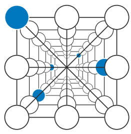

Project no. 248828

ADVANCE

Strategic Research Partnership (STREP)
ASYNCHRONOUS AND DYNAMIC VIRTUALISATION THROUGH PERFORMANCE ANALYSIS TO SUPPORT CONCURRENCY ENGINEERING

SVP Program Transformation and Implementation Evaluation
D24

Due date of deliverable: Jan 31, 2013
Actual submission date: Mar 11, 2013

Start date of project: February 1st, 2010

Type: Deliverable
WP number: WP3
Task number: WP3c

Responsible institution: UvA
Editor & and editor's address: Clemens Grelck
University of Amsterdam
Computer Systems Architecture group
Science Park 904, 1098XH Amsterdam, The Netherlands

Version 1.0 / Last edited by Clemens Grelck / Mar 11, 2013

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
0.1	25/02/2013	Clemens Grelck	UvA	Deliverable frame
0.2	27/03/2013	Clemens Grelck	UvA	Executive sumary
0.3	01/03/2013	Clemens Grelck	UvA	Draft Chapter 1
0.4	03/03/2013	Clemens Grelck	UvA	Draft Chapter 2
0.5	04/03/2013	Clemens Grelck	UvA	Draft Chapter 3
0.8	06/03/2013	Clemens Grelck	UvA	Mostly complete
0.9	07/03/2013	Clemens Grelck	UvA	Mising bits and pieces
1.0	10/03/2013	Clemens Grelck	UvA	Final version

Reviewers:
Alex Shafarenko, Clemens Grelck

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
WP3c	SVP Program Transformation and Implementation Evaluation	UvA*

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

Executive Summary

This document summarises the progress made in Work Package 3 on hardware virtualisation during the 3rd reporting period and outlines the correspondances with other work packages, namely WP4 (static analysis), WP5 (compilation methods) and WP6 (runtime resource management).

Following the concretisation of the notion of hardware virtualisation as a dynamic execution and resource management platform for S-Net streaming networks of asynchronous components in previous project phases, the 3rd project year has seen continuous analysis of and technical improvement in the design and implementation of the SVP hardware virtualisation layer.

Focussing on task WP3c (program transformations) we have created the technical prerequisites for fully dynamic task migration between physical execution resources of our first reference hardware platform: commodity hardware of multiprocessor (multi-socket) systems with multi-core processors, potentially internally hardware-multithreaded, but with a shared address space memory.

Major project resources have been scheduled to expanding hardware virtualisation to systems with distributed address spaces: strongly coupled clusters of workstations or compute nodes as well as more loosely coupled networks of systems. While much progress has been made in a variety of technical areas, this work is continuing into the 4th reporting period.

An important aspect of the Advance project in general and the hardware virtualisation layer in particular is to report system behavior back to higher layers of the Advance technology stack. The monitoring system developed in the previous reporting period has continuously been improved and intensively been used in the dynamic analysis of the Advance use case applications, which in turn initiated several improvements to the hardware virtualisation layer.

We complete the deliverable with a summary of planned activities in the 4th reporting period.

Contents

Chapter 1

Introduction

1.1 Overview and Context

Figure **??** contextualises Work Package 3 within the ADVANCE project. Core to the work package is the design, implementation and evaluation of the SVP System Virtualisation Platform. During the first two years of the Advance project SVP has been concretised to a dynamic execution and resource management platform for S-Net [**?**] streaming networks of asynchronous components, where components are implemented either in plain C or preferably in the functional array language SAC [**?**].

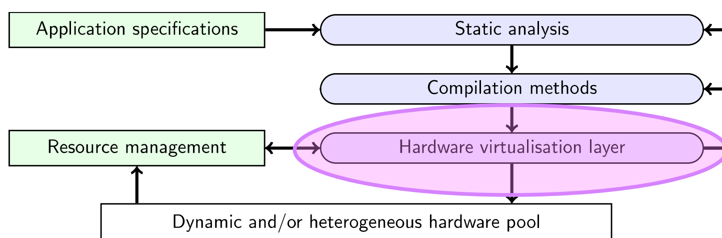

Figure 1.1: Positioning of SVP in the context of ADVANCE

SVP is the mediator between S-Net streaming networks and the concrete hardware they are supposed to run on. S-Net exposes concurrency and dependencies of computational tasks, but is inherently resource-agnostic and resource-unaware. The system virtualisation platform SVP, in contrast, is aware of the computational resources at hand (compute nodes, processors, cores, hardware threads, memories) and maps S-Net tasks to concrete execution units in an efficient way. Furthermore, it continuously monitors the dynamic behaviour of the streaming network and reports the corresponding information to the upper layers of the Advance technology stack, namely static analysis (WP 4) and compilation methods (WP 5) for refinement of an application's implementation. SVP likewise interfaces with external

runtime resource management (WP 6), which makes advanced mapping decisions based on the monitoring data. These decisions are communicated back to SVP, which dynamically implements them by adjusting the task-to-resource mapping accordingly.

1.2 Advance Technology Stack

Fig. **??** illustrates the Advance technology stack from a more technical perspective. Going from top to bottom, the S-Net compiler takes an S-Net coordination program and compiles it to the S-Net *Common Runtime Interface (CRI)*. This is a well-defined interface that exposes the structure of an S-Net streaming network as an application-specific call tree of application-agnostic library functions instantiated with application-specific data structures. The library functions of the common runtime interface can be (and have been) instantiated with alternative implementations and thus allow for entirely different technical realisations of S-Net streaming networks.

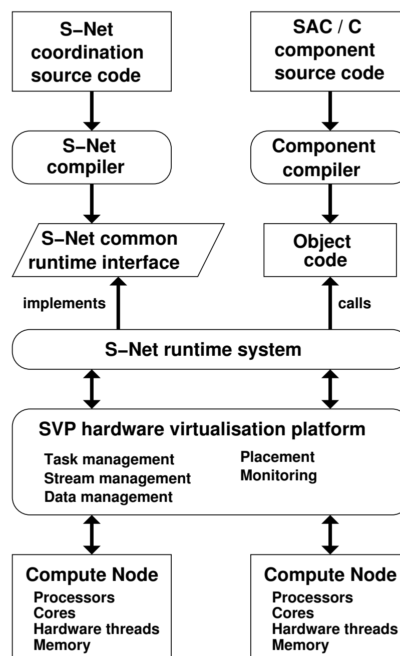

Figure 1.2: Advance technology stack

For the Advance project only one implementation is relevant, which we refer to as *the* S-Net runtime system for simplicity. This runtime system [**?**] follows an approach similar to communicating sequential processes (CSP). Each S-Net component, including a number of internal components for splitting and merging streams, is instantiated as such a sequential process. Internally, an S-Net compo-

nent executes an event loop that reads a record from the input stream (potentially blocking on an empty stream) and processes that record, depending on the kind of component and the record data. In the case of S-Net boxes this usually involves calling an external function implemented in a component language and compiled to binary code by the corresponding component compiler.

This may trigger the dynamic instantiation of further streaming network parts (due to dynamic serial and parallel replication) and usually results in one or more records to be sent to the output stream(s). To this end, component execution may block on a full output stream[1]. The process continues until a special input record signals the component to terminate. This can be due to global network shutdown or due to partial network garbage collection [**?**].

The S-Net runtime system, as described above, is resource-agnostic. The implementation of S-Net tasks and streams and their mapping to a constraint set of resources is the function of the system virtualisation layer. As a part of SVP we use the *Light-weight Parallel Execution Layer (LPEL)* [**?**] to implement the above S-Net components and the streams via which they communicate. LPEL maps the S-Net components to a given fixed number of kernel worker threads for shared memory execution platforms and organises their orderly interaction.

1.3 Advances in System Virtualisation

The primary aim (and achievement) of the previous reporting period was to implement the hardware virtualisation layer SVP on at least one architectural platform. The platform chosen was multi-core commodity hardware, more precisely multiprocessor (multi-socket) systems with multi-core processors, potentially internally hardware-multithreaded, but with a shared address space memory. In this reporting period the existing platform infrastructure has been refined in many aspects. For example, it is now possible to run internally data-parallel SAC-implemented components [**?**] within the realm of the SVP shared memory layer LPEL [**?**]. This work has been described in great detail in deliverable D22, and we refrain from reproducing the same information here.

With particular respect to task WP3c (program transformations) and strong interaction with WP 6 on resource management we rather focus on the aspect of task migration between physical execution resources. This required an adjustment of the boundary between the S-Net runtime system and the SVP layer, but we now have created the technical means to migrate tasks from one resource to another if an asynchronous oracle decides so. We have implemented and evaluated basic task migration policies within SVP itself, but our ultimate goal is to use the resource management policies developed in WP 6. To this end we teamed up with partner TWENTE to define the necessary interfaces and expect a wealth of experimental results during the final reporting period. Our work in this area is detailed in

[1] Streams are bounded to create back pressure and thus make sure that the S-Net streaming network as a whole makes progress and produces output.

Chapter **??**.

Major project resources have been scheduled to expanding hardware virtualisation to systems with distributed address spaces. Much progress has been made towards our long-term objective, but we also encountered unexpected and time-consuming issues that delayed our implementation progress. In Chapter **??** we report on some of these issues and give an outlook how this work will continue into the 4th reporting period.

An important aspect of the Advance project in general and the hardware virtualisation layer in particular is to report system behavior back to higher layers of the Advance technology stack. The monitoring system developed in the previous reporting period [**?**] has continuously been improved and intensively been used in the dynamic analysis of the Advance use case applications. While the produced monitoring data forms the basis for work in WP 4 on static analysis and WP 6 on resource management, it also provides relevant feedback for the refinement of the SVP implementation. In Chapter **??** we report on two such cases that kept us busy for a while.

From a project management perspective, we have continued to merge team efforts between partners UvA, HERTS, HWU, USTAN and TWENTE around the same software code base, and build a synergy of personal involvements. Team members across consortium partners continue to interact on a daily basis, both at design and at implementation level.

Chapter 2

Task Migration and Mapping

2.1 Problem identification

SVP, more precisely its shared memory layer LPEL, uses a fixed number of worker threads individually bound to one computing resource, e.g. a core or a hyperthread on some core. The worker threads act as representatives of the corresponding hardware resources on the software side. Whenever an S-Net runtime system component is instantiated, a placement oracle decides which worker thread shall take care of the component. In S-Net runtime component instantiation does not only happen at network deployment, but due to dynamic serial and parallel replication combinators component execution and further deployment of newly instantiated subnetworks are typically interleaving activities at runtime.

It is characteristic for LPEL that once a component instance is mapped to a worker, this mapping persists until the component instance terminates, either due to global network shutdown or due to partial network garbage collection. It is likewise clear that in the presence of bounded computing resources any useful S-Net streaming network cannot grow, and thus continuously instantiate new components, forever. We can, thus, identify two categories of well behaving networks: (1) networks that converge towards a stable state and (2) networks where instantiation and garbage collection of components are reasonably balanced. In particular, the former category will quickly not require, and thus not allow, any new placements of components to worker threads, and thereby computing resources. However, even networks of the latter category are not really suitable for dynamic load balancing because both network re-instantiation and network garbage collection are application-driven and, therfore, put LPEL into a reacting position rather than the desired direction position.

The main reason for the given design lies in the interaction between the S-Net runtime system and the LPEL threading layer. As explained in Section **??**, the S-Net runtime system follows the concept of Communicating Sequential Processes, and once instantiated it executes its own event loop without giving control back to LPEL between completing processing one record and starting to process the next.

This situation is unsatisfying for two reasons as it does not allow us to dynamically re-map component instances to different worker threads for improved workload distribution and thus improved latency and throughput. In particular, it prevents us from addressing central objectives of the Advance project in actually reacting on statistically aggregated data from execution monitoring.

Before we move on to solving this issue, we should, however, point out that the rather simple one-shot placement heuristics introduced by LPEL work surprisingly well for most streaming networks that we have investigated so far.

2.2 Solution: Task migration

To allow for dynamic migration of component instances from one worker thread to another we need to reconsider the interplay between the S-Net runtime system and the threading layer. In particular, we must dispense with the pure Communicating Sequential Processes (CSP) approach on the runtime system level and move towards a Continuation Passing Style (CPS) approach. In a pure CPS approach an S-Net runtime system component no longer has an event loop, but merely processes a single record only. As soon as a component is finished with processing this one record, it re-instantiates itself, connects its input and its output stream to the new component instance and terminates. As a consequence, after processing a single record the S-Net runtime system yields control to the LPEL threading layer, which thus receives the opportunity to map the new instance onto a worker thread of choice.

The appealing aspect of this apporach is that it would allow us to effectively migrate S-Net runtime system components at each incoming record. Here, we clearly capitalise on the semantics of S-Net that deliberately requires (and to the degree possible enforces) the absence of any information flow from one activation of a component instance by an incoming record and the subsequent such activation by the next record. This supports fairly light-weight task migration without the need for costly identification and even costlier shipping of a potentially large state space.

The not so appealing aspect of a pure Continuation Passing Style approach is the fact that re-instantiating a component, terminating the old one and re-wiring the streams inflicts considerable overhead that would occur for each component instance activation, regardless of whether or not some placement oracle actually suggests migration. In practice, the concern for runtime overhead rather suggests to only migrate component instances occasionally when there is a good reason to do so, e.g. observed load imbalance.

To find a suitable compromise we actually implemented a hybrid approach, combining aspects of both the CSP and the CPS models. An S-Net runtime system component, i.e. a *task* from the perspective of LPEL, does finish after processing one record and thus returns control to thethreading layer. However, the task context on the LPEL layer stores a so-called *continuation* in the form of a function pointer.

If LPEL chooses not to migrate the task, which should be seen as the default behviour, LPEL immediately calls that continuation function and thus the additional overhead with respect to the original CSP approach remains minimal.

2.3 Asynchronous placement oracle

The alert reader may have spotted the potential contradiction at the end of the previous section: immediately calling the continuation function may be inexpensive, but how does LPEL decide whether or not to migrate? Depending on the heuristics used this placement oracle may easily consume cosiderable compute time, e.g. for assessing the workload status of other worker threads. If such an oracle would be consulted each time after an S-Net component instance yields control to LPEL and before potentially calling the continuation, the proposed solution would not be cheap at all.

In essence, we must move the poentially expensive placement oracle off the critical path of component execution. To achieve we use two further properties in an LPEL task context: the current worker id and the desired worker id. Whenever a component instance yields control to the LPEL threading layer, the latter merely compares these two values. In most cases they can be expected to coincide and, thus, LPEL immediately calls the continuation function. Only if the target worker thread id differs from the current worker thread id, LPEL invests the now unavoidable overhead to actually respawn a new task on the desired worker thread and re-establishes the streams accordingly.

The actual placement oracle is thus as far as possible decoupled from the operation of the S-Net streaming network. At the same time it runs on the level of the system virtualisation platform and, therefore, has access to the relevant monitoring information that forms the basis of educated placement choices.

2.4 Initial strategies and analysis

To put our design at test we developed two rather straightforward migration strategies: *random migration* and *waiting time migration*. With random migration a task is marked for migration with a given probability p. If so, the placement oracle updates the target worker property to some random worker. This strategy is useful to experimentally verify whether task migration has any noticeable effect on latency and throughput (or jitter), be it positive or negative. It also allows us to quantify the overhead introduced.

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

2.5 More information

We have summarised our technical contributions as well as the experimental findings in a research paper [**?**] that was presented at the 2nd HiPEAC Workshop on Feedback-Directed Compiler Optimization for Multicore Architectures (FD-COMA'13) during the 8th HiPEAC Conference in Berlin, Germany. For convenience, a copy of this paper is attached to this document as Appendix **??**.

2.6 Interfacing with Work Package 6

Our main task in Work Package 3 is to provide the technical means to exercise educated placement decisions with little overhead on the level of the system virtualisation layer, not to make clever placement decisions ourselves. The latter calls for close collaboration with our partner TWENTE and their advanced resource management policies and techniques developed in Work Package 6. The design of our asynchronous placement oracle and its maximal decoupling from the operational parts of SVP/LPEL provide an ideal basis for this. At the end of the reporting period TWENTE and UvA agreed on a fairly simple interface to link S-Net/LPEL-based applications with the WP6 placement technology and thus create an asynchronous placement oracle based on that technology.

Whereas our waiting-time based oracle described above took the privilege to directly access internal data structures, the WP 6 technology will use the monitoring facilities. For the interpretation of these data, the TWENTE resource management technology requires some insight into the static as well as dynamic structure of an S-Net streaming network. This marks another interfacing requirment. Here we capitalise on the implementation architecture of S-Net, more precisely the Common Runtime Interface (CRI, see Section **??**). Through a rather straightforward re-implementation of the CRI interface functions TWENTE and UvA jointly set up a convenient runtime representation for any S-Net streaming network as a basis for resource management.

Chapter 3

SVP on Distributed Memory Systems

3.1 Overview and status

During the previous reporting periods a major objective of Work Package 3 was to implement the SVP system virtualisation platform on one architecture, which was chosen to be commodity shared memory systems with multiple sockets, multi-core processors and potentially multiple hardware threads per core. A major undertaking during the 3rd reporting period has been to extend SVP to distributed memory architectures with a focus on clusters of the above architectures as nodes.

In this work we build on our experience with an MPI-based distributed memory implementation of S-Net, named Distributed S-Net [**?**]. As illustrated in Fig. **??**, Distributed S-Net consists of two additional combinators for explicit placement of subnetworks on nodes, either statically by specifying a specific node or dynamically through a variant of indexed parallel replication where the tag value not only determines the network replica but at the same time the node id which hosts that network replica. While the exact mapping of subnetworks to nodes in the cluster is decided by the Distributed S-Net application itself, all further organisational aspects of implementing the given mapping is taken care of by the Distributed S-Net runtime system, an extension of the S-Net runtime system (see Fig. **??**).

3.2 From Distributed S-Net to Distributed SVP

One of the objectives of the Advance project is run vanilla S-Net streaming networks without placement combinators on distributed memory architectures and to implicitly control the mapping of components to nodes based on monitoring of dynamic behaviour. While the mapping itself is beyond the scope of Work Package 3, UvA have had to face a major implementation effort to move the distribution layer from the level of the runtime system to the level of the system virtualisation layer (see again Fig. **??**). Technically speaking, it needs to be integrated with LPEL, the

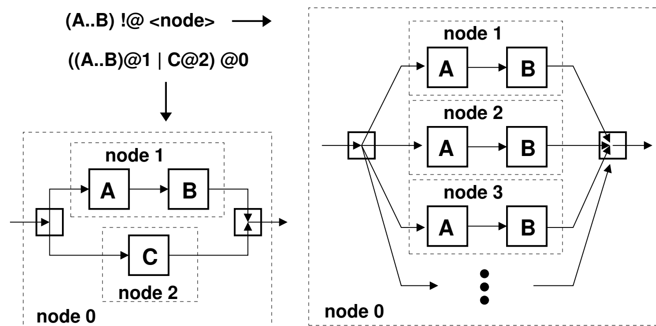

Figure 3.1: Example applications of static placement (left) and indexed dynamic placement (right), where we assume the tag `<node>` to feature values between 1 and some upper limit

threading layer of SVP such that the S-Net runtime system itself is no longer aware of the distributed nature of the computing system, and, instead, SVP transparently connects streams across nodes and ensures the necessary data tranfers.

Fig. **??** illustrates the internal organisation of an SVP node. In practice, each node hosts a number of independent network sections. These are implemented by means of the LPEL threading layer in exactly the same way as in a pure shared memory scenario. Node boundaries are hidden from the rest of the LPEL system and in particular from the S-Net runtime system within specific implementations of streams. To manage streams that cross node boundaries each SVP node runs two special kernel threads (in addition to the LPEL worker threads): an *input manager* and an *output manager*. The output stream of one network section and the input stream of the subsequent network section must be considered as the opposite ends of the same stream on different SVP nodes.

Output and input managers transparently move records along these streams and take care of the necessary data marshalling and unmarshalling. In contrast to our design for Distributed S-Net, input and output managers are internally not multi-threaded but single-threaded and multiplex the various incoming and outgoing streams internally. It is very important to keep the number of pre-emptively scheduled kernel threads under control as the monitoring system relies on controlled cooperative scheduling of tasks and no pre-emption of worker threads.

In a naive approach data attached to record fields would be serialised alongside the records themselves whenever a record moves from one node to another. This obviously inflicts high overhead due to marshalling and unmarshalling of potentially large data structures and puts high demand on the network performance of a distributed system. It is also generally undecidable even at runtime whether data really needs to be sent to the node hosting a follow-up network section. In particular due to flow inheritance in S-Net, records typically piggy-back data that has been produced by earlier network stages and/or will be needed by later network

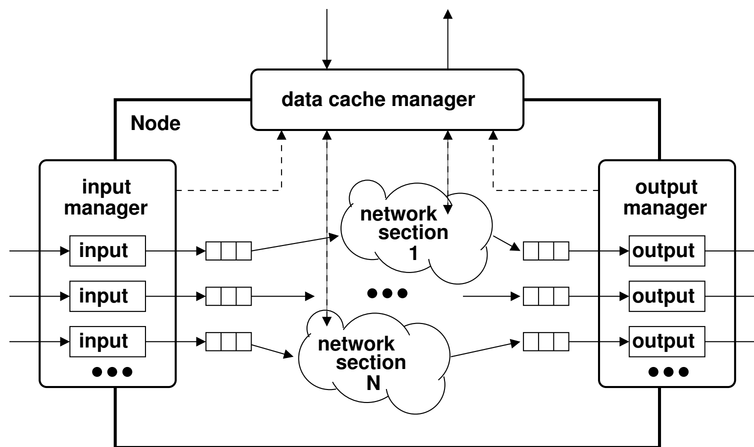

Figure 3.2: Internal organisation of SVP nodes

stages that network sections in between are not even aware of and, hence, are not needed on nodes that execute such intermediate network sections.

To avoid unnecessary data transfers we follow the design of Distributed S-Net and separate data management from stream management. Data associated with record fields is never transferred between the nodes alongside the records themselves. Instead, only a representation of the data, consisting of the field label, the *unique data identifier (UDI)* of the data and the current location of the data are sent. The data itself is only fetched on demand when a box has unpacked the required fields from an incoming record and is about to process the corresponding data. As illustrated in Fig. **??**, the *data manager* controls the movement of data across nodes and effectively implements a distributed software data cache. Actually, the data manager is not an active component (like the input and the output manager of an SVP node, but rather is a kind of service API that allows to request data to be materialised in local memory. LPEL tasks doing so block until the required data has arrived. Assuming a high level of concurrent activities on each node, the high latency of on-demand data fetching is effectively hidden. Incoming data is received by the node's input manager which is the universal contact point for all incoming messages.

While the move from Distributed S-Net to Distributed SVP has already challenged UvA's resources in the Advance project, we discovered two unexpected shortcomings in the original design of Distributed S-Net that required us to find new solutions for our distributed memory implementation of SVP. We elaborate on these issues in the following two sections.

3.3 Distributed network instantiation

A particular challenge of any S-Net implementation is the fact that S-Net streaming networks are not static but evolve at runtime through dynamic replication of networks, both in serial and in parallel. For a distributed memory implementation this is a particular challenge as the re-instantiation of the argument network of a star (serial replication) or blink (parallel replication) combinator generally involves multiple compute nodes. So the output and input managers, in addition to the tasks described in the previous section, send and receive messages to

The briefly described network creation process can be decoupled and abstracted into two sub-activities: The first one is the generation of the network while the second one is the actual instantiation.

This distinction allows for a more flexible management of the execution of the distinct parts of the network. To do this, a new abstract data structure has been devised and introduced. This required adaptations of both the compiler and the runtime system: in the new implementation, an S-Net program, by the tree of function calls generated by the compiler and mentioned in the previous paragraph, has been modified to do not create and instantiate the network in one operation, but to generate and populate an abstract tree data structure that represents the network.

Every operator, component and their parameters has now an abstract representation that holds the parameters and the information about their neighbors.

The instantiation procedure can be now done at run-time according to this data structure. During this operation, the streams among the components are created and interconnected and the components are created according to the parameters.

One immediate advantage is the possibility to instantiate only portions of the network and, therefore of the program. The idea is to enable the runtime system to decouple, replicate and manage only portions of the program across multiple computing resources as required.

To achieve this goal, an indexing and look up mechanism for this tree structure was required: every node of the tree data structure is indexed using an integer map which follows the pre-order visit of the tree during its generation.

3.4 Distributed reference counting

We have summarised our technical contributions in a research paper [**?**] that was submitted to the 4th annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13) in Seattle, USA. For convenience, a copy of this paper is attached to this document as Appendix **??**.

Chapter 4

Feedback from Monitoring to SVP Implementation

4.1 Overview

The availability of a detailed runtime monitoring system [?] for SVP/LPEL not only serves its primary purpose to guide program transformations, mapping decisions, etc, but it has the added side benefit to also provide us with valuable feedback on the quality of design decisions and implementation choices within the system virtualisation layer itself. In close collaboration between partners HERTS, USTAN and UvA feedback from the monitoring system has led to many refinements in the SVP/LPEL implementation. In the following two sections we elaborate on two representative cases to illustrate the nature of these improvements.

4.2 Network garbage collection

Fig. **??** illustrates the S-Net realisation of the X-ray use case [?] provided by Philips Healthcare. It is an example of a category-2 streaming network in the classification introduced in Section **??**, i.e. a network where continuous expansion due to serial replication (in this case) and continuous network garbage collection are in a stable relationship. Every incoming record triggers the unfolding of one more replication of the star subnetwork. However, the leading synchrocell within the star subnetwork only synchronises once and, consequently, only one record with both field `T` and field `d` is emitted, which either takes the top or the middle branch. All subsequent records will pass the (dead) synchrocell and take the lower branch, which only consists of an empty filter. Thus, the entire instance of the star network becomes obsolete and will be garbage collected.

Figure 4.1: Excerpt from Philips Healthcare X-ray use case

While the garbage collection system apparently worked well (no resource deadlock), overall performance was unsatisfactory. Thanks to the availability of the monitoring system we were able to trace the problem back to an effective synchronisation barrier at the internal collector component, i.e. the stream merger, that feeds the output stream of the star subnetwork. Despite the fact that the star is a non-deterministic one, the second token that effectively bypasses the first instance of the star could not pass the collector and so none of the following records either. Taking into account that both the upper and the middle branch involve substantial computing this situation effectively leads to a sequentialisation of the entire operation and, thus, to the observed system behaviour.

To understand why it comes to the observed behaviour, we need to explain more about the principles of network garbage collection. When the synchrocell on the left hand side synchronises, it sends the synchronised record followed by a sync-token with a pointer to the input stream and the sync type ({T,d}) to the output stream and terminates.

The parallel splitter therafter (an implicit S-Net runtime system component in Fig. **??** shown as the split of the synchro-cells output stream into three streams) interprets the sync-token. Given the sync-type, two of its three output streams can no longer be taken. The splitter sends terminate-tokens to these streams and forwards the sync-token (without the sync-type) to the third stream. After that the splitter terminates.

When the collector (another implicit S-Net runtime system component in Fig. **??** shown as the merge of three streams into one output stream of the star subnetwork) receives a terminate token, it removes the input stream from its stream set. As soon as its stream set is empty (terminate tokens received on all input streams), it forwards the last terminate token to the output stream and terminates. The collector eventually receives two terminate-tokens and one sync-token. After that it forwards the sync-token to the output stream and terminates.

The collector must not forward the sync-token before having received the terminate-tokens on the other streams. Those streams are likely to produce relevant data still, which must be forwarded to the output stream, whereas forwarding of the sync-token will lead to premature deconstruction of the network. This effectively, however, results in the collector becoming a synchronisation barrier, which was not intentional.

Once painfully understood, the solution was fairly simple. The (non-deterministic) collector must receive the sync-token and put it on hold for now. There can only be one such token, so constant storage is sufficient. At the same time the collector replaces the input stream on which it received the sync-token by the stream stored inside the sync-token. The collector then continues to receive and forward records on all input streams as usual. This avoids the currently observed barrier synchronisation. Eventually, the collector receives the corresponding terminate-tokens on the other streams. After receiving the last terminate-token the collector forwards the sync-token kept on hold and terminates. In the given example the last step above directly connects two adjacent star splitters, which is another case

for garbage collection detected within the star splitter independent of the above proposal.

Without detailed monitoring capabilities it had probably been impossible to identify this rather intricate misbehaviour of the SVP/LPEL layer.

4.3 Analysis of ant colony optimisation use case

Our second example is also a good example of intensive technical collaboration between partners HERTS, USTAN and UvA, which led to a joint research paper [**?**] presented at at the 2nd HiPEAC Workshop on Feedback-Directed Compiler Optimization for Multicore Architectures (FD-COMA'13) during the 8th HiPEAC Conference in Berlin, Germany. For convenience, a copy of this paper is attached to this document as Appendix **??**.

In this case not unsatisfactory performance triggered our interest, but the aim of partner USTAN to analyse the ant colony optimisation use case [**?**, **?**] provided by SAP with statistical means. More precisely,

Chapter 5

Conclusion and future work

This report gives a representative overview on research and development activities in Work Package 3, beyond the concrete task W3c, during the 3rd reporting period. It highlights the cintinued close collaboration of partners HERTS, TWENTE, USTAN and UvA to realize the Advance objectives in this area.

Substantial progress has been made in a number of directions, but some unforeseen technical and research challenges in connection with (too) limited resources have not allowed us to achieve all original objectives within the initially planned project time span. We are confident that the recently granted extension of the Advance project puts us into the state to still achieve these objectives, provided that additional resources become available as discussed within the consortium.

Towards the end of the Advance project we plan to complete our work on a distributed memory implementation of SVP according to the outline given in Chapter **??**. The agreed interface beween SVP and the WP6 resource management techniques are currently being implemented and we expect soon to be in the position to run experiments to validate their effectiveness.

Bibliography

[1] W. Cheng, F. Penczek, C. Grelck, R. Kirner, B. Scheuermann, and A. Shafarenko. Modeling streams-based variants of ant colony optimisation for parallel systems. In *HiPEAC Workshop on Feedback-Directed Compiler Optimization for Multicore Architectures (FD-COMA'12), Paris, France*, pages 11–18, 2012.

[2] C. Grelck. The essence of synchronisation in asynchronous data flow. In *25th IEEE International Parallel and Distributed Processing Symposium (IPDPS'11), Anchorage, USA*. IEEE Computer Society Press, 2011.

[3] C. Grelck, J. Julku, and F. Penczek. Distributed S-Net: Cluster and grid computing without the hassle. In *Cluster, Cloud and Grid Computing (CC-Grid'12), 12th IEEE/ACM International Conference, Ottawa, Canada*. IEEE Computer Society, 2012.

[4] C. Grelck and F. Penczek. Implementation Architecture and Multithreaded Runtime System of S-Net. In S.B. Scholz and O. Chitil, editors, *Implementation and Application of Functional Languages, 20th International Symposium, IFL'08, Hatfield, United Kingdom, Revised Selected Papers*, volume 5836 of *Lecture Notes in Computer Science*, pages 60–79. Springer-Verlag, 2011.

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

[6] Clemens Grelck. Shared memory multiprocessor support for functional array processing in SAC. *Journal of Functional Programming*, 15(3):353–401, 2005.

[7] Clemens Grelck and Sven-Bodo Scholz. SAC: A functional array language for efficient multithreaded execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006.

[8] Chris Jesshope, Mike Lankamp, Michiel van Tol, Thomas Bernard, and Raphael Poss. Svp model reference (the blue book). `https://notes.svp-home.org/book2.html`, 2009.

[9] 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. In C. Grelck, K. Hammond, and S.B. Scholz, editors, *2nd HiPEAC Workshop on Feedback-Directed Compiler Optimization for Multicore Architectures (FD-COMA'13), Berlin, Germany*. HiPEAC, 2013.

[10] V.T.N Nguyen, R. Kirner, and F. Penczek. A multi-level monitoring framework for stream-based coordination programs. In *12th International Conference on Algorithms and Architectures for Parallel Processing, Fukuoka, Japan*, 2012.

[11] F. Penczek, W. Cheng, C. Grelck, R. Kirner, B. Scheuermann, and A. Shafarenko. A data-flow based coordination approach to concurrent software engineering. In *2nd Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM 2012), Minneapolis, USA*. IEEE, 2012.

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

[13] M. Schrijver and M. Creemers. Running real-time and best-effort applications concurrently on common off-the-shelf hardware. In C. Grelck, K. Hammond, and S.B. Scholz, editors, *2nd HiPEAC Workshop on Feedback-Directed Compiler Optimization for Multicore Architectures (FD-COMA'13), Berlin, Germany*. HiPEAC, 2013.

[14] Jaroslav Sykora and Sven-Bodo Scholz. Towards self-adaptive concurrent software guided by on-line performance modelling. In C. Grelck, K. Hammond, and S.B. Scholz, editors, *2nd HiPEAC Workshop on Feedback-Directed Compiler Optimization for Multicore Architectures (FD-COMA'13), Berlin, Germany*. HiPEAC, 2013.

[15] M. Verstraaten, R. Poss, and C. Grelck. Deferred diffusion tree joining: A distributed reference counting optimisation. In *4th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'13), Seattle, USA*, 2013. submitted.

[16] Merijn Verstraaten, Stefan Kok, Raphael Poss, and Clemens Grelck. Task migration for s-net/lpel. In C. Grelck, K. Hammond, and S.B. Scholz, editors, *2nd HiPEAC Workshop on Feedback-Directed Compiler Optimization for Multicore Architectures (FD-COMA'13), Berlin, Germany*. HiPEAC, 2013.

Appendix A

Paper:
Task Migration for S-Net/LPEL

By: Merijn Verstraaten, Stefan Kok, Raphael Poss and Clemens Grelck

Presented at: 2nd HiPEAC Workshop on Feedback-Directed Compiler Optimization for Multicore Architectures (FD-COMA'13), Berlin, Germany, 2013

Appendix B

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

By: Kenneth MacKenzie, Philip Hölzenspies, Kevin Hammond, Raimund Kirner, Nga Nguyen Vu Thien, René te Boekhorst, Clemens Grelck, Raphael Poss and Merijn Verstraaten

Presented at: 2nd HiPEAC Workshop on Feedback-Directed Compiler Optimization for Multicore Architectures (FD-COMA'13), Berlin, Germany, 2013

Appendix C

Paper:
Deferred Diffusion Tree Joining

By: Merijn Verstraaten, Raphael Poss and Clemens Grelck

Submitted to: 34th annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13), Seattle, USA