Project no. 248828

ADVANCE

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

Implementation of SVP on at least one target (Software) D16

Due date of deliverable: Nov. 30, 2011
Actual submission date: Mar 20th, 2012

Start date of project: February 1st, 2010

Type: Deliverable
WP number: WP3
Task number: WP3b

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

Version 1.2 / Last edited by Clemens Grelck / March 20th, 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
1.0	09/03/2012	Raphael Poss	UvA	First draft
1.1	16/03/2012	Raphael Poss, Clemens Grelck, Merijn Verstraaten	UvA	Complete document
1.2	20/03/2012	Clemens Grelck	UvA	Final version

Reviewers:
Clemens Grelck

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
WP3b	Efficient implementation of SVP on different platforms	UvA*, HERTS, TWENTE

°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 the SVP implementation efforts during the second reporting period and outlines the work planned for the third period. Figures 1 and 2 position this work within the context of the ADVANCE project from a high level and from a more technical perspective.

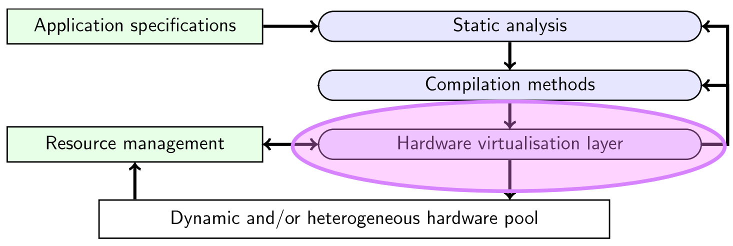
Application specifications
Static analysis
Compilation methods
Resource management
Hardware virtualisation layer
Dynamic and/or heterogeneous hardware pool

Figure 1: Positioning of SVP in the context of ADVANCE (high level)

The primary aim of the work reported here was to implement the hardware virtualization layer SVP on at least one architectural platform. The platform chosen is multi-core commodity hardware, more precisely multi-processor (multi-socket) systems with multi-core processors, potentially internally hardware-multithreaded. The upcoming reporting period will see this work extended to networks of such computers (both on-chip and off-chip, homogeneous and heterogeneous) and multi-cores with accelerators (e.g. GPUs). We also describe our preliminary work in these directions.

As shown in Figures 1 and 2 a key feature of the hardware virtualization layer is to report system behavior back to higher layers of the ADVANCE technology stack. A monitoring system to this effect has been devised and implemented, and is described in this document.

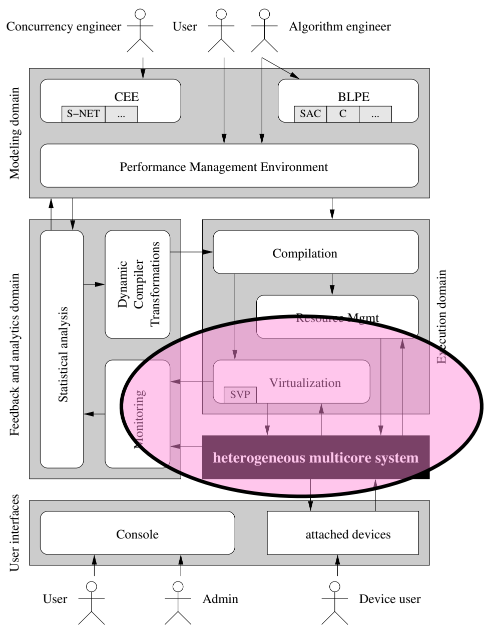

Figure 2: Positioning of SVP in the context of ADVANCE (technical perspective)

Contents

Chapter 1

Introduction

1.1 Overview

The main goal of WP3b was to provide a software implementation of the SVP instruction set, defined as an API to a set of concurrency control instructions that create, manage synchronize and schedule SVP tasks.

This has been completed for hardware platforms made of commodity multi-cores, with partial support for GPU accelerators. The work has been performed in collaboration with partner HERTS.

During the reporting period it was found that *monitoring* of the application's activity is a crucial feature of the SVP layer in ADVANCE, more crucial than the diversity of supported platforms or refined SVP management primitives. Consequently, we have allocated resources to analyze the shortcomings of the initial monitoring support and devise a solution for the next reporting period.

Meanwhile, work is ongoing on two fronts. One direction is the integration of SVP with S-NET. Here work is ongoing to identify the *communication activities* in the S-NET run-time system. Simultaneously, effort is invested towards extending support to other platforms, namely heterogeneous clusters of multi-cores.

1.2 SVP and the S-NET operating software stack

During the last reporting period the "nature" of SVP has been further refined. The SVP run-time *operating software* of S-NET applications in ADVANCE is formed by the following layers:

1. the low-level *hardware abstraction* layer, in charge of providing virtualization of hardware processors and memories;

2. the *system access* layer, in charge of providing *access* to the low-level resources, including managing *process identities* for monitoring;

3. the *concurrency management* layer, in charge of organizing the distribution of work units within a process over multiple resources, and organizing the synchronization and communication between them;

4. the *application network execution* layer, in charge of defining new work units based on the abstract application description and flowing record data between them.

Prior to ADVANCE, the fourth aspect was under the responsibility of the "S-NET run-time system", whereas aspects #1 to #3 were offloaded to a traditional multithreaded"operating system" such as GNU/Linux. In fact, two complementary run-time systems existed: one targeting shared memory multi-core multiprocessors through the POSIX multi-threading API, the other targeting clusters of such systems via MPI.

In the context of ADVANCE, more control was desired and standard interfaces for portability across platforms. *Introspection* of platform characteristics and run-time behaviors mandated to customize these layers, and the *System Virtualization Platform* (SVP) is now extending this vision to coordinate all layers of the run-time stack and provide the desired portability.

The second reporting period was thus a coming of age with the joint realization of partners HERTS and UVA that there should not be a clear separation of concerns between a *S-NET run-time system* and the abstraction interfaces below. To guarantee the success of ADVANCE, knowledge must be provided to statistical analysis across all the layers, in a coordinated fashion. Because of this, the *S-NET operating software*, constituted by all the infrastructure around the application-specific components ("boxes" in S-NET), has become a joint endeavor and a strong partnership between partners HERTS and UVA. This partnership is now also extended to TWENTE, as the entity in charge of providing control of placement of work will be provided by TWENTE.

From a project management perspective, this has involved merging the team efforts around the same software code base, and build a synergy of personal involvements. Team members across consortium partners are interacting on a daily basis, both at design and at and at implementation level.

Chapter 2

SVP and S-NET on Multi-cores

The cornerstone of an implementation of SVP for shared memory multi-cores has been the implementation of the Light-Weight Parallel Execution Layer (LPEL) in the first reporting period, described in [8]. LPEL realizes the goal of SVP to abstract the concurrency management layer of the operating software stack. It was also designed to reduce jitter by binding cooperatively scheduled *tasks* to long-lived "worker" threads, which in turn are bound to hardware cores. This is in contrast to the pre-ADVANCE situation when an underlying operating system autonomously controlled the placement of "fat" threads without knowledge of their interaction on the S-Net level. The LPEL layer provides first-class primitives to create *streams* of record field data between tasks, whose activity participate in scheduling decisions.

In the second reporting period, work has continued on LPEL jointly between partners UVA and HERTS. A particular focus has been its integration with the S-NET run-time environment. This constitutes the first SVP platform, and has already successfully been exploited by project partners [1].

This implementation includes support for application components ("boxes") implemented in a sequential language, either C, C++ or SAC, connected in parallel applications using the S-NET language, and executed concurrently over multiple processors, cores and hardware threads in a shared memory system.

2.1 Achievements for this reporting period

The following developments have been carried out since the first reporting period:

- The system has been extended to trace scheduling activities and save monitoring data to files for off-line analysis.
- The tasking layer has been optimized to "reuse" task contexts and reduce jitter caused by allocation and deallocation activities.
- Portability of the system has been extended to a diversity of underlying platforms, namely GNU/Linux, Solaris, OS X, BSD;

- The network execution layer has been extended to perform "garbage collection" of sub-networks that have been active once but are known not to be used any more [2].

- Implementation support for the "feedback combinator" feature of S-NET, as described in WP2c (D11), has been added to the platform.

- The code base has been re-hauled to remove the distinction between "multi-core S-NET" and "distributed S-NET" so as to provide common interfaces independently of the targeted computing platform. This prepares for the upcoming period where support for clusters will be integrated into the system.

- The system has been modified to identify explicitly "resource usage" *besides* processors: heap memory allocations and system calls for I/O, in particular, were identified in preparation for the upcoming period where optimized implementations will be provided for both.

- The system was analyzed and profiled for inefficiencies. These are identified below.

2.2 Areas for improvement

The following aspects have been identified as shortcomings in the current implementation, and possibly detrimental to the work of the next reporting period unless addressed:

- The system does not yet track memory usage per application component, only globally. Accounting per application component, and also per system component involved in the application's execution, is necessary to profile activity at run-time and provide feedback to compilation (WP5).

- The system does not yet track bandwidth usage over data streams between application components. On a shared memory system, streaming activities are translated to cache misses per processor, which should be measured and reported. This information is needed to fully characterize behavior in statistical models (currently it is a "hidden" statistical variable that hinders correlations).

- The system does not yet provide a unified access to I/O devices and more generally communication of an application with "the outside world": file system, I/O channels, network, etc. Originally, the S-NET language specification required application components to be fully state-free and perform no I/O operations directly. In accordance to this, the current support assumes that application components do not perform calls to the underlying operating system, or perform them in a way fully invisible to the ADVANCE operating software stack. However, close interaction with industrial project partners has shown that support for I/O by application components is both practical and desirable. Because of this, the monitoring infrastructure must be modified to properly account for external I/O activity separately from scheduling and communication within the application.

- There is yet little support for inspecting and troubleshooting the overall state of the application network at run-time. The current S-NET facilities do allow to inspect *data in streams*, and place debugger breakpoints *within* application components (boxes). Unfortunately the second reporting period has revealed that some higher-level behaviors between application components, for example unexpected placement of tasks to workers, cannot be observed with the existing facilities alone.

2.3 Plans for the third period

The remaining effort will be dedicated in priority to addressing the shortcomings identified in the second period, namely by:

- enhancing the monitoring infrastructure to track memory allocations and cache activity;
- providing hooks to integrate the mapping technology from TWENTE, which will use heuristics to optimize task to worker placement;
- extending the SVP implementation as an integrated framework where components for observation, statistical analysis and placement decision can run within one system process, to reduce overheads;
- providing additional tools and interfaces to visualize the behavior of running programs;
- co-designing the *run-time network description language* with partner HERTS, which will implement introspection over VR-nets (WP2, D11). With this abstraction in place, it will be possible to track back both stream communication events and memory allocation events to application components for monitoring.

Simultaneously with these efforts, the following work will continue as background, continuous activities:

- infrastructure support for any new S-NET features that will simplify testing and adoption by partners and prospective users;
- support, maintenance and documentation as required by project partners.

2.4 Preliminary support for accelerators

Boxes that wish to exploit GPUs are partially supported, as follows. A box expressed in SAC can be compiled automatically to use GPU primitives, or a box in C in C++ can use a GPU API explicitly. If this box is connected with S-NET into the application, and *is the only one to request GPU access*, the application will run as desired.

Currently, GPU usage is invisible from the S-NET operating software perspective, as it is entirely encapsulated within the box abstraction. Effort will be assigned to integrating accelerators in the machine model used by the placement sub-system.

2.5 Dissemination

The first version of software covered by this deliverable has been made available in Q3 2011 to all project partners via the ADVANCE document repository. Installable versions are disseminated regularly, and the source code of the operating software and corresponding utilities has been made available online under an open source license at `https://github.com/organizations/snetdev`.

Chapter 3

The S-Net Monitoring System

3.1 Current Implementation

The goal of the S-NET monitoring system is to provide information that can be used to allocate resources based on a statistical model of a program's runtime behavior.

3.1.1 How and What to Monitor?

The S-NET monitoring system tracks two separate categories of monitoring events. The first category is produced in the runtime implementation of S-NET entities. The (optional) second category is produced by the threading implementation. Currently each worker (LPEL) or entity (pthread) opens a monitoring file during initialization. All events generated within this worker or entity are written to this monitoring file.

Boxes, syncrocells and filters are the entities that produce monitoring events in the first category. There are two such events; one for incoming records and one for outgoing records. Incoming record events are generated when a box, syncrocell or filter reads a record from its input stream or when a record enters the network. Outgoing record events are generated whenever a box, syncrocell or filter writes a record to its output stream. Both types of events have a time stamp and a record id associated with them. When these events are generated they are queued in the threading layer until box execution finishes, upon which they are written out to the monitoring file.

Unlike the original directly pthread-based S-Net runtime system implementation, the LPEL threading implementation, provides several extra monitoring events in addition to the runtime events described above. These events include:

Wrapper creation: wrappers are dedicated pthreads that run only one task (i.e. input or output managers). This event has a time stamp and start message associated with it.

Worker creation: workers are pthreads that run multiple tasks. This event has a monitoring version, time stamp and start message associated with it.

Worker destruction: this event has a time stamp and end message associated with it.

Worker unblocking: runs whenever a worker unsuspends. This event has a time stamp, wait cause and idle time associated with it.

Task finishing: runs whenever a task is suspended. This event has a time stamp, task state/blocked on information, task identifier and execution time associated with it.

3.1.2 Monitoring Problems

Our goal, in the context of ADVANCE, is to provide runtime adaptivity based on feedback from statistical models. This means that the tools implementing the statistical models need access to S-NET's monitoring information. At this time the monitoring information is simply written to files. This is not suitable for our goals because of the following reasons:

1. File IO introduces jitter,
2. output format needs to be parsed, adding to latencies,
3. feeding information back to the runtime is cumbersome, as a new external interface must be defined.

The first two aspects are large overhead in the feedback loop which is not negligible compared to the activity of application components. Meanwhile, both a parser for the output format and the external interfaces for controlling the S-NET run-time system introduces an overhead in development and maintenance to the project's effort budget, not counting an extra opportunity for software bugs.

Meanwhile, the statistical modelling tools need a way to provide runtime adaptation information to the runtime. Currently there is no method to do this, so one has to be implemented.

3.2 Future work

Two important questions for future adaptation of the monitoring system are: What monitoring information shall we expose and how shall we expose it?

3.2.1 How to Monitor?

The current method of exposing monitoring information to the outside world is undesirable because it's slow, costs a lot of effort to maintain and requires us to implement a new interface to feed adaptation decisions back to the runtime.

Providing hooks for callbacks is a better approach to exposing the monitoring information. These callbacks can then send the relevant information to the statistical analysis tool(s) or even run (parts of) the analysis code directly. This avoids slow file accesses, removes the need to (de)serialize information (avoiding the additional work of maintaining a parser) and provides a direct way to change runtime behavior, since the hooks can directly access any data structures/interfaces used for runtime adaptation.

Effort to implement these hooks has been invested in a close collaboration between partners UvA, HERTS and TWENTE.

3.2.2 What to Monitor?

The question of what to monitor can be answered by taking a look at the intersection of what monitoring information we want and what monitoring information is available. At the very minimum we can keep the currently provided information, which is specified by partner USTA.

The following additional metrics will be sampled as well:

- maximum amount of memory allocated by a box,
- total memory allocated by a box,
- record and field transfer times (Distributed S-NET).

Other events that can be tracked, but aren't currently are:

Event name	**Event description**
WorkerWaitStart	Worker execution suspended.
TaskDestroy	Task execution has finished.
TaskAssign	Task assigned to worker.
TaskStart	Task execution started.
StreamOpen	Stream opened.
StreamClose	Stream closed.
StreamReplace	Stream replaced with another stream.
StreamReadPrepare	Started read from stream.
StreamReadFinish	Finished reading from stream.
StreamWritePrepare	Started write to stream.
StreamWriteFinish	Finished write to stream.
StreamBlockon	Blocked on full/empty stream.
StreamWakeup	Woken up by read from/write to stream.

Further interactions with the project partners will help identify which metrics are the most relevant for statistical analysis and behavior aggregation.

Chapter 4

SVP on Distributed Systems

While work is ongoing to provide an execution environment on single multi-core systems (chapter 2), effort is also invested towards providing support for S-NET on networks of multi-cores.

Early support for S-NET over homogeneous clusters had been implemented prior to ADVANCE using MPI, reported on in [3,4]. This implementation is based on explicit placement by the concurrency engineer of application components to named nodes in a network.

Since this early investment, partners UVA and HERTS have teamed up to extend this work in two directions. One is a general improvement of the interfaces to extend the system to other platforms than MPI, for added portability and efficiency. Another is to create a framework where the application components can be dynamically re-assigned during the execution, with placement information computed automatically at run-time. This is intended to enable plugging in the technology from partners TWENTE and USTA. We report on both these directions below.

4.1 Improvements to Distributed S-Net

Under the management of UVA the distributed S-NET operating software was ported to Intel's Single-Chip Cloud computer, a platform which exposes 48 cores over a fast on-chip network. On this platform, the most efficient form of communication must exploit dedicated hardware structures to map memory regions directly across cores, instead of using message passing.

Because of the distinct communication machinery, the effort at UVA has extended the S-NET code base with a more generic distribution layer specializable to both MPI and other forms of inter-node communication like the one used for Intel's Single Chip Cloud Computer (SCC) [5, 7]. Thanks to this investment, the mechanisms for data communication between components have been streamlined, removing unnecessary data copies, excess synchronization and excess serialization. This work was reported on in [9, 10]. The resulting framework, once distributed separately from the main S-NET operating software, is now integrated into the main code base and available to project partners and the wider public.

4.2 Work migration and automatic placement

In the work described above placement of components is under the responsibility of the application. S-Net boxes and subnetworks may statically be placed onto named computing resources, or a mapping can be computed at runtime by some S-Net box. The (Distributed) S-Net runtime system automatically detects streams that pass the boundaries of memory domains and automatically takes care of the necessary explicit communication and the associated serialization/deserialisation requirements.

The dynamic unfolding of S-Net networks involves dynamic network creation and network garbage collection across multiple nodes, any box or subnetwork once placed at some node cannot be migrated. In the next reporting period, UVA and HERTS will work jointly to remove this restriction, make placement fully flexible, and create opportunities for fully automatic placement of components. The following technical directions have been identified:

- The S-NET tasks that wrap around application components will be decomposed using continuation passing style, so that their state and stream endpoints can be captured and migrated to different nodes between activation events.

- Memory management, once delegated to the underlying operating system, will be taken under the responsibility of the S-NET operating software, so that the distribution layer can choose between data migration and data caching when activities on multiple network nodes operate on the same record data.

- S-NET's automatic garbage collection mechanisms, responsible for reclaiming system resources left over after task termination, will be extended to reclaim task resources and stream state after tasks are migrated across the network.

- The distributed monitoring system already implemented using standard TCP/IP network sockets will be extended and adapted to feed observed metrics from a distributed execution into the aggregation and placement technologies from partners TWENTE and USTAN.

Again, as above the work will occur in the main S-NET source code repository and will be made available to partners as it is implemented.

Chapter 5

Conclusion

This report gives an overview of the SVP platforms made available to project partners during the second reporting period. It highlights the close collaboration of partners HERTS and UVA to realize execution environments and monitoring capabilities for S-NET applications on a diversity of platforms.

5.1 Summary of future work

The last reporting period will strengthen the multi-core platform used for designing statistical model, provide additional facilities for monitoring and extend the implementation to support networks of multi-cores. This extension will simultaneously focus on portability of the framework's services and the integration of the partners' technologies for seamless use over all SVP platforms.

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. to appear.

[4] Clemens Grelck, Jukka Julku, and Frank Penczek. S-Net for multi-memory multicores. In *DAMP '10: Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming*, pages 25–34, New York, NY, USA, 2010. ACM.

[5] J. Howard, S. Dighe, Y. Hoskote, S.R. Vangal, et al. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *IEEE International Solid-State Circuits Conference (ISSCC'10), San Francisco, USA*, pages 108–109. IEEE, 2010.

[6] 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.

[7] T.G. Mattson, R.F. van der Wijngaart, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Sriram Vangal, Nitin Borkar, Greg Ruhl, and Saurabh Dighe. The 48-core scc processor: the programmerâĂŹs view. In *Conference on High Performance Computing Networking, Storage and Analysis (SC'10), New Orleans, USA 2010*. IEEE, 2010.

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

[9] M. Verstraaten. High-level programming of the single-chip cloud computer with S-Net. Master's thesis, University of Amsterdam, Amsterdam, the Netherlands, February 2012.

[10] Merijn Verstraaten, Clemens Grelck, Michiel W. van Tol, Roy Bakker, and Chris R. Jesshope. Mapping distributed S-Net on to the 48-core intel scc processor. In *3rd Many-core Applications Research Community (MARC) Symposium*. KIT Scientific Publishing, September 2011.