffers or explicit waiting times, sacrificing latency and / or throughput.

2.2.3 Throughput

The throughput of a box is the (average) number of executions of that box per unit of time. Throughput is related to latency if all boxes are executed in serial. Several executions of a single box may be run in parallel (i.e. through pipelining), in which case the throughput is increased without altering the latency.

Chapter 3

Observation of performance

In this chapter we describe the tools used to observe an application's performance under a given mapping. Examples are given based on running the example application described in chapter 1 on a multicore machine. Observations are currently processed offline through logfiles.

3.1 LPEL

The runtime management of S-Net is implemented by a Light-Weight Parallel Execution Layer (LPEL), which is described in [7]. In LPEL, each processing element in a multi-core system is assigned a *worker*. Workers disjointly manage a set of *tasks*. Workers themselves are scheduled by the operating system. Tasks are assigned to and scheduled by specific workers. Tasks are managed entirely in user-space within an operating system thread. As a result, switching between tasks does not involve expensive context switching to and from the operating system's kernel.

3.1.1 Workers

A worker is an operating system thread that is mapped to a single processing element (i.e. a single core in a multicore machine). Workers manage tasks (see next subsection) and the streams used to send or receive records to and from other tasks. Tasks are executed in a non-preemptive fashion; once a task is started it blocks the execution of other tasks scheduled to the same worker. A direct consequence of the bijective mapping from workers to cores, tasks assigned to different workers may be executed in parallel. However, dependencies between tasks may still restrict the two tasks from being executed in parallel due to synchronisation. Therefore, when assigning tasks to workers these dependencies must be taken into account.

3.1.2 Tasks

A task in LPEL represents a single-input single-output entity in S-Net. Such an entity may be one of the following:

Box Wraps a user-defined function. A box is mapped to either a single task or a set of tasks, depending on network combinators applied to the network in which the box is embedded: A box in a network that is serially replicated is mapped to one task for each instance of the serial replication. As a result, each time such a box is replicated the task-to-worker assignment is remade, with a potentially different outcome.

Collector A collector entity merges split subnetworks again to result in a single output stream. Collector entities co-exist with *split* entities (see below).

Star For serial replication. The star combinator (see [3]) allows particular records written out by networks to re-enter a replication of that same network. This allows, for instance, recursion to be modelled in S-Net.

Split Parallel replication is handled by the split entity. At split points of the network, there are entities with a single input and multiple outputs. For each split point, a collector entity is created that merges the subnetworks again to result in a single output stream.

Parallel Implements the router part of the parallel composition operator in S-Net. Since parallel composition is non-determnistic, the routing behaviour of this entity is determined by the types of input records and signatures of downstream networks. If the type of an input record matches only a single downstream network's signature, the record is routed to that network. If multiple matches exist, the entity may route the record to either.

3.1.3 Communication

Communication between tasks takes place by message passing. Each worker has its own mailbox, which consists of a message queue in which messages are enqueued by other workers and removed by the owning worker. Mailboxes are also used to notify workers to wake up dependent tasks. If a stream-operation within a certain task t causes another task u at another worker to become ready, the worker managing t sends a notification to the worker managing u. In order to keep mailbox synchronisation overhead low, LPEL uses a variation of the two-lock queue algorithm of Michael and Scott [6] is used (see [7] for details).

3.2 Performance observation

The LPEL subsystem exposes a monitoring function, which may be used to record several kinds of event information. Each of the tasks described in section 3.1.2

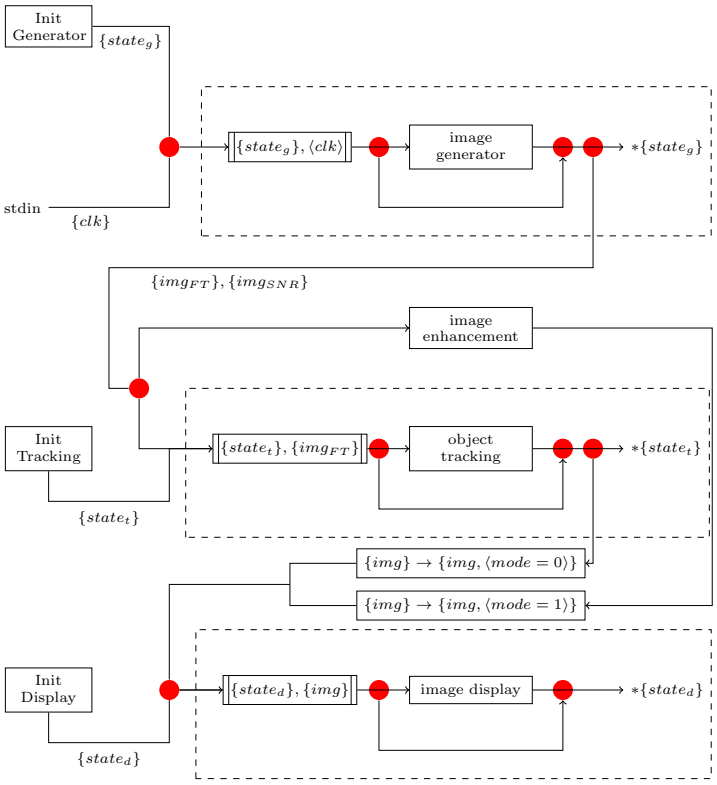

Figure 3.1: S-Net of example application. Red dots indicate LPEL tasks for which information is logged by the monotoring function.

can be subjected to the monitoring function. Figure 3.1 shows, with red dots, the topological location of the different LPEL tasks in the example application's S-Net. Note that some of the boxes and synchrocells are inside a star-network. This means that execution of these boxes and synchrocells will yield a new task each time the network is replicated. The same holds for the split and collect points of these star-networks.

Currently, the monitoring function writes its information to a logfile. The contents of these logfiles may be analysed offline for evaluation of the applied mapping. There are several monitoring levels, with different levels of detail. At the highest detail, the following information may be monitored:

Worker waiting time Should the set of tasks scheduled to be executed by a worker be empty or consist only of tasks blocked on input, the worker may enter a waiting state and wake up whenever new tasks are available for execution. This waiting time is written to the logs.

Task blocked on input or output A task may be blocked on input or output. In both cases, information regarding the streams the task is connected to is logged. This information details on which stream the task is blocked on, plus

the number of records that have been written to or read from each stream. A special *activity* flag indicates whther reading from or writing to a stream unblocked the task on the other side of that stream.

Task execution time As soon as a task has finished executing, its total execution time is written to the logs. Task execution time includes time spent blocking on input or output.

Message trace Each record that flows through an S-Net has a unique identifier. Each time a record is produced or consumed a timestamp is recorded and written to the logs. These message traces allow for linking corresponding messages together. Note that the message traces do not store the type (i.e. tags and fields) of the message.

3.3 Performance metrics using LPEL

Using the logs written by the monotoring function of LPEL, the basic metrics and performance metrics described in the previous chapter may be computed. Note that the only basic metric that is used for deriving the performance metrics is *latency*.

3.3.1 Latency

Calculating earliest or late response consists of the following steps:

1. Find the entry task of the network
2. Find the message-id
3. Determine descendants of the message
4. Find the output stream of the network
5. Find the earliest / latest time one of the descendant messages is written to the output stream

Measuring latency involves keeping track of correspondence: messages produced by a box due to the consumption of a message correspond. In case of networks with a high *gain*, tracking correspondence may become a resource consuming task.

Some measures of latency are straightforward to derive from the LPEL logs. These are the latency of the execution of a single *task* (task latency) and the latency of the existence of a single message as it is created when produced by a task and destroyed by a subsequent consuming task.

3.3.1.1 Utilisation

Utilisation of a single processing element (i.e. core) is calculated from the reported task latencies. These task latencies are gathered from the execution times reported by the LPEL worker executing the task. Using the worker's start and end time, we may then also obtain the *active time* of element e, $T_{\text{active}}(e)$. System utilisation, U, is then calculated by summing individual core utilisations, as is described in section 2.1.1.

3.3.2 Communication overhead

Communication overhead is obtained by calculating the cumulative message travel times, T_{travel} and dividing it by the sum of the execution times of the individual cores. The cumulative record travelling time is found by iterating over all the records that have existed in the system. A record m was produced by a task at time $t_O(m)$ and, at a later point in time, consumed by another task at time $t_I m$. The cumulative record travelling time is then found by summing over all messages:

$$T_{\text{travel}} = \sum_{m \in \mathcal{M}} t_I(m) - t_O(m)$$

and communication overhead is then obtained by dividing by the total execution time:

$$C = \frac{T_{\text{travel}}}{\sum_{c \in \mathcal{E}} U(c) \cdot T_{\text{active}}(c)}$$

3.3.2.1 Synchronisation overhead

Each synchrocell in an S-Net contributes to synchronisation overhead. Synchronisation overhead of a single synchrocell is calculated by taking the difference between the time the first record was consumed by a synchrocell and the time the synchronised output record was produced.

3.4 Case study: X-Ray image processing

In this section we measure the three key metrics defined in the previous chapter. We use the example application described in chapter 1 as a use case. The computational intensity of the image processing application can be controlled by varying the rate at which images are produced. We therefore obtain performance metrics for three different settings:

- Low frame-rate: 40ms between two successive images (25 frames per second)
- Medium frame-rate: 33ms between two successive images (33 frames per second)

Figure 3.2: Part of the task-to-worker assignment logfile for a run of the example S-Net application. The application was run on a machine with two cores.

Figure 3.3: Part of the logfile produced by a single LPEL worker. The snapshot was obtained from a single run of the example S-Net application.

- High frame-rate: 25ms between two successive images (40 frames per second)

Performance metrics are obtained from the log files. A run of the S-Net application yields a number of logs. There is one central log which shows the task-to-worker assignment. A part of this log is shown in figure 3.2.

For each worker, a separate log contains task-level timing information. From this information, travelling time of messages, waiting times and task execution times may be obtained. A small part of such a worker's log is shown in figure 3.3.

For each framerate, we measure per core the time used for computation and use that to calculate the core's utilisation. At system-level, we measure the cumulative communication time (total waiting time for messsages) and use that metric to calculate the communication overhead. Finally, synchronisation overhead is measured to gain insight in the nature of the application. Results are shown in tables 3.1, 3.2 and 3.3.

The three tables show a number of characteristics. First of all, core and system utilisation increases as the framerate increases. This is to be expected, as the amount of information the application needs to process within the same amount of time increases. The second observation is that communication time increases fast as the framerate increases. This means that the time a record resides on a communication channel (i.e., a memory location in a shared-memory system) increases. Communication time increases faster than computation time, which leads to an increase in the communication overhead. The third and final obvservation is that, synchronisation overhead decreases. Again, this is to be expected, as computations are primarily synchronised on the framerate. Note that most of the synchrocells in the network operate in parallel, which is why the reported synchronisation time is higher than the total runtime of the application.

	Core 1	Core 2	System
Computation time (s)	12.874	10.484	23.358
Communication time (s)	-	-	45.498
Utilisation	0.115	0.094	0.105
Communication overhead	-	-	1.948
Synchronisation time (s)		-	302.7

Table 3.1: Aggregated performance metrics for a 100-second run of the example application at 25fps (2640 clock records with T = 40ms)

	Core 1	Core 2	System
Computation time (s)	14.988	12.615	27.603
Communication time (s)	-	-	65.377
Utilisation	0.134	0.113	0.124
Communication overhead	-	-	2.368
Synchronisation time (s)		-	298.1

Table 3.2: Aggregated performance metrics for a 100-second run of the example application at 30.3fps (3200 clock records with T = 33ms)

	Core 1	Core 2	System
Computation time (s)	18.887	17.411	36.298
Communication time (s)	-	-	90.461
Utilisation	0.169	0.156	0.163
Communication overhead	-	-	2.492
Synchronisation time (s)		-	278.2

Table 3.3: Aggregated performance metrics for a 100-second run of the example application at 40fps (4224 clock records with T = 25ms)

Chapter 4

Conclusion and future work

This report gives an overview of the performance metrics required to allow the constrution of statistical resource usage models. The basic metric used to derive performance metrics is task latency. Performance metrics defined in chapter 2 and implemented in chapter 3 are translated from the domain of multiprocessor systems on chip (MPSoCs): existing metrics of *fragmentation* and *communication distance* as described in [5] and [9] have been replaced with *utilisation* and *communication overhead*.

As a case study, this document reports on the performance metrics obtained for the X-ray image processing use case from work package 7.

4.1 Future work

The observed performance data must be made available to higher levels in the tool chain. As a result, decisions to alter the current mapping or re-compile parts of the system will lead to a change in performance observation. The final result is an iterative optimisation of the application's performance.

The next step is the design of this feedback mechanism, including protocols for communicating the observed performance and interfaces the observation module should implement. This involves the following activities:

- The design and implementation of an online observer rather than the offline processing of LPEL logfiles.
- A communication protocol to deliver the observed performance data to higher levels (i.e., resource usage models).
- Gathering the requirements for the observation module, represented in a single coherent interface.

These activities will be reported upon in forthcoming deliverables.

Bibliography

[1] Rob Albers. Modeling and control of image processing for interventional x-ray. In *PhD thesis*, 2010.

[2] Giorgio C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2004.

[3] C. Grelck, A. Shafarenko (eds):, F. Penczek, C. Grelck, H. Cai, J. Julku, P. Hölzenspies, S.B. Scholz, and A. Shafarenko. S-Net Language Report 2.0. Technical Report 499, University of Hertfordshire, School of Computer Science, Hatfield, England, United Kingdom, 2010.

[4] P K F Hölzenspies, T D ter Braak, J Kuper, G J M Smit, and J L Hurink. Run-time Spatial Mapping of Streaming Applications to Heterogeneous Multi-Processor Systems. *International Journal of Parallel Programming*, 38(1):68–83, November 2009.

[5] Philip Kaj Ferdinand Hölzenspies. *On run-time exploitation of concurrency*. PhD thesis, Enschede, April 2010.

[6] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM.

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

[8] T D ter Braak. Run-time Spatial Resource Management in Heterogeneous MPSoCs. Master's thesis, Univ. of Twente, August 2009.

[9] T D ter Braak, P K F Hölzenspies, J Kuper, J L Hurink, and G J M Smit. Run-time Spatial Resource Management for Real-Time Applications on Heterogeneous MPSoCs. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE 2010), Dresden*, pages 357–362. European Design and Automation Association, March 2010.