Project no. 248828

ADVANCE

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

Hardware virtualisation notation
D6

Due date of deliverable: November 2010
Actual submission date: XXXXXX

Start date of project: February 1st, 2010

Type: Deliverable
WP number: WP3
Task number: WP3a

Responsible institution: UvA
Editor & and editor's address: Computer Systems Architecture group
Institute for Informatics
University of Amsterdam
Science Park 904
1098XH Amsterdam, the Netherlands

Version 1.0 / Last edited by Raphael 'kena' Poss / November 2010

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	01/11/2010	Raphael 'kena' Poss	UvA	Initial version
1.1	02/11/2010	Raphael 'kena' Poss	UvA	Addressed comments from Chris Jesshope
1.2	03/11/2010	Raphael 'kena' Poss	UvA	Restructured document after discussion with C. Jesshope and C. Grelck
1.3	16/11/2010	Raphael 'kena' Poss	UvA	Separated document into multiple CSA notes. Automated construction of report from CSA notes.
1.4	05/03/2011	Raimund Kirner	HERTS	Document renamed from D3.6 to D6.

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
WP3a	Extensions to SVP and the SL language	UVA, HERTS, STAND, TWENTE

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

*Task leader

Executive Summary

Task WP3a was carried out and proposals for change to SVP and the SL language are presented in this deliverable. This completes the first milestone of WP3 successfully.

In the process of accomplishing task WP3a, we have clarified the role of SVP in the project and technical aspects of the interactions between SVP, S-NET and the mapping technology provided by TWENTE. This is also reported in this document.

The output of the research performed as part of this deliverable was documented in the documentation system of the CSA group at the University of Amsterdam. This report is constructed as a compilation of the relevant technical notes from our documentation system.

Contents

Chapter 1

Introduction to D3.6

Key: adv15

Status: Draft

Authors: Raphael 'kena' Poss

Date: 2010-11-17

Source: http://notes.svp-home.org/adv15.html

Version: adv15.txt 4158 2010-11-17 23:42:55Z kena

The report for deliverable D3.6 covers task WP3a, "Extensions to SVP and the SL language". This defines three sub-tasks:

- extension of SL to support the expression of extra-functional concurrency;
- definition of a method/language to describe and/or access hardware characteristics;
- definition of a method to monitor the execution of programs.

1.1 Organisation

In **??** we first introduce SVP at an abstract level, by giving its fundamental principles. We then detail in **??** what SVP is and what it means in the context of ADVANCE. In particular, we outline its composition as a technology.

We also recognize in **??** a potential shortcoming of the implementation of SL for the analysis of extra-functional requirements, namely the lack of typing information in the compiler. In **??** we propose to enhance SL to address this shortcoming. Then we propose a SL extension to express arbitrary extra-functional requirements in **??**.

Then we describe resource management in SVP. In **??** we identify actual hardware targets planned for use in ADVANCE. Then we propose an abstract scheme to identify and describe hardware for use by SVP programs. We then propose in

?? how to integrate the partner technology Kairos (from TWENTE) for on-line resource mapping within SVP.

Next we describe the interaction between SVP and S-NET in **??**. In particular, we recognize the need to adapt the S-NET implementation to express its concurrency using SVP. We then propose an implementation scheme for S-NET and we highlight its benefit for later stages in the project, in particular in the area of resource management.

Also, in XXX we propose a scheme to enable run-time monitoring of activities. This is based on the attribution of process identities to work units, and a mechanism to gather run-time observations about entire processes.

Finally in **??** we summarize the concepts introduced in this document and in **??** we summarize the implementationt tasks.

Chapter 2

Abstract machine model

Key: adv2

Authors: Raphael 'kena' Poss

Status: Draft

Date: 2010-07-06

Version: adv2.txt 3911 2010-07-06 16:27:24Z kena

Source: http://notes.svp-home.org/adv2.html

Abstract: This note describes an abstract machine model for the ADVANCE "hardware virtualization layer".

2.1 Overview

Our concurrency management system is a programming system where:

- *asynchrony* is available as a program primitive;
- where asynchronous computations are *located* into virtual *execution resources*;
- where the finite nature of execution resources is acknowledged and integrated into the execution semantics;
- where resource-specific opportunities for compile-time optimizations are visible in program semantics.

The peculiarity of this abstract machine is that it does not expose *memory* (from hardware storage) as a shared device, nor does it identify *channels and communication* (from interconnects), to be used arbitrarily by concurrent processes. Instead, interactions between asynchronous activities are controlled by *data dependencies* and *dependency patterns* of different kinds. The mapping of data dependencies and synchronization around them is delegated entirely to the environment.

2.2 Fundamental principles

2.2.1 Asynchrony, scheduling and data dependencies

The first fundamental principle is that programs *express* asynchrony and loose mappings of computations to resources but can make no assumption about the *scheduling* of this asynchrony within each resource, both over time (interleaving) and over space (mapping of individual units of work to independent physical hardware). These aspects are entirely delegated to the execution environment.

The model for this principle can be defined as follows:

- *schedule-agnosticism*: if two program fragments A and B are expressed to be asynchronous relative to each other, then the execution of A may precede the execution of B in time, or it may succeed the execution of B, or the executions of A and B may be (partly or completely) simultaneous, or the executions of A and B may be interleaved.

  This indeterminate schedule applies whether A and B are located on the same virtual resource or not; that is, a single virtual resource may or may not provide actual (time-simultaneous) concurrency.

- *interference within places*: programs cannot assume any scheduling, in particular fair scheduling, to guarantee progress in non-terminating or misbehaving programs mapped to the same resource. In other words, if A and B are expressed to be asynchronous and co-located onto the same virtual resource, and the execution of A does not terminate, then either the execution of B does not start, or it starts and may not terminate even if B would otherwise terminate if the execution of A didn't start yet. Also, A and B might be configured to execute so that if A performs an invalid action with side-effects, the execution of B might become undefined (i.e. A and B are not *isolated*).

- *non-interference across places*: fair scheduling is guaranteed across independent virtual resources. In other words, if A and B are expressed to be asynchronous and located onto two distinct virtual resources, then whether the execution of A does or does not terminate, the execution of B will start at some point in time and be guaranteed progress.

  Additionally, if the two virtual resources are also associated to distinct *protection domains*, then invalid actions in A will not influence the execution of B (i.e. separate protection domains provide isolation between places).

Despite these properties (or lack thereof) on scheduling, asynchrony can be controlled by means of *data dependencies*. Programs can express that two program fragments A and B are relatively asynchronous except for their uses of a specific *data item*. The *kind* of the data item and the *dependency pattern* are both expressed and determine the semantics of program actions that access the data item during execution. This can be defined as follows:

- *data-driven synchronization*: when two program fragments *A* and B* are expressed to be asynchronous but also declare to share a collection of data items (*d1*, *d2*...) of one or more kinds (*k1*, *k2*, ...) according to a dependency pattern *p*, then the scheduling of all actions that operate on the data items is further constrained by scheduling properties determined by *p* and the set of kinds (*k1*, *k2*, ...).

Another document will detail how to define data items, what basic kinds are available, how to describe a dependency pattern, and the scheduling constraints corresponding to compositions of kinds.

2.2.2 Exclusion

The second fundamental principle is a unique exception to the first principle: execution resources are also the vessel for *exclusion*. This can be defined as follows:

- *exclusive composition*: if two fragments *A* and *B* are expressed to be asynchronous, and *exclusively co-located* onto the same virtual resource *R*, then either: the start of the execution of *A* succeeds (in time) the termination of the execution of *B*, or the start of the execution of *B* succeeds the termination of the execution of *A*.

As will be demonstrated below, this composition mechanism allows both for safe sharing of state (memory, I/O channels) and cooperative reclaiming of resources.

2.2.3 Capped resource usage

The third fundamental principle is that resources are *allocated* for an (arbitrarily large) *finite amount of time*; this contract is enforced by the environment and causes abnormal termination of computations that exceed their time limit. Also, resources have a *finite capacity* for simultaneous work.

This can be defined as follows:

- *completion status*: if three fragments *A*, *B* and *C* are expressed so that *B* is designated as the *normal continuation of A* and *C* is designated as the *failure continuation of A* (see the rest of this document for the specific constructs), then a choice will be performed by the execution environment when *A* terminates (either normally or abnormally) and proceed accordingly either with the execution or *B* or the execution of *C* -- and not both.

  There can be three kinds of failure continuations, one for each failure condition described immediately below.

- *abnormal situations*: some action in a program fragment *A* might cause an abnormal termination during its execution. The abnormal continuation is then executed if defined, otherwise the abnormal continuation of the outer program is execution, and so on recursively.

- *overflow*: if a program fragment A is localized at some virtual resource R; and if at run time the capacity of R cannot accomodate the execution of A (either because the exclusive resource is currently busy, or because no more concurrent resources are available), then the execution of A is interrupted abnormally with an overflow status. The overflow continuation is then executed if defined; if the overflow continuation is not defined the status is handled as if it was indicating abnormal termination.

- *time contract*: if a program fragment A is located at some virtual resource R; and if at run time the execution of A exceeds the time limit of R, then the execution of A is interrupted abnormally with an excess status. The excess continuation is then executed if defined; if the excess continuation is not defined the status is handled as if it was indicating abnormal termination.

- *allocation service*: the environment provides a service that can be queried by means of program actions to request allocation and deallocation of resources with time limits.

Another document will detail how to manipulate resources in programs.

2.2.4 Specialized dispatch

The fourth fundamental principle is that program *implementations* are *specific* to some kind(s) of virtual resources. This means that while hardware resources are virtualized, the program representation (in term of which effective actions will drive its execution) is dependent on the resource where it is located. In other words, we do not describe a "virtual machine" with a uniform set of basic instructions that can be executed on any virtual resource.

Instead, we isolate the following concepts:

- *program descriptions* ("source code"), that are annotated to indicate which kinds of effective resources it can actually target (one or more);

- *executable program representations* ("binary code"), which can actually execute on (virtual) resources;

- *specialization paths* ("compiler tool chains"), which are lazily evaluated functions which produce executable representations as output and take the following as input:

  - a (set of) program description(s),
  - a description of the target virtual resource,
  - optionally, any known data dependency pattern related to program composition,
  - optionally, extra run-time information that is relevant to optimization engines.

These concepts are related as follows:

- program descriptions are *identified*;
- program composition is expressed *by name*, i.e. program reuse is expressed by a composition operator and the identifier(s) of some other program description(s);
- at run-time, when the point of composition is reached, the environment uses its known set of specialization paths to select an appropriate executable representation and satisfy the composition, based on which effective resource is being targeted;
- when program composition expresses localization, the two following conditions must be met:
  - the description of the target program in the composition must indicate that it can run on (at least) the target resource kind, and
  - the environment must know of (at least) one specialization path which produces a valid executable representation for the target program on the target resource;
- shared data items and data dependency patterns can be related explicitly to program composition points in programs, so that the visibility of data dependencies is exposed to the specialization paths and can be used for optimization.

Other notes will detail how program descriptions are annotated, and how executable representations can be further selected in case of ambiguity (multiple representations for a single target resource), and how to relate data dependencies and program composition for the purpose of optimization.

Chapter 3

SVP from the ADVANCE perspective

Key: adv11

Status: Draft

Date: 2010-11-09

Authors: Raphael 'kena' Poss, Michiel W. van Tol, Clemens Grelck

Source: http://notes.svp-home.org/adv11.html

Version: adv11.txt 4157 2010-11-17 23:05:57Z kena

3.1 Background on SVP

The Self-Adaptive Virtual Processor (SVP) as designed in the CSA group at the University of Amsterdam is, foremost, a combination of:

- a software *paradigm*[1]; and
- a set of "best practices" for the implementation of *concurrency support systems* in software and hardware to run programs that embody the SVP paradigm.

We review these two perspectives in the following sections.

[1] The following definition of "paradigm" is used here:

> *a philosophical and theoretical framework of a scientific school or discipline within which theories, laws, and generalizations and the experiments performed in support of them are formulated; broadly : a philosophical or theoretical framework of any kind* -- Merriam-Webster Online, http://www.merriam-webster.com/dictionary/paradigm

3.1.1 The SVP paradigm

SVP touches system software engineering by addressing the following:

- the *expression of concurrency* in program source, by requiring programs to express concurrency, synchronization and communication according to a set of predefined patterns. As such, it can be seen as a *concurrency model*;

- *interactions between programs and their environment*, in particular how programs interact with I/O services and data storage. As such, it can be seen as a framework, taking over many roles usually served by an operating system.

From this perspective, SVP does not mandate the use of a specific programming language. Instead, it imposes guidelines on the expression of concurrency in programs, like how concurrency is created or terminated and how communication is defined. The broad vision of SVP is described in **??**.

The programming language SL (described below) incorporates the SVP principles and has been designed specifically to serve as a common representation to target multiple SVP platforms, but each SVP platform may be targeted by multiple programming languages.

3.1.2 SVP as a concurrency support system

When used as a framework to run concurrent applications, SVP provides the following:

- eager dataflow scheduling: programs issue asynchronous operations that are immediately schedulable, but unsatisfied (explicit) data dependencies cause threads to suspend;

- hierarchical composition of dynamically created, concurrent bundles of execution threads called *families*, executing the same program code but in different execution contexts;

- efficient fine-grained synchronization through *dataflow channels*;

- hierarchical and composable binding of families to named execution resources, called *places*, that are hierarchically defined and managed;

- sharing of state and bounded non-determinism (in particular time-synchronized side-effects on I/O devices) through placement to *exclusive places*, which subsume exclusion to the resource management system;

- protection and state isolation through *place boundaries* between resources instead of attributing identities to activities directly.

It is by making assumptions about the expression of concurrency programs that a SVP system can provide an efficient execution of this concurrency on a given resource. Again, there is no canonical "SVP platform"; run-time systems for different execution resources can implement the management of concurrency in different ways, including support for programs that do not specifically embody the SVP programming paradigm.

However, for the purpose of ADVANCE we have selected a particular implementation route, which will be described below.

3.1.3 SVP as a model of execution

In an environment where a given platform only executes programs written using the SVP paradigm in a SVP concurrency framework, one can also use SVP as an *execution model*[2] to reason about the run-time behavior of programs, including resource usage and communication patterns.

Intuitively, this model is only valid for SVP programs on SVP-aware platforms, i.e. it does not apply when non-SVP forms of concurrency are executed on SVP-aware platforms or when SVP programs are executed on non-SVP platforms.

3.2 Aspects outside of the scope of SVP

The following aspects are not specifically addressed by SVP:

- *portability*, especially across existing (legacy) operating systems: SVP does not provide a set of programming APIs or system interfaces that ensures that the same program code can run unmodified (modulo recompilation) across different operating systems.

  However, it is foreseen that the implementation of an SVP framework for ADVANCE will guide portability and suggest a portability strategy for the partner technologies.

- *hardware virtualization* in the sense commonly in use in 2010: SVP does not provide a virtual hardware platform upon which entire existing operating systems and application code can run unmodified. This would be an abstraction mismatch: SVP drives the implementation of programming languages and concurrency management systems, not the interface layer between software and hardware.

As the second point caused some concern among the partners during the reporting period, we need to clarify the following points here:

[2] The following definition of "model" is used here:

a system of postulates, data, and inferences presented as a mathematical description of an entity or state of affairs -- Merriam-Webster Online, http://www.merriam-webster.com/dictionary/model (definition 12)

- how can the software stack introspect the hardware on which it is running in a uniform way across the envisioned platforms?

  For this, we extend SVP with a "resource description language" that captures the characteristics of hardware uniformly. Although this aspect was not conceptually part of SVP originally, partnership with UTwente has allowed to construct a solution to this.

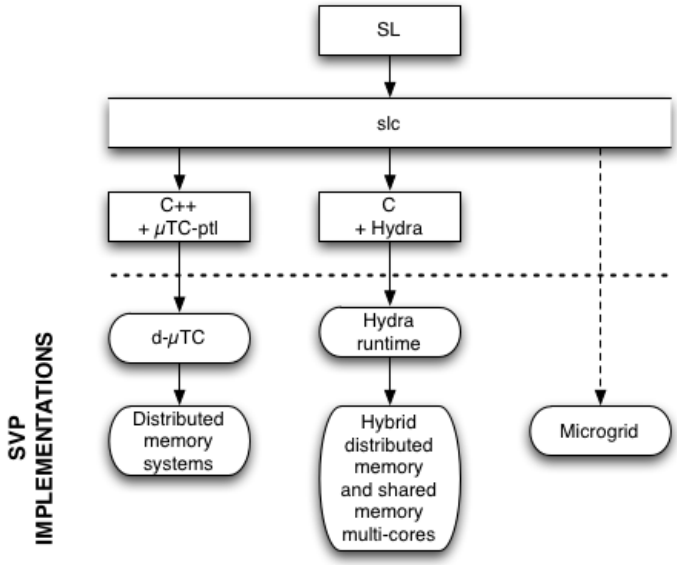
- how can the technology be structured so that single program representations can run on multiple resources with radically different architectures, e.g. a general-purpose CPU in combination with a GPU accelerator or dedicated hardware?

  For this, no clear technical solution at the level of SVP has emerged yet (although the CSA group at the University of Amsterdam will consider this outside of the scope of ADVANCE). However, between partners in ADVANCE it was agreed that the main vehicle to express architecture-neutral algorithm (box) code would be SAC, which has separate compilation routes to SVP and accelerators. The interaction between compiled SAC code targeting SVP and SAC-generated accelerator code will be hidden behind the SVP place and family abstractions.

3.3 SVP as a technology

To exemplify the benefit of SVP in run-time systems, multiple SVP-aware environments have been implemented that exploit the assumptions provided by the paradigm: a custom on-chip distributed many-core architecture (the Microgrid), a concurrency manager running over distributed memory (d-µTC), and another concurrency manager targeting hybrid shared memory and distributed memory systems (e.g. cluster of many-cores) called Hydra ([**?**]).

To support the use of the paradigm in programs, multiple SVP-aware languages have evolved: µTC, designed to program the Microgrid; the µtc-ptl library for C++, designed to program the d-µTC environment; and SL, intended to serve as a common intermediate representation for all these platforms and future SVP implementations.



In ADVANCE we intend to reuse the SL language, with the Hydra implementation as a "primary" target. Depending on partner interests and available resources, other SVP implementation will be considered as well.

3.3.1 The Hydra framework

The Hydra framework is a combination of a software runtime system for SVP programs and the code generation layer for a SL compiler that is able to target this runtime system.

As a runtime system, it implements the necessary functionality for running SVP programs expressed in SL across networks of multi-core computers, taking advantage of the parallelism supported by the hardware and the concurrency expressed in the programs. In particular, it implements low-overhead concurrency creation and synchronization, and a distributed global address space between nodes in a network. The code generation layer in the compiler takes advantage of its various features, as they were developed in tandem.

It currently runs over GNU/Linux on 64-bit architectures. More details are available in [**?**].

In the context of ADVANCE, we plan to extend this framework to support named resources, mutual exclusion and delegation of work to hardware accelerators.

3.3.2 The SL language and compiler

The SL language was designed specifically to enable retargeting a single program representation to multiple SVP implementations. It is an extension of the C language (version C99) and captures the SVP semantics as a number of primitives

for family creation, synchronization on termination, and dataflow synchronization. With the Hydra framework, it is further extended with constructs that facilitate the use of partial memory consistency offered by Hydra between nodes in a distributed memory system.

The specification of SL and information about its implementation in a compiler as of the start of ADVANCE is available in [**?**] and [**?**].

The SL compiler takes C code with SL extensions as input and transforms all the SL constructs to C source code with annotations and optional inline assembly. This C output is then passed to the native underlying C compiler for the target SVP architecture.

3.3.3 Current and future status of the implementation of SL

The attentive reader will notice when exploring references that the SL language, as documented, exposes SVP features with a syntax and semantics that are not specifically human-friendly. This situation is the result of three aspects:

- the original purpose of SL was to serve as an intermediate representation to be generated by higher-level compilers, including the SAC compiler produced by HERTS;
- at the time of its design, another key requirement was the ability to rapidely prototype automatic program transformations in a compiler;
- at the time of its initial implementation, no complete C front-end was available that would both be open-source and easily modifiable for the purpose of research in program transformation.

To summarize, this SL language is regular and homogeneous, but extremely verbose and requires manifest types for variable uses in concurrency constructs. Any C code outside or between SL constructs is passed unmodified through the SL compiler which builds no representation of its semantics other than the sequential control flow around the concurrency constructs.

While the lack of "aesthetics" is irrelevant to ADVANCE because SL will also be used as an intermediate representation in this project, the lack of a *type system* for the native C code is a recognizable potential hindrance for some ADVANCE goals; in particular, the expression of non-functional requirements like throughput constraints are highly dependent on understanding data types and their machine representation.

In order to enable a more comprehensive research with SL in ADVANCE, we therefore propose to extend the SL language with a new syntax and a new compiler front-end which enable full analysis of program semantics.

This would be made possible by, and take advantage of a new compiler technology, the CIL framework ([**?**]), that was uncovered during the first reporting period. CIL provides an infrastructure for performing research on C language extensions

and program transformations, a good fit for the SL language. With this technology in mind, we were able to design a new syntax and flexible semantics for SL, documented in **??**, that we hope to introduce and implement early during the next reporting period.

3.4 Summary of concurrency concepts in SVP

The following concepts are provided by SVP and are explained in **??**:

- Base SVP concurrency concepts:
  - thread, family, singleton family
  - creation (action), family handle
  - place, exclusive place, delegation (action)
  - synchronizing data dependencies
  - synchronization on termination (action)
  - asynchronous termination (action)
  - parent, index
  - thread function, thread context
- Terms from [**?**]:
  - consistency domain, memory object
- Terms from **??**:
  - function, spawn (action)

Chapter 4

Revised SL specification

Key: adv8

Authors: Michiel W. van Tol, Raphael 'kena' Poss

Date: 2010-11-18

Status: Draft

Version: adv8.txt 4159 2010-11-18 00:24:45Z kena

Source: http://notes.svp-home.org/adv8.html

Abstract: This note attempts to reconstruct our SVP language incrementally in very small steps, starting from plain C99 and preserving generality at every step and providing new useful features along the way.

4.1 Introduction

As the research on SVP was generalized from programming single multithreaded cores to on-chip grids and heterogeneous systems, the need for a uniform, resource-agnostic and multi-granularity intermediate language has become strong. This was the original purpose of the Microgrid's µTC, although the initial design of µTC has been conducted in isolation from other works in this field and specialized to control the Microgrid architecture -- not to mention that for implementation purposes µTC has two separate syntaxes, the original µTC and its "SL" macro-like overlay. As a result, the µTC/SL specification looks foreign, unique, complicated and only remotely connected to current state-of-the-art concurrent language design. In other words it was not very attractive.

As required by the ADVANCE project, an intermediate representation more rich and general than the existing µTC/SL language is required to push the boundaries of our research. Simultaneously, modularity is key as different use cases have different requirements on the language semantics and its implementations. For example functional concurrency requires asynchronous function calls to "return" a value, whereas data parallelism requires good control on locality but does not have

a "return value". Also, low-level concerns like scheduling should be accessible and controllable for the programmer of higher-level language run-time systems. As shown below, unifying these requirements in a uniform, modular interface is possible. Additionally, modularity in the semantics and specification achieves separation of concerns in concurrency management: resource allocation, resource partitioning, thread distribution, scheduling, communication and synchronization can be controlled separately by programs, but also *understood* and reasoned about separately.

As a result, this note proposes an incremental (re-)definition of a SVP languag which starts from the plain C99 specification and incrementally adds the desired language features in clearly isolated small steps.

As in most low-level concurrency language specifications, the following guidelines apply: language features that are not used do not entail any mandatory run-time cost (neither space nor time). Functional and non-functional specifications are clearly separated in the language to make (static) reasoning about functional behavior in presence of concurrency possible.

> **Note**
>
> Existing casual µTC/SL users will likely have the impression that this specification designs a quite new language. In reality, it merely exposes and sanitize language semantics that were embedded at various levels of the SVP model, up from programming patterns down to the machine model and the internal stages of our compilers. In other words, most of the semantics presented here are ultimately already implemented on existing SVP resources.
> A section Expressing previous µTC semantics using the new paradigm explains how µTC/SL programs can be rewritten. The design is such that translation can be automated.

4.1.1 Acknowledgements / related work

This thought exercise has been inspired and guided by previous work on Cilk, Chapel, Fortress, the upcoming C1x standard, distributed operating systems, the Apple-CORE and ADVANCE projects.

Ideas and criticisms have been contributed by the SAC and S-NET teams, the Master thesis work of A. Matei, A. Visser and D. Prokesh, and many others.

4.2 Summary

The following section explain how the following are added to the language incrementally:

- a standard header `svp_syntax_aliases.h`;
- implicit typing with `auto`;

- split-phase asynchrony via `async` *types* and a `sync` primitive to wait for completion;
- a special `hold` primitive;
- asynchronous function calls with `spawn`, with optional explicit delegation;
- fault tolerance with `otherwise` and `synctest`;
- parallel replication with `create`, with optional explicit delegation and placement control;
- asynchronous channels where function call arguments can be computed concurrently with the call, using special channel declarations and primitives `produce` and `consume`;
- mixing parallel replication with channels, how to daisy-chain replication using channels ;
- re-expressing the existing μTC/SL constructs for thread functions and create/sync using the new semantics.

Some added bonuses:

- Most of the syntax introduced is functionally equivalent to C code with the new syntax removed, like in Cilk. When equivalence is not reached by syntax removal, it can be reached with a (more-or-less) trivial text substitution.
- Apple-CORE's "deadlock prevention" scheme can be formalized in the new semantics and do not require behind-the-scene magic any more.

4.3 Environment

A standard header `svp_syntax_aliases.h` exists and can be optionally included in programs.

4.4 Limited implicit typing

Summary Here we introduce a simple form of type inference into the language.

A. We specify that the following definition is present in `svp_syntax_aliases.h`:

```
#define auto _Auto
```

So that programs can use the syntax "`auto`" instead of "`_Auto`" in all source code where the identifier `auto` is not otherwise used.

B. We add the keyword `auto` to the language for variable definitions in blocks, with name `_Auto`. This can be used as follows:

```
auto x = EXPRESSION;
```

The variable `x` is declared with the type of `EXPRESSION`. This is equivalent to the following code using the non-standard, but widely supported syntax `typeof`:

```
typeof(EXPRESSION) x = EXPRESSION;
```

Note

Contrary to the C++ feature of the same name documented in http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1705.pdf, we do not propose to allow `auto` in any place of a declaration using a more complex type. Instead, we opt for the simple, fixed-format syntax:

declaration:

...
`_Auto` *auto-declarator-list*

auto-declarator-list:

auto-declarator
auto-declarator `,` *auto-declarator-list*

auto-declarator:

identifier = *initializer*

This restriction may be lifted in the future.

4.5 Split-phase asynchrony

Summary Here we introduce a separation between issuing a computation and waiting for its completion.

A. We specify that the following definitions are present in `svp_syntax_aliases.h`:

```
#define async _Async
#define sync _Sync
#define detach _Detach
```

B. We add the type *modifier* `_Async` to the language. Like `const` or `restrict` this modifies the type to which it is applied. (A precise specification of the language semantics of `async` is given below.)

An initialized object with an `async` type establishes a binding between an asynchronous expression evaluation and the eventual synchronization on termination of the evaluation to use its eventual result. We call it a *promise* for the eventual result type. For example, in:

```
int async x;
```

`x` is a "promise for `int`", and `int` is the *synchronized type* of `x`.

It is valid to apply the qualifier `_Async` to the special type `void`, which indicates that a bound void expression is to be computed asynchronously.

> **Note**
> As described below, we are extending C's *derived types* with *async types*. An async type is said to be derived from its synchronized type, and it (type-wise) incompatible with its base type.

C. We add the polymorphic primitive expression `_Sync` to the language: `_Sync` takes an `async` expression as input, and evaluates to the result of the computation associated with the `async` object. For example:

```
int async x = ....;

int y = sync x;
```

Informally, `sync` "waits" on completion of the asynchronous computation to which it is applied, and returns the computed value.

D. We add the polymorphic primitive expression `_Detach` to the language: `_Detach` takes an `async` expression as input, and evaluates to a void expression. For example:

```
int async x = ...

detach x;
```

The meaning of `detach` is to declare that the evaluation of the `async` expression is to take place but its result can be discarded, i.e. no further use of `sync` will be applied.

4.5.1 Specification for `async`

More than a mere type qualifier or storage specifier, `async` adds a new kind of *derived type* to C. C's derived types comprise array types, structure types, union

types, function types and pointer types; we propose to introduce *async types* to this category.

More specifically, async types are *derived declarator types*. An async type may be derived from an object type, called its *synchronized type*. An async type derived from the synchronized type *T* will be called "promise for *T*"; it describes an object which references an asynchronous computation producing an object of the synchronized type (see Model for async expressions and asynchronous computations below). The construction of an async type from a synchronized type is called "async type derivation".

An async type derived from an incomplete type[1] other than `void` is also an incomplete type; `void async` is a complete type.

The phrase grammar for `async` extends C's *declarator* grammar:

declarator:

pointer[opt] direct-declarator

pointer:

...
`_Async`
`_Async` *pointer*

While the `async` word appears in the same syntax position as C's "`*`" pointer syntax, its semantics are distinct. We specify them here.

First, recall the semantics of declarators in C:

A. In the following subclauses, consider a declaration "`T D1`" where `T` contains the declaration specifiers that specify a type *T* (such as `int`) and `D1` is a declarator that contains an identifier *ident*. The type specified for the identifier *ident* in the various forms of declarator is described inductively using this notation.

B. If, in the declaration "`T D1`", `D1` has the form *identifier* then the type specified for *ident* is `T`.

C. If, in the declaration "`T D1`", `D1` has the form

```
( D )
```

then *ident* has the type specified by the declaration "`T D`" (a declarator in parentheses is identical to the unparenthesized declarator).

Then we define the declarator semantics for `async`:

[1] Incomplete types lack information to determine their size and cannot be used for declarations that also define objects.

D. If, in the declaration "`T D1`", `D1` has the form

```
_Async D
```

and the type specified for *ident* in the declaration "`T D`" is "*derived-declarator-type-list T*", then the type specified for *ident* is "*derived-declarator-type-list async T*".

E. For two async types to be compatible, their base types must be compatible.

F. We extend C's *conditional operator*[2] as follows: either the second and third operand satisfy the base C constraints, or they are both promises to compatible types.

The following example demonstrates the difference between a "promise for a constant value" and a "constant promise to a variable value":

```
const int async promise_to_constant;
int async const constant_promise;
```

In this example, the contents of the object resolved by `sync` on `promise_to_constant` shall not be modified, but `promise_to_constant` itself may be changed to promise a different object. In contrast, `constant_promise` may not be modified.

The following example names several different types:

```
int async              // promise for int
int async[3]           // array of 3 promises for int
int (* async)[3]       // promise for a pointer to an array of 3 int
int async (void)       // type of a function with no parameters
                       // returning a promise for int
int (* async)(void)    // promise for a pointer to a function
                       // with no parameters and returning int
```

4.5.2 Specification for `sync` and `detach`

A. We extend C's phrase grammar as follows:

unary-operator:

...
`_Sync`
`_Detach`

B. The operand of the unary `sync` and `detach` operator shall have an async type.

[2] for example: `a ? b : c`

C. The synchronized type of the operand to `sync` and `detach` shall not be an incomplete type.

D. As described above, `sync` denotes the resolution of a promise for an object. It evaluates to the result of the bound asynchronous computation. If the operand has type "promise for *type*", the result has type "*type*". If no computation is bound, the behavior of `sync` is undefined.

E. The result of `sync` is not a lvalue. (In particular, it has no address and cannot be modified.)

F. A `detach` expression is a void expression[3].

4.6 Basic asynchrony

4.6.1 Asynchronous function calls

Summary Here we introduce the ability to perform function calls concurrently with the caller, in a way similar to Cilk.

A. We specify that the following definition is present in `svp_syntax_aliases.h`:

```
#define spawn           _Spawn
```

B. We add the keyword `_Spawn` to the language, which extends the function call syntax as follows:

spawn-expression:

`_Spawn` *postfix-expression* `(` *argument-expression-list-opt* `)`

async-def-expression:

...
spawn-expression

unary-expression :

...
async-def-expression

Here is an example use:

```
spawn foo(...)          /* expands to _Spawn foo(...) */
```

[3] As per the C specification: the (nonexistent) value of a void expression (an expression that has type `void`) shall not be used in any way, and implicit or explicit conversions (except to `void`) shall not be applied to such an expression. [...] A void expression is evaluated for its side effects.

If the function call would otherwise have return type `R`, the entire `spawn` construct evaluates to an expression of type `async R`, so that the return value can be ultimately retrieved by `sync`. For example:

```
int foo(int);
...
int async f = spawn foo(42);
...
int r = (sync f) + 123;
```

This is functionally equivalent to the same C code with `async`, `spawn` and `sync` removed from the text:

```
int f = foo(42);
int r = (f) + 123;
```

The meaning of `spawn` is to allow the function call to compute asynchronously with the caller thread.

C. We extend the spawn syntax with specifiers:

spawn-expression:

`_Spawn` *spawn-specifier-list[opt] postfix-expression*
`(` *argument-expression-list-opt* `)`

spawn-specifier-list:

spawn-specifier
spawn-specifier spawn-specifier-list

This syntax reads as follows: between `spawn` and the function call syntax there can be additional syntax constructs that we call "spawn specifiers". These influence the semantics of the spawn construct as described below.

4.6.2 Model for async expressions and asynchronous computations

Summary Here we explain the interactions between `async` expressions, `spawn`, `sync` and `detach`.

For the purpose of specify the concepts introduced above more in more detail, we introduce a distinction between `async` expressions and the asynchronous computations themselves as follows:

A. A *computation* is a program entity representing the execution of an expression evaluation *scheduled non-deterministically*. During its lifetime, it has a state and associated storage.

B. As a computation consumes resources, the question of resource reclamation must be addressed. We define that a computation can either be *synchronizable* or *detached*. When synchronizable, its lifetime of its associated storage extends to the first `sync` event that applies to it. When detached, its lifetime extends to the end of its execution.

C. An `async` expression is a *reference* to a computation. As such, it has two states: either *bound* or *unbound*. At the start of its lifetime, an `async` expression is unbound. When bound, it refers to exactly one computation.

D. The `spawn` construct defines a synchronizable computation and evaluates to a reference to this computation, i.e. a bound `async` expression.

E. Both the `sync` and `detach` constructs apply to synchronizable computations via a bound `async` expression. The behavior is undefined if they are applied to an unbound `async` expression or to a detached computation.

   When `detach` is applied to a synchronizable computation, the computation becomes detached.

F. Exactly one `sync` or `detach` construct must be applied to a synchronizable computation, irrespective of which `async` expression is used to access it. If a computation is accessed twice by `sync` or `detach`, the behavior becomes undefined.

G. The behavior is undefined if the last `async` expression bound to a synchronizable computation reaches the end of its lifetime and neither `sync` nor `detach` was applied to it. In particular, *the computation may not be scheduled at all*.

The following example illustrates a situation where the behavior is undefined:

```
int async x = spawn foo(...);
int async y = x;
detach x;
int t = sync y;
```

Here the behavior is undefined because `x` and `y` refer to the same computation, and both `sync` and `detach` are applied. In the following example:

```
{
   int async x = spawn foo();
   /* end of scope */
}
```

the behavior is undefined because `x` reaches the end of its lifetime and neither `sync` nor `detach` was applied.

> **Note**
> From a resource usage perspective, the state of a computation indicates the responsibility for releasing the state and storage allocated to the computation. When a synchronizable computation terminates before a `sync` event, its storage stays allocated until a `sync` operation retrieves the results and releases the resources. If a synchronizable computation has already terminated before a `detach` event, its storage is simply released by `detach`. When a detached computation terminates, it (conceptually) releases its own resources. It is a race condition with undefined behavior when the execution allows both a `sync` event and a `detach` event to reach the same computation.

4.6.3 Split-phase asynchrony and real concurrency

The constructs and semantics described above and the rest of this document merely *expose* concurrency but does not *require* that the associated computations be executed fully concurrently.

In particular:

A. If an implementation does not support real concurrency for a given `spawn` construct, a sequential schedule is still possible;

B. If an implementation does not support fully detached concurrency, it can implement `detach` as an alias to `sync` without violating the semantics.

4.6.4 Pseudo-asynchrony

Summary Here we introduce the ability to use the split-phase asynchrony semantics without actual asynchrony. This not useful *per se* but is used later.

A. We specify that the following definitions are present in `svp_syntax_aliases.h`:

```
/* human namespace alia: */

#define here     _LocalContext

/* syntactic sugar: */

#define hold     spawn here
```

B. We add the polymorphic primitive expression `_LocalContext` (also named `here`) to the language: `here` causes the spawn construct to

perform the work locally (as function calls) in the caller's computation context but evaluates to an `async` expression, such that the function return value is produced when the `async` expression is passed through `sync`.

For example:

```
async int x = spawn here foo();
     // also: hold foo();

int y = sync x; // equivalent to y = foo();
```

This syntax extends the C grammar as follows:

spawn-specifier:

`_LocalContext`

Note
The basic function calls syntax in C has equivalent semantics to "sync spawn here". This is intended.

4.7 Bundling work into families

4.7.1 Parallel replication

Summary Here we introduce simple but general support for parallelism.

A. We specify that the following definitions are present in `svp_syntax_aliases.h`:

```
/* human namespace aliases: */

#define linear(V, ...)              _Linear(V, # __VA_ARGS__)
#define ispace(IterSpace)           _SpawnIterSpace(IterSpace)

/* syntactic sugar: */

#define create(V, ...)              spawn ispace(linear(V, # __

#define parallel_for(V, ...)        sync create(V, # __VA_ARGS_
```

B. We add some spawn specifiers to the language:

spawn-specifier:

...
index-space-specifier

index-space-specifier:

`_SpawnIterSpace` (*index-space-form*)

index-space-form:

`_Linear` (*identifier* , *assignment-expression* , *assignment-expression* , *assignment-expression*)

This syntax reads as follows. The list of spawn specifiers can contain a construct of the form `_SpanIterSpace(...)`. Between the parentheses there can be a construct of the form `_Linear(ID, EXP, EXP, EXP)`. As an additional requirement not expressible in the phrase grammar, the identifier listed in the `_Linear` construct must appear in simple form at the start of the argument list in the function call.

This construct expresses the concurrent application of a function call over a range of values. The return value of each function call is ignored; the entire spawn expression produces no value, but its type is `void async` as it can run concurrently with the caller.

Each execution of the function call will provide a different index value as first argument, over the specified range in an unspecified, *nondeterministic* order. The expression parameters to `_Linear`, as well as the declared type of the first argument for the called function type, must be *compatible scalar types* as per the C definition.

This can be used as follows:

```
auto p = create(i, 0, 10, 1) foo(i, a, b);

...

sync p;

/* or, to synchronize immediately in the caller: */

parallel_for(i, 0, 10, 1)  foo(i, a, b);
```

Both examples express that the function call `foo(i, a, b)` is iterated over the specified range in any order, possibly concurrently. When using the `parallel_for` macro the parallel application is immediately synced, whereas with `create` it can run concurrently with other code in the caller.

When there are no data races, a `create` construct is functionally equivalent to the C code obtained by replacing the construct with a block expression containing an *undeterministic* C `for` loop, as follows:

```
_Spawn _SpawnIterSpace(_Linear(i, range...))
              foo(i, args...)

->
   ({
      long i;
      for (i = range... /* undeterministic order */)
          foo(i, args...);
   })
```

The work designated by such a construct is also named a *family* in reference to previous work with SVP.

4.7.2 Replication with optional interruption

Summary Here we allow a replication to be interrupted non-deterministically by the spawned work.

A. We specify that the following definition is present in `svp_syntax_aliases.h`:

```
#define interruptible      _Interruptible
```

B. We introduce the following spawn specifier:

spawn-specifier:

```
...
_Interruptible
```

This specifier is only valid in combination with parallel replication.

When `interruptible` is specified, the return value of the function call during parallel replication is not ignored, and is used as follows:

- the return type of the function must be convertible to `_Bool`;
- if any of the function calls returns a non-zero value, then when the spawn expression completes, evaluation the function call may not have been applied to some values in the range, non-deterministically;

The type of an `interruptible` spawn expression is `_Bool async`. When passed through `sync`, it returns 0 if all function calls returned value 0, and it returns 1 otherwise.

Here are some examples:

```
parallel_for(i, 0, 10, 1) interruptible foo(i, a, b);
```

```
int b = parallel_for(i, 0, 10, 1) interruptible foo(i, a, b);
if (b != 0)
   printf("interrupted!\n");
```

When there are no data races, an `interruptible` spawn is functionally equivalent to the C code obtained by replacing the spawn with a block expression containing an undeterministic C `for` loop, as follows:

```
_Spawn _SpanIterSpace(_Linear(i, range...)) _Interruptible foo(i, arg

->
   ({
      bool _b = false;
      long i;
      for (i = range... /* undeterministic order */) {
          _b = foo(i, args...);
          if (_b != false) break;
      };
      _b; /* return break status to context */
   })
```

4.8 Non-deterministic evaluation order

Summary Non-determinism was introduced for schedules using `async` expressions in Basic asynchrony, then for application order of many calls to the same function with the Parallel replication. Here we introduce non-determinism for the evaluation order of heterogeneous expressions.

Here we would like to overcome a limitation of the C language: it is not possible to order expression evaluation in C without providing a total order.

For example: a program may need to write location `*p` *after* performing function calls `foo()` and `bar()`, but without any local order between the calls to `foo()` and `bar()`. It is desired to express this dependency such that a compiler is free to reorder the calls to `foo` and `bar` wrt each other but ensuring that the memory write is executed after both. In the base C language, it is not possible to express such a construct.

Note

One may argue here that evaluation order of function call arguments is not specified, and thus it is possible to make the memory write dependent on the two calls without order via a function call, e.g.:

```
inline void writemem(int *p, int value,
                     int a /* unused */, int b /* unused */)
{ *p = value; }

/* at point of use: */

   writemem(p, value, foo(), bar());
```

In this example the calls `foo()` and `bar()` are not mutually ordered but the memory write is dependent on both as the function arguments must be evaluated before the function body is run.
However there are two issues with this construct. The first is that the construct is cumbersome as it requires a different function definition for every context where such a dependency needs to be expressed. Also the function must take as explicit arguments all the dependencies of the dependent expression. The second issue is that while the C standard indicates that argument evaluation order is unspecified, most C compilers actually force an order (either right-to-left or left-to-right) and never exploit the opportunity for reordering.

A. We specify that the following definition is present in `svp_syntax_aliases.h`:

```
#define after    _After
#define discard  ((void)0) after
```

B. We extend the expression syntax as follows:

postfix-expression :

...
postfix-expression `_After (` *argument-expression-list* `)`

This construct is said to express a *dependent expression*. The evaluation of the expression on the left hand side is dependent on the evaluation of all expressions on the right hand side, *in any order*.

For example:

```
(*p = x) after (foo(), bar());
```

This expresses that the memory write on the left is dependent on the calls `foo()` and `bar()` but without specifying an order between

`foo()` and `bar()`. The expressions on the right hand side are the *dependencies* of the entire expression and are evaluated in a *non-deterministic* order at run-time.

Note

Care must be taken when manipulating `async` expressions at the right hand side of `after`. For example, withe following code:

```
foo() after (spawn bar());
```

the behavior is undefined because the result of `spawn` is discarded before either `sync` or `detach` is applied to it.

4.9 Asynchronous communication

4.9.1 Delayed argument passing

4.9.1.1 Example with spawn

Example:

```
int add(void) channels(in int x, in int y)
{
    return (consume x) + (consume y);
    /* functionally equivalent to:
       int a = consume x;
       int b = consume y;
       int c = a + b;
       return c;
    */
}
```

Point of use:

```
auto c = spawn add()
             waiting_for (in int x, in int y);

produce to(c) x = foo();
produce to(c) y = bar();

int result = sync c; // wait for sum
```

This forces the arguments to be computed in this order (first `x`, then `y`). By using `after` this order can be relaxed:

```
auto c = spawn add()
             waiting_for (in int x, in int y);
```

```
int result = sync c
          after (
              produce to(c) foo(),
              produce to(c) bar()
              );
```

New syntax:

- `in`: defines an input-only i-structure
- `waiting_for`: defines input i-structures for an async call
- `produce`: write i-structure
- `consume`: read i-structure

Equivalent functionally to C code where the function call is computed at the point of `sync`. When no real concurrency is present, the `waiting_for` definition implies a data structure that captures the produced values until `sync`.

4.9.1.2 Specification for channel declarations

A. We specify that the following definitions are present in `svp_syntax_aliases.h`:

```
#define in           _In
#define out          _Out
#define inout        _InOut
#define channels     _DeclChannels
#define waiting_for  _UseChannels
```

B. We extend the syntax of *function declarations* and *definitions* as follows:

direct-declarator:

...
direct-declarator `(` *parameter-type-list[opt]* `)` `_DeclChannels` `(` *channel-decl-list[opt]* `)`

channel-decl-list:

channel-decl
channel-decl `,` *channel-decl-list*

channel-decl:

channel-specifiers declaration-specifiers declarator
channel-specifiers declaration-specifiers abstract-declarator[opt]

channel-specifiers:

channel-direction

channel-direction:

`_In`
`_Out`
`_InOut`

C. We extend C's *derived types* with *function types with channels*. A function type with channels describes a function with specified return type, number of type of its parameter, and its *channel interface*. For example, in the following declaration:

```
int foo(int a) channels (in int c);
```

The object named `foo` is declared to have type `int ()(int) channels(in int)`.

Function types with channels are distinct from and incompatible with regular function types. In particular, they cannot be used with C's base function call syntax (i.e. they can only be used with `spawn`).

D. We extend C's *derived types* with *channel types*. A channel type describes a channel with specified value type and direction.

Channels establish a mean for the caller to communicate with the callee after the point of call through async.

> **Note**
> In the basic form, channels can only be communicated through once (single shot).

When a channel interface declaration appears in a function definition, each channel declarations in the interface declares the corresponding identifier in the block of the function body with channel type. For example:

```
void foo(void) channels (in int c)
{
    /* here "c" has channel type "in int" */

    {
        int c; /* this "c" shadows the function channel "c" */
    }
}
```

Function channel names and parameters are declared in the same scope, so in particular a channel cannot have the same name as a parameter.

E. We specify that the following definition is present in `svp_syntax_aliases.h`:

```
#define pending      _Pending
```

F. We add the *type specifier* `_Pending` to the language, to be used in combination with `_Async`. Its syntax is:

pointer:

...
`_Async _Pending (` *channel-decl-list[opt]* `)`
`_Async _Pending (` *channel-decl-list[opt]* `)`
pointer

G. We extend C's *derived types* with *async types with channels*. An async type with channels is bound to an asynchronous computation returning a given return type, and using a channel interface. For example:

```
float async pending(out int c) a = ...;
```

the object `a` is declared to have type `float async pending(out int)`.

Async types with channels are distinct from and incompatible with base async types. Two async types with channels are compatible if their channel interfaces are compatible and their synchronized types are compatible. An async type with channels using an incomplete channel type is also incomplete.

In an async type with channels, the names given to each channel are part of the type, like field names are part of C's aggregate types. (However, unlike C's aggregate types, async types with channel use structural equivalence instead of name equivalence.)

H. We extend the spawn syntax:

spawn-expression:

`_Spawn` *spawn-specifier-list[opt] postfix-expression*
`(` *argument-expression-list-opt* `)` *channel-use[opt]*

channel-use:

`_UseChannels (` *channel-decl-list[opt]* `)`

The syntax reads as follows: after the function call syntax in a spawn there can be an additional channel interface declaration.

When `waiting_for` is used, it causes the entire spawn construct to have an async type with channels instead of a regular async type. In

the resulting async type, the channel directions are *reversed* from the function declaration. Here is an example:

```
int foo(void);
int bar(void) channels(in int);

auto a = spawn foo();
auto b = spawn bar() waiting_for(in int x);
```

In this example, object `a` has type `async int`, whereas `b` has type `async pending(out int x) int`. For the second construct the explicit syntax would be:

```
int async pending(out int x) b = spawn bar()
                                 waiting_for (in int);
```

The call to function `bar` defines an input channel for `bar` which is visible as an output channel in the caller.

Note

Because a base async type and an async type with channels are not compatible, it is not possible to mix and match on both sides of the `otherwise` operator.

Note

It is invalid to use `here` (see Pseudo-asynchrony above) with `waiting_for`.

4.9.1.3 Reading from channels, in callee

A. We specify that the following definition is present in `svp_syntax_aliases.h`:

```
#define consume      _Consume
```

B. We add the polymorphic primitive expression `_Consume` to the language:

consume-expression:

`_Consume` *unary-expression*

unary-expression:

...
consume-expression

This reads as follows: `consume` followed by an expression. The entire construct is an expression. For example:

```
int foo(void) channels(in int x)
{
     int a = consume x; /* consumes from foo's channel "x" */

     return a + 1;
}
```

This expresses reading from the designated input channel.

4.9.1.4 Writing to channels, in callee

A. We specify that the following definition is present in `svp_syntax_aliases.h`:

```
#define produce      _Produce
```

B. We add the polymorphic primary expression `_Produce` to the language:

produce-expression:

`_Produce` *unary-expression "=" *assignment-expression*

unary-expression:

...
produce-expression

This reads as follows: the word `_Produce` followed by an expression, the assignment operator and another expression. For example:

```
void foo(int x) channels(out int y)
{
     produce y = x + 1;
}
```

The first expression must designate an output channel. This expresses communication of the value of the second expression to the channel.

Like with C assignments, the entire expression evaluates to the righ hand side, as a rvalue.

4.9.1.5 Writing to channels, in caller

A. We specify that the following definition is present in `svp_syntax_aliases.h`:

```
#define to      _Peer
```

B. We extend the syntax of `_Produce` as follows:

produce-expression:

...
`_Produce _Peer (` *assignment-expression* `)`
identifier = *assignment-expression*

This describes constructs of the form `produce to(EXP) ID = EXP`. The first operand must evaluate to a value of type `async pending`, and the second operand must be a valid channel name in that type.

The construct describes providing the value of the third operand to the designated channel of a `async pending` call.

For example:

```
int async pending(out int x) a = spawn foo(...) waiting_for(in int);

produce to(a) x = 123;

/* same as: */

auto a = spawn foo(...) waiting_for(in int x);

produce to(a) x = 123;
```

Note

Using `sync` on a `async pending` call without providing the channel input values required by the functions has undefined runtime behavior. In particular, deadlock can occur but is not guaranteed; the function may compute using invalid channel inputs instead.

4.9.1.6 Reading from channels, in caller

A. We specify that the following definition is present in `svp_syntax_aliases.h`:

```
#define from      _Peer
```

B. We extend the syntax of `_Consume` as follows:

produce-expression:

...
`_Consume _Peer (` *assignment-expression* `)`
identifier

This describes constructs of the form `consume from(EXP) ID`. The first operand must evaluate to a value of type `async pending`, and the second operand must be a valid channel name in that type.

The construct expresses reading from the designated input channel.

For example:

```
void foo(void) channels(out int);

auto a = spawn foo() waiting_for(out int x);

sync a;
int y = consume from(a) x;

/* the two last lines can be re-expressed as: */

int y = (consume from(a) x) after (sync a);
```

Note

Consuming from channels on an `async pending` call before using `sync` has undefined runtime behavior. In particular, deadlock can occur or the construct may evaluate to an undefined value.

4.9.1.7 Local multi-way communication with `hold`

When both `here` (`_LocalContext`) and `waiting_for` are used, the spawn creates a local placeholder for channel values with type `async pending`:

- the function, call argument values, and values produced to `out` channels (`in` in the `async pending` type) are preserved until the `sync`;
- at the point of `sync`, the function is called and the preserved values are provided to the `in` channels within the function;
- after the sync, the values produced during the function calls can be consumed from the `out` channel in the parent.

For example:

```
void add(void) channels(inout int xs, in int y)
{
  produce xs = consume xs + consume y;
}
```

```
/* at point of use: */

auto f = hold add() waiting_for(inout int a, in int b);
produce to(f) a = 123; // this reserves 123 for the call below
produce to(f) b = 456; // this reserves 456 for the call below
sync f; // call add(), make both values visible during call
int s = consume from(f) a; // read value set during the call to add
```

Importantly, the function arguments are evaluated during the spawn and not at the point of `sync`, for example:

```
void foo(int x) channels();
int x = 10;

void async f = hold foo(x + 1) waiting_for();
x = 11;
sync f;

/* is equivalent to: */

int t = x + 1;
x = 11;
foo(t);
```

Similarly, if the function being called is referenced to by an expression (which produces a function pointer), this expression is evaluated during the spawn and reserved until the point of `sync`. For example:

```
void (*ftable)(void) channels();

int x = 1;
void async f = hold ftable[x]() waiting_for();
x = 2;
sync f; // calls ftable[1], not ftable[2]
```

4.9.2 Mixing channels with parallel replication

A. We extend the semantics of parallel replication as follows: all `in` channels in the callee are connected to a single `out` channel in the caller. A value provided by the caller is communicated to every call in the parallel replication.

   For example:

```
void copy(int idx) channels(in  int *a, in  int *b)
{
```

```
    (consume a) [idx] = (consume b) [idx];
}

/* at point of use: */

int A[N];

auto f = create (i, 0, N, 1)  copy(i)
             waiting_for (in  int *a, in  int *b);


produce to(f) a = A;
produce to(f) b = B;

sync f;
```

In this example, each call to `copy` receives the pointers to `A` and `B`. Only the value of the formal parameter `idx` differs.

B. We extend the semantics of parallel replication as follows: The "in-out" channel of the first call is connected to an "out" channel in the caller usable before `sync`. For the second call onwards, each "inout" channel is connected to the corresponding channel of the previous call. The "inout" channels of the last call in the sequence are connected to "in" channels in the caller consumable after `sync`.

   For example:

```
void cumul (int idx, int *a) channels(inout  int sum)
{
  produce sum = a [idx] + (consume sum);
}

/* at point of use: */

int A[N];

auto f = spawn_parallel_apply (i, 0, N, 1)
               cumul(i, A)
                 waiting_for (inout  int sum);

produce to(f) sum = 0;

auto t = sync f;
```

```
int x = consume from(t) x;
```

In this example, each call to `cumul` consumes the value produced by the previous call. The caller produces the value consumed by the first call before the `sync`, and consumes the value produced by the last call.

4.10 Resource management

4.10.1 Local environment

Summary Here we extend the language to capture the fact programs run within concurrent environments.

As described in **??**, any instance of a function execution has two environment boundaries:

- the local, actual (virtual) unique processor where this specific function call instance is executing;
- the logical (virtual) processor *group* that was reserved by the caller for this work.

We call the former the *local processor* and the latter the *default place*.

> **Note**
> These two resources can differ as follows: when delegation designates a processor or processor group without specifying a change on the default place, the default place is propagated from the caller to the callee, whereas the local processor can change implicitly for every spawn.

For the purpose of use in the delegation primitives introduced later, we extend the language as follows.

A. We specify that the following definition is present in `svp_syntax_aliases.h`:

```
/* human namespace aliases: */

#define PLACE_DEFAULT   _DefaultPlace
#define PLACE_LOCAL     _LocalPlace
```

B. We specify that the identifiers `_DefaultPlace` and `_LocalPlace` pre-exist in the environment and evaluate dynamically to values suitable for use with the delegation syntax introduced above.

   The meaning of these pre-defined names are as follows: when they are evaluated, their value designates a valid place identifier for the default place and local processor, respectively.

These identifiers evaluate to rvalues, in particular they have no address and cannot be assigned to. They can only be set for a callee by the caller during delegation, using mechanisms described below.

Unless specified otherwise by a program, the default place is propagated implicitly from the caller to the callee, and the local place/processor changes implicitly according to the actual scheduling performed.

4.10.2 Delegation

Summary Here we introduce the ability to perform function calls at a designated resource.

A. We specify that the following definitions are present in `svp_syntax_aliases.h`:

```
/* human namespace aliases: */

#define map(Placement)  _SpawnMap(Placement)
#define at(P)           _AtPlace(P)

/* syntactic sugar: */

#define map_at(P)       map(at(P))
```

B. We add some spawn specifiers to the language:

spawn-specifier:
    ...
    placement-specifier
placement-specifier:
    `_SpawnMap (` *map-specifier-list* `)`
map-specifier-list:
    map-specifier
    map-specifier map-specifier-list
map-specifier:
    `_AtPlace (` *expression* `)`

This syntax reads as follows. The list of spawn specifiers can contain a construct of the form `_SpawnMap(...)`. Between the parentheses there can be a *list of mapping specifiers*. One such mapping specifier is a construct of the form `_AtPlace(P)`, where `P` is an expression. It can be used as follows:

```
spawn map_at(P) foo(...)

/* this expands to: _Spawn _SpawnMap(_AtPlace(P)) foo(...) */
```

The meaning of this optional placement syntax is to *place* the computation at the designated resource. How resource identifiers are computed is not discussed here. The placement syntax will be extended in further steps below.

This construct is ignored if `here` or `here_late` is also specified. However the place expression is still evaluated for side effects.

Specifying placement has no effect on the functional behavior of the program.

When `_AtPlace(P)` is specified, the default place in each function call instance becomes `P`, i.e. `P` is propagated implictly to the callee's `_DefaultPlace`.

4.10.3 Asynchrony with fault tolerance

Summary Here we extend the previous mechanisms to allow checking for success/failure of asynchronous computations and provide alternatives.

A. We specify that the following definition is present in `svp_syntax_aliases.h`:

```
#define otherwise  _Otherwise
```

B. We extend the spawn syntax as follows:

async-def-expression-list :

async-def-expression
async-def-expression `_Otherwise` *async-def-expression-list*

assignment-expression :

...
async-def-expression-list

This operator relates to the lazy evaluation syntax of C's `&&` and as follows: when it is reached during execution, the left hand side spawn is attempted first. Only if this fails, the right hand side spawn is computed instead. Any side effects expressed at the right hand side of `otherwise` are not performed if the left hand side succeeds. Both sides of `otherwise` must have the same `async` type.

Here is an example use:

```
int async x = spawn map_at(P1) foo(12)
                 otherwise
              spawn here bar(45);

...
```

```
int y = sync x;
```

In this example, a spawn of `foo` is attempted at place `P1`. If this fails, a call to `bar` is computed inline and its value is reserved for later. Which value is computed is eventually revealed by `sync`. From a functional perspective, this example is equivalent to the C code obtained by removing all right hand sides of `otherwise` and removing the remaining µTC syntax:

```
int x = foo(12); // otherwise right branch removed
...
int y = x;
```

C. We specify that the following definitions are available in `svp_syntax_aliases.h`:

```
#define synctest        _SyncTest
#define valid(X)        (0 == (X))
#define undefined(X)    (0 != (X))
```

D. We add the polymorphic primitive expression `_SyncTest` to the language:

synctest-expression :

`_SyncTest` *unary-expression*

unary-expression :

...

synctest-expression

This is applied to an expression with an `async` type as input, and returns an integer value.

The meaning of `synctest` is to wait on completion of the asynchronous computation to which it is applied, and indicte whether the computation was successful. It returns 0 only if the operation was successful. The value can then be retrieved with `sync` as usual. For example:

```
int async x = spawn foo(12);

...

int y;
if(valid(synctest x))
    y = sync x;
else
    y = ... ; // put substitute here, e.g. y = sync spawn bar
```

When there is no failure, this is functionally equivalent to the C code where `_SyncTest X` is replaced by 0, and the other µTC constructs are removed, e.g.:

```
int x = foo(12);
...
int y;
if (0 == 0)
   y = x;
else .... /* irrelevant because 0 == 0 */
```

4.10.4 Distribution control in replication

Summary Here we allow a program to control how parallel replication is spread over a target resource during delegation.

A. We specify that the following definitions are present in `svp_syntax_aliases.h`:

```
/* human namespace aliases: */

#define spread(N)                _Spread(N)

/* syntactic sugar: */

#define map_spread(N)                map(spread(N))
#define map_spread_at(P, N)          map(at(P) spread(N))
```

B. We add some new map specifiers to the spawn specifier syntax:

map-specifier:

`_Spread (` *assignment-expression* `)`

This syntax reads as follows: inside the `_SpawnMap` syntax, a construct of the form `_Spread(EXP)` can appear.

The meaning of this optional control is to control how parallel execution is deployed onto the target resource. The parameter to `spread` describes a "blocking factor", ie. a hint to maximum chunking factor for the mapping of concurrency to processing agents at the target resource.

This construct is ignored if `_LocalContext` is also specified. However the spread expression is still evaluated for side effects.

4.10.5 Resource partitioning via restriction on the default place

Summary Here we introduce a rough mechanism to allow a program to partition resources during delegation, by restricting the default place of the spawned work.

A. We specify that the following definitions are present in `svp_syntax_aliases.h`:

```
/* human namespace alias: */

#define narrow(N)                _Narrow(N)

/* syntactic sugar: */

#define map_distribute(N, M)         map(spread(N) narrow(M))

#define map_distribute_at(P, N, M)  map(at(P) spread(N) narrow
```

B. We add some new map specifier to the spawn specifier syntax:

map-specifier:

`_Narrow (` *assignment-expression* `)`

This syntax reads as follows: inside the `_SpawnMap` syntax, a construct of the form `_Narrow(EXP)` can appear.

The purpose of this optional control is to implictly segregate each function call instance onto disjoint subsets of the computing resources.

Its meaning is to cause the default place of the spawned work to be computed as follows. We use the following definition:

> The *designated target place*, is where the entire delegation takes place; either explicitly given with `_AtPlace(P)` or implicitly the caller's `_DefaultPlace`.

When `narrow(N)` is specified, then for each function call instance in the spawned work, given the specific instance's `_LocalPlace`, the new value for `_DefaultPlace` becomes the subset of the designated target place that is rooted `N` levels up from the new `_LocalPlace`.

For example, if the designated target is structured as follows:

```
          root

          ....

      c0      c1

  p0   p1    p2 p3
```

Then `_Narrow(1)` will cause the default place in every call instance running on `p0` and `p1` to be the sub-tree rooted at `c0`, and the default place of every instance running on `p2` and `p3` to be the sub-tree rooted at `c1`.

Note
The behavior is left unspecified if `N` is too large or negative. This might be refined later.

This construct is ignored if `_LocalContext` is also specified. However the argument expression is still evaluated for side effects.

4.11 Mutual exclusion

A. We specify that the following definitions are present in `svp_syntax_aliases.h`:

```
/* human namespace alias: */
```

#define exclusive_at(P) _ExclusiveAt(P)

B. We add some spawn specifiers to the language:

spawn-specifier:

...
exclusive-specifier

exclusive-specifier:

`_ExclusiveAt (` *expression* `)`

The syntax reads as follows. The list of spawn specifiers can contain a construct of the form `_Exclusive(P)`, where `P` is an expression. It can be used as follows:

```
spawn exclusive_at(P) foo(...)

/* this expands to: _Spawn _ExclusiveAt(P) foo(...) */
```

The parameter to `_ExclusiveAt` is the *target exclusive place*.

This meaning of this optional syntax is to indicate that the computation cannot execute concurrently with any other computation placed at the same target exclusive place.

The `_Exclusive` specifier cannot be used in combination with `_SpawnMap`.

4.12 Expressing previous µTC semantics using the new paradigm

4.12.1 Thread functions

- thread functions are converted to regular functions with return type `_Bool`

- uses of thread `break` are replaced by `return 1`
- uses of `return` in a thread function are replaced by `return 0`
- an implicit `return 0` is inserted at the end of the control flow for each thread function body
- for each thread function a `long` parameter is inserted at the start of the parameter list, and the thread index variable is initialized to that parameter
- global thread parameter definitions are replaced by an "`in`" channel declaration
- shared thread parameter definitions are replaced by an `inout` channel declaration
- each read-use of a thread parameter `p` is replaced by a `consume p` construct;
- each write-use of a thread parameter `p` is replaced by a `produce p` construct.

For example, the following μTC code:

```
thread fibo(shared int p, shared int p2, /*global*/ int* fib)
{
   index i;
   int n = p + p2;
   p2 = p;
   p = n;
   fib[i] = n;
}
```

would lower to:

```
void fibo(long _I) channels(inout int p, inout int p2, in int* fi
{
   long i = _I;
   int n = consume p + consume p2;
   produce p2 = consume p;
   produce p = n;
   (consume fib)[i] = n;
}
```

4.12.2 Create / Sync

Constructs of the form:

```
create(fid; P; start; limit; step; block)
       fun ( args ...)
```

would lower to:

```
auto fid = create(i, start, limit, step) /* range */
             map_spread(P, block)  /* blocking factor */
                fun (i)  /* application */
                   waiting_for (chans...);  /* i-structure list */

produce to(fid) a = ...;
produce to(fid) b = ...; /* etc */
```

Then µTC's `sync` is converted to either the new `sync` or `synctest` depending on whether the family exit status is used.

4.12.3 Re-expressing deadlock prevention with the new semantics

Constructs of the form:

```
create(fid; P; start; limit; step; block)
       fun ( args ...)
```

would lower to:

```
auto fid = create(i, start, limit, step) /* range */
                  map_spread(P, block)  /* blocking factor */
                    fun (i);  /* application */
                      waiting_for (chans...)
            otherwise
               create(i, start, limit, step) /* range */
                  here
                     fun (i);  /* application */
                       waiting_for (chans...)

produce to(fid) a = ..., to(fid) b = ...; // etc
```

When the first branch of `otherwise` fails, `here` (`_LocalContext`, a.k.a. `hold`) causes the 2nd spawn to capture all the data needed for the work and delay the computation until the point of `sync`.

4.13 Memory consistency

We recognize two classes of functions (with or without channels) when used in a spawn expression:

- functions with *known dependencies*: the set of all objects accessed by the function are indicated somehow;
- functions with *unknown dependencies*: some dependencies are unknown or incompletely characterized.

Any function is considered to have unknown dependencies at the point of spawn, unless either of the following is true:

- the function *definition* is visible at the point of spawn, and static analysis can derive all the function dependencies; or
- the function has only fully determined argument and channel types (i.e. no pointers or aggregates containing pointers) and is declared to be *enclosed*.

> **Note**
> Aggregates containing arrays are determined types for the purpose of this condition. However, as arguments and channel values are copied when communicated to the callee, this is inefficient to pass large objects. Syntax to pass large determined types by reference will be provided later.

> **Note**
> Syntax to express "enclosedness" will be provided later; the purpose of this annotation is a programmer-provided guarantee that the function does not access global variables or have side-effects on memory that may be visible to the caller.

Knowledge about function dependencies impacts the implementation of spawn as follows:

- if an explicit target place is indicated (e.g. with `map_at`) for delegation, this is used always. If the target place lies in a different *consistency domain* from the caller's local place, the behavior of the program becomes undefined.
- otherwise (no explicit target place is indicated):
  - if a function has unknown dependencies, or if the `memlocal` spawn specifier is expressed, then the spawn will force the selection of a target place within the same consistency domain as the caller's local place; otherwise

– if all dependencies are known and `memlocal` is not expressed, then the spawn may select a target place in a different consistency domain from the caller's local place. Additionally, parallel replication can span multiple consistency domains.

4.14 Optional compatibility with popular non-standard C extensions

4.14.1 Extending GNU's block expressions with spawn

There is a GNU C extension called "expression blocks". This has the syntax `({ ... })`. It can contain declarations and statements but is evaluated like an expression: the type and value of the entire block expression is the type and value of the last expression evaluated inside the block. For example:

```
int x = ({ int y = 21; y + y; });
```

In this example the block expression evaluates to the `int` value `42`.
We could extend this syntax as follows:

A. if the last expression of a block expression has type `async`, then the construct is valid and the entire block expression evaluates to this async expression. For example:

```
int async x = ({ .....; E; }); // valid if E has type int async.
```

B. if the last expression of a block expression is a `spawn`, then the block expression is a valid operand for the `otherwise` operator. For example:

```
async int x =
       ({ P1 = ...; spawn map_at(P1) foo(123); })
     otherwise
       ({ P2 = ...; spawn map_at(P2) bar(456); })
     otherwise
       spawn here baz(678);
```

In this example, a spawn of a call to `foo` is attempted at place `P1`. If this fails, `P2` is computed an a spawn of a call to `bar` is attempted at `P2`. If this fails, `baz` is called in place and its value is reserved for the later `sync`.

Such block expressions can be used e.g. to avoid evaluating the place identifier for a right hand side to `otherwise`

4.14.2 Anonymous closures with Apple's blocks and C++0x `lambda`

In 2010 Apple has introduced the technology Grand Central Dispatch, which includes a conceptual extension to the C language: "blocks". These define anonymous closures that capture dependencies implicitly from the point they are defined and can receive additional arguments at their point of use. Here is an example:

```
dispatch_apply(count, dispatch_get_global_queue(0, 0), ^(size_t i
   results[i] = do_work(data, i);
  });
total = summarize(results, count);
```

In this example, the 3rd argument to `dispatch_apply()` is a closure with a pending argument of type `size_t`.

Similarly the upcoming C++ standard defines anonymous closures:

```
std::vector<int> some_list;
int total = 0;
int value = 5;
std::for_each(some_list.begin(), some_list.end(), [&, value](int
                total += x * value * this->some_func();
   });
```

In this example, the construct `[&, value](int x) { ... }` defines a closure with a pending argument of type `int`, that captures all its dependencies by reference except `value` which is captured by value.

As a possible extension to our SVP language, we could support dispatching GCD blocks and/or C++-like closures with `spawn`.

4.15 Summary of language extensions

4.15.1 Summary of constructs for basic asynchrony

Alias	Keyword	Position in syntax	Described in
`auto`	`_Auto`	*declaration*	Limited implicit typing
`async`	`_Async`	*declarator*	Split-phase asynchrony
`sync`	`_Sync`	*unary-expression*	Split-phase asynchrony
`detach`	`_Detach`	*unary-expression*	Split-phase asynchrony
`spawn`	`_Spawn`	*unary-expression*	Asynchronous function calls
`here`	`_LocalContext`	*spawn-specifier*	Pseudo-asynchrony
`after`	`_After`	*postfix-expression*	Non-deterministic evaluation order

Syntactic sugar	Expansion
`discard`	`(void)0 after`
`hold`	`spawn here`

4.15.2 Summary of parallel replication constructs

Alias	Keyword	Position in syntax	Described in
`ispace`	`_SpawnIterSpace`	*spawn-specifier*	Parallel replication
`linear`	`_Linear`	*index-space-form*	Parallel replication
`interruptible`	`_Interruptible`	*spawn-specifier*	Replication with optional interruption

Syntactic sugar	Expansion
`create(...)`	`spawn ispace(linear(...))`
`parallel_for(...)`	`sync create(...)`

4.15.3 Summary of channel-based communication

Alias	Keyword	Position in syntax	Described in
`channels`	`_DeclChannels`	*direct-declarator*	Specification for channel declarations
`in`	`_In`	*channel-direction*	Specification for channel declarations
`out`	`_Out`	*channel-direction*	Specification for channel declarations
`inout`	`_InOut`	*channel-direction*	Specification for channel declarations
`pending`	`_Pending`	*type-specifier*	Specification for channel declarations

... continued on next page

Alias	Keyword	Position in syntax	Described in
`waiting_for`	`_UseChannels`	*spawn-expression*	Specification for channel declarations
`consume`	`_Consume`	*unary-expression*	Reading from channels, in callee, Reading from channels, in caller
`produce`	`_Produce`	*unary-expression*	Writing to channels, in callee, Writing to channels, in caller
`from`, `to`	`_Peer`	*produce-expression*, *consume-expression*	Writing to channels, in caller, Reading from channels, in caller

4.15.4 Summary of resource management constructs

Alias	Keyword	Position in syntax	Described in
`map`	`_SpawnMap`	*spawn-specifier*	Delegation
`at`	`_AtPlace`	*map-specifier*	Delegation
`otherwise`	`_Otherwise`	*async-def-expression*	Asynchrony with fault tolerance
`synctest`	`_SyncTest`	*unary-expression*	Asynchrony with fault tolerance
`spread`	`_Spread`	*map-specifier*	Distribution control in replication
`narrow`	`_Narrow`	*map-specifier*	Resource partitioning via restriction on the default place
`exclusive_at`	`_ExclusiveAt`	*spawn-specifier*	Mutual exclusion

Syntactic sugar	Expansion
`map_at(...)`	`map(at(...))`
`valid(...)`	`(0 == (...))`
`undefined(...)`	`(0 != (...))`
`map_spread(...)`	`map(spread(...))`

... continued on next page

Syntactic sugar	**Expansion**
`map_spread_at(...)`	`map(at(...) spread(...))`
`map_distribute(...)`	`map(spread(...) narrow(...))`
`map_distribute_at(...)`	`map(at(...) spread(...) narrow(...))`

Chapter 5

SL annotations for extra-functional properties and requirements

Key: adv14

Status: Draft

Date: 2010-11-15

Authors: Raphael 'kena' Poss

Source: http://notes.svp-home.org/adv14.html

Version: adv14.txt 4148 2010-11-16 13:13:33Z kena

Abstract: In this note, we propose to extend the SL language as introduced in **??** with a system where function definitions and use points can be instrumented with arbitrary annotations usable by a run-time system.

5.1 Introduction

In ADVANCE there are two use cases for extra-functional properties and requirements:

- during run-time behavior analysis, they are used to construct a TLJ model;
- during run-time concurrency management, they are used to facilitate/optimize placement of computations to execution resources (hardware cores / interconnects).

As the name implies, extra-functional properties and requirements encompass information that cannot be derived from the functional specification of work units. In our setting (see **??**), the unit of work is defined by *families* of threads.

We are interested in the following, among others:

- *properties*:
  - physical *size* (bytes/words/etc...) of the data consumed and produced by a family, as this is required to define the throughput of a computation;
  - the *replay factor* of the family, which estimates how often the algorithm re-accesses its input data (e.g. in multi-pass code); this is defined for each input item and is relevant for modelling latency (number of steps per input item), and throughput/jitter (impact of caching effects);
  - the *locality factor* of the family, which estimate the number of different memory locations accessed by individual steps of the computation; this impacts throughput/jitter (caching effects).

- *requirements*:
  - whether a family can *escape*, i.e. may request and use execution resources outside of its initial placement;
  - whether all internal and external data dependencies of a family are known, and the whether the family is *enclosed* (see **??**, "memory consistency" for a complete definition), i.e. whether the run-time system can assume that all communication is characterized by the external interface of the family (this information places constraints on placement of a family w.r.t. it parent thread).

We consider the points above to be only examples. We expect that the remainder of the project will reveal additional types of extra-functional properties and requirements which will need to be integrated.

Therefore, instead of proposing a fixed scheme to express extra-functional aspects with predefined semantics, we propose a generic and extensible scheme that integrates with our compiler technology but does not require future changes to the compiler after its initial implementation.

The scheme is articulated as follows:

- in the *source code* of programs, *annotations* can be expressed by means of the C "`#pragma`" syntax. However, instead of interpreting the annotation in the C/SL compiler, we delegate the annotation to an external *annotation system* implemented as an external, dynamically loadable/configurable compiler extension;

- during *execution*, points where families are defined can be *instrumented* to reflect on annotations and influence the execution or otherwise observe the behavior.

5.2 Static and dynamic aspects of extra-functional requirements and properties

Families are defined by a *function*, some *argument* values, *communication endpoints* and an optional *placement*. As such, a family can only be completely characterized by bringing together the following sources:

- available at *compile time*:
  - information about the *definition* of the function being run in the family; this is fully determined at the point the function is defined,
  - information about the argument *types* given at the point of use,
  - (optionally) information about the definition of the function enclosing the point of use;
- available at *run-time*:
  - information about the argument *values* given at the point of use,
  - information about *placement* and *communication endpoints*.

  Both of these latter points are fully determined only after the point of use is reached during execution.

For example, an input size is only known by looking at the input dynamically when a family creation point is reached during execution. Moreover, usually some other variable (argument) must be read to determine this input. This is why any system to describe extra-functional requirements and properties ultimately needs to be active during execution at the creation points.

5.3 Language-side specification

5.3.1 Identifying spawn points

A. We specify that the following definitions are present in `svp_syntax_aliases.h`:

```
#define  tag(Id)          _SpawnTag(Id)
#defined namedcall(Id)    sync spawn here tag(Id)
```

B. We add a spawn specifier to the language:

spawn-specifier:

...
`_SpawnTag (` *identifier* `)`

This syntax reads as follows: the list of spawn specifiers can contain a construt of the form `_SpawnTag(ID)`. For example:

```
int foo(int x) {
    int async a = spawn tag(first) bar(x);
    int async b = spawn tag(second) bar(x);
    return sync a + sync b;
}
```

Here is another example:

```
int fib(int n) {
    if (n < 1) return 1;

    return namedcall(p1) fib(n-1)
         + namedcall(p2) fib(n-2);
}
```

This construct allows to give an optional name to an occurrence of `spawn`. When it is used, the identifier denotes this specific point of `spawn` in the context of the function where `spawn` occurs.

A spawn identifier lies in a different name space than other C identifiers. Like C's label names, it has *function scope*: it can be used anywhere in the function where it appears, is declared implicitly by its syntactic appearance, and the same identifier cannot be used to name two distinct `spawn` constructs in the same function.

5.3.2 Program-specified annotations

A. We introduce a new pragma syntax to the language:

```
#pragma annotate_def(CSTR, CSTR {, ARG}...)
```

This syntax reads as follows: the phrase "`#pragma annotate_def(`" followed by a character string, a comma, another character string, then optionally a comma and a comma-separated argument list, and finally terminated by a closing parenthesis. Each argument in the argument list can have the syntactic form of an expression.

This denotes an annotation for a *function definition*.

The first character string specifies which *annotation system* will capture the annotation. This is described below. The second character string specifies the name of the function being annotated. A definition for this function must be present in the translation unit. The additional arguments can be identifiers or expressions and are provided to the selected annotation system as-is.

B. We introduce a new pragma syntax to the language:

```
#pragma annotate_use(CSTR, CSTR, CSTR, {, ARG}...)
```

This syntax reads as follows: the word `#pragma annotate_use(` followed by three character strings separate by commas, followed optionally by a comma and an optional comma-separated argument list, and finally terminated by a closing parenthesis. Like for `annotate_use`, Each argument in the argument list can have the syntactic form of an expression.

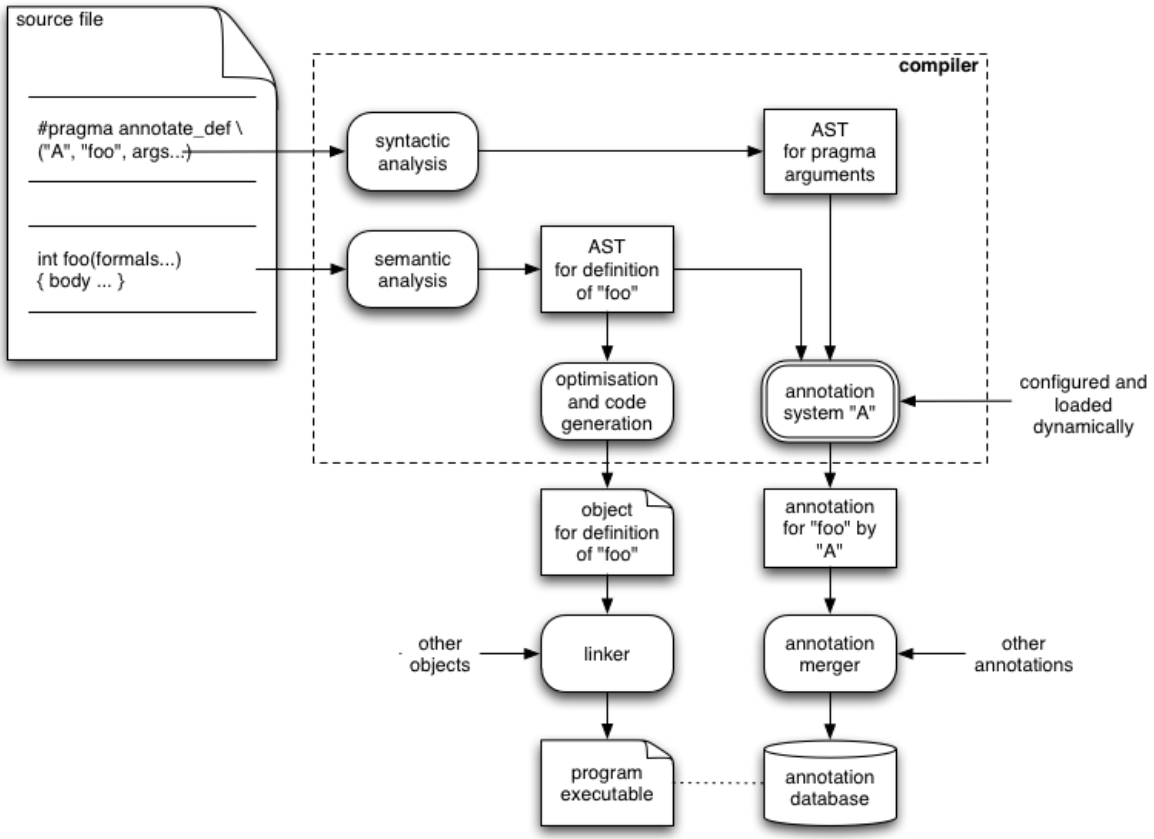
This denotes an annotation for a *function use*.

The first character string specifies the name of the annotation system to use. The second character string specifies the name of the function containing the use point. A definition for this function must be present in the translation unit. The third character string specifies the spawn tag being annotated. The specified function's body must contain a spawn tag definition with that name. The additional arguments can be identifiers or expressions and are provided to the annotation system as-is.

5.4 Compiler-side specification

5.4.1 Annotations for function definitions

The translation environment would support the following architecture:



After a translation unit goes through preprocessing, syntax and semantic analysis, an abstract syntax tree is constructed. For every `#pragma annotate_def`, an abstract syntax tree of the optional pragma argument list is constructed as well.

Then for every different annotation system name requested through `#pragma annotate_def`, the compiler should load an externally defined extension module containing an annotation service for definitions. This service defines a single function `annotate_def` which takes as input:

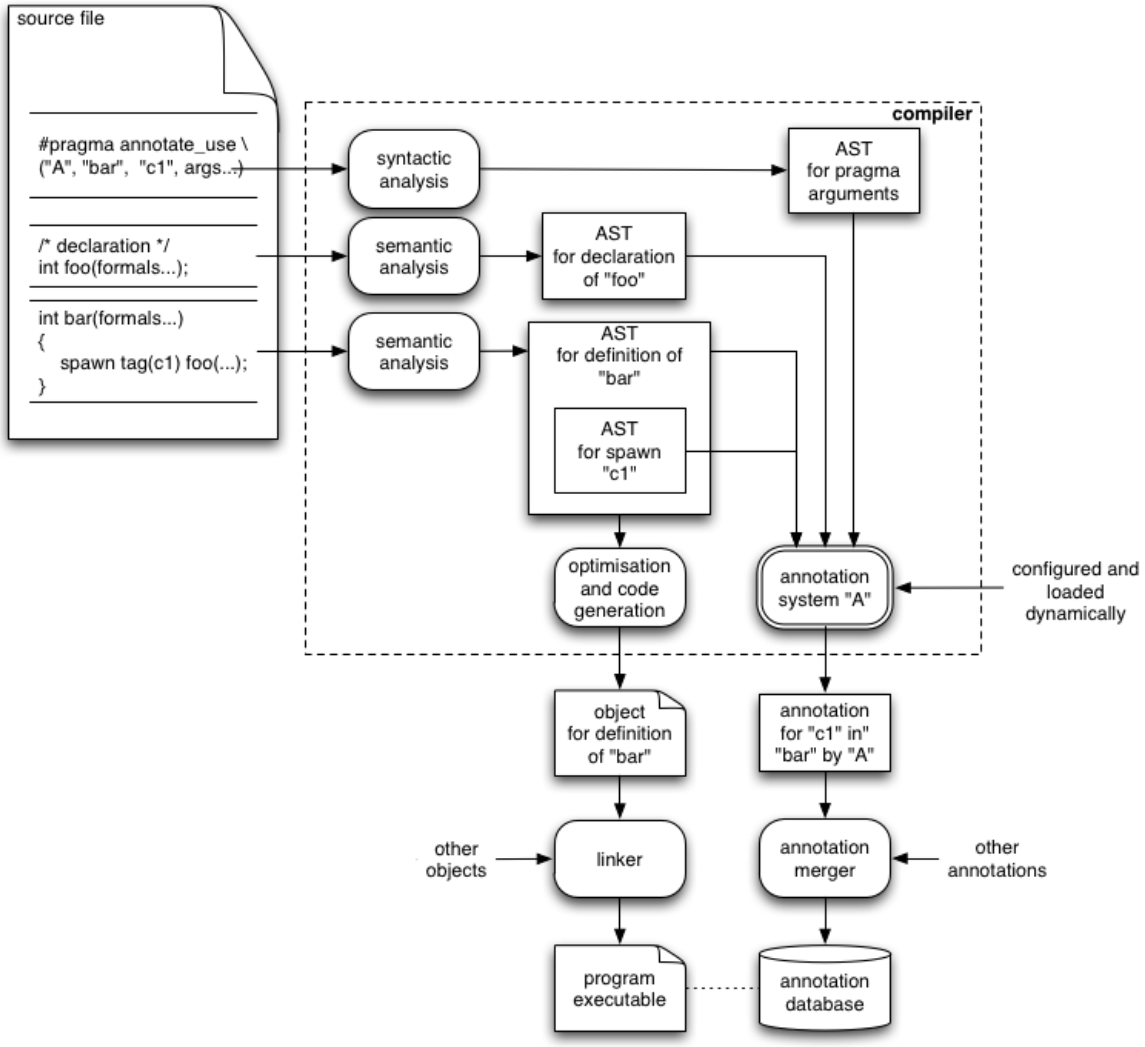
- the abstract tree for the designated function; and
- the abstract tree for the optional pragma arguments.

This function can then compute and optionally return an annotation which is preserved alongside the object code towards an annotation database for the program. By design, the annotation system cannot influence further translation of the definition.

(The functional semantics of a program should be preserved when all annotation pragmas are removed from the program source).

5.4.2 Annotations for function uses

The translation environment would support the following architecture:



Like for Annotations for function definitions above the system is enabled after semantic analysis. For every different annotation system name requested through `#pragma annotate_use`, the compiler should load an externally defined extension module containing an annotation service for uses. This service defines a single function `annotate_use` which takes as input:

- the abstract tree for the designed spawn; and
- the abstract tree of the function enclosing the spawn; and
- the abstract tree for the prototype declaration for the function used by the spawn; and

- the abstract tree for the optional pragma arguments.

This function can then compute and optionally return an annotation which is preserved alongside the object code. Again, the annotation system cannot influence further the translation.

5.4.3 Implementation notes

Our proposed implementation (see **??**) uses the CIL framework for reading and analyzing C code with language extensions. This framework is written using the functional language ML. It defines ML types for all abstract concept in the C language semantics.

To implement our annotation scheme, we propose that compiler extension modules that define annotation systems are written as ML modules. These would contain funcitons with the following interface:

```
val annotate_def:
    Cil.fundec * Cil.attribute
    -> string

val annotate_use:
    Cil.instr * Cil.fundec * Cil.varinfo * Cil.attribute
    -> string
```

For more information about CIL types, see http://hal.cs.berkeley.edu/cil/.

5.5 Propagating annotations from compilation to execution

We propose an interchange format and tool chain features that support communicating non-functional requirements between the compiler tools and the SVP middleware.

Instead of bundling the annotation data within the executable code itself, we propose to store the annotations in binary databases that will accompany executable modules. The proposed format is UNIX `dbm` (cf [**?**]), a simple yet efficient disk-based hash table implementation suitable for fast loading and access by readers. The proposed use is to have a different annotation database for each implementation of a given program component (there can be multiple implementations when the component is compiled for multiple targets). The dbm files would then need to be shipped along the executable formats to the execution platform, and would be loaded automatically by SVP as needed.

We justify the choice to store the annotations separately from the program code as follows:

1. We deem desirable to keep the ability to introduce annotations *after* compilation.

2. Not all executable formats on all platforms allow arbitrary interleaving of code with annotations within the executable file. Even for those formats that allow such interleaving the encoding of annotation and code sections differs between operating systems and binary formats.

3. There exists many tools to query and modify dbm files, while tools to query and modify executable files are scarce.

5.6 Run-time specification

In the proposed SVP implementation (**??**) we propose to integrate reflection on annotations. This will take the form of an API with the following services:

- query for annotations on function definitions. This takes the name of a function, looks up the annotation in the annotation database and caches the result for subsequent queries.

- query for annotations on function uses. This takes two forms:

  - observation of static annotations: this takes the name of a function and optionally the name of a use point, and looks up the annotation for this use point (or all use points in the function if no use point name is given). The result is cached for subsequent queries.
  - callback for dynamic introspection: this registers a service to the SVP run-time system for a given (set of) function(s) and use point(s), which is called dynamically whenever the specified use point(s) is(are) reached during execution. The callback function is given the static annotation as input, as well as run-time information about argument values, placement and communication endpoints.

The exact definition of this API will be refined as the partners express interest in using this interface.

Chapter 6

Approach to the description of hardware

Key: adv1

Authors: Sven-Bodo Scholtz, Stephan Herhut, Frank Penczek, Raymund Kirner, Clemens Grelck, Michiel W. van Tol, Chris Jesshope, Raphael 'kena' Poss

Status: Draft

Date: 2010-11-10

Version: adv1.txt 4129 2010-11-10 17:39:57Z kena

Source: http://notes.svp-home.org/adv1.html

Abstract: This note describes how physical hardware is described in terms of virtual resources in the ADVANCE "hardware virtualization layer".

6.1 Inventory of actual hardware

The ADVANCE project intends to target the following hardware:

- multicore systems (SMP, cache-coherent)

  Examples:

  - Niagara T2-based systems (8T x 8C x 1 CPU, visible as 64 virtual cores)
  - 1, 2 or 4-core places in a Microgrid

- multicore NUMA systems (optionally cache-coherent)

  Examples:

  - SunFire X4440 (4-core x 4 CPUs, visible as 16 virtual cores)

  - Intel's Single-Chip Cloud (2-core frequency islands x 4-island voltage clusters x 6 clusters, visible as 48 virtual cores)
  - 8-core or larger places in a Microgrid

- clustered networks of multicores

  Examples:

  - LISA cluster: 8-core x 2 CPUs x N nodes, mesh topology
  - multiple places in a Microgrid

- multi-core systems attached to one or more accelerators

  Examples:

  - GPGPUs (large data parallelism, e.g. CUDA-compatible hardware)
  - FPGAs (reconfigurable hardware)
  - Specialized I/O cores in a Microgrid

6.2 Overview of the approach

The goal of the approach is to abstract the properties of actual hardware while retaining control over the placement of data and computations over them, and letting programs introspect and filter at run-time the properties of the hardware they are mapped onto.

The overall approach can be summarized as follows:

- the actual hardware is *described* in terms of a tree of abstract nodes. The abstract nodes in this tree capture various *visible properties* of resources, especially *locality* between hardware components, the *programming models* at each level, and clustering of components that have *similar performance*. However, it does not necessarily describe the actual structure of the hardware.

  We call this tree the *virtual hardware description tree*, short name *hardware tree*.

- From this tree of virtual hardware, software can perform *selections*. Selections are sub-trees of the description tree that identify which resources are usable by software. Selections are described by a tree separate from the description tree; its node set is a sub-set of the node set of the description tree, and its edge set is a sub-set of the edge set of the description tree.

  We call such selection trees *virtual hardware places*, short name *virtual place*.

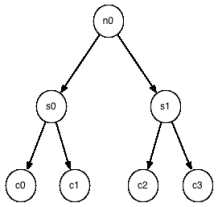
- When delegating software explicitly to a place in the SVP programming model, the following two items must be provided:

- the entire *selected place*, a reference to a virtual place (tree structure);
- the specific *execution place*, a reference to the leaf node of the selected place where execution actually occurs. This must be part of the selected place.

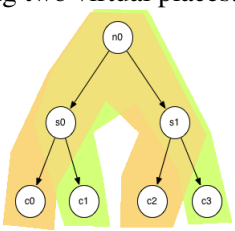
(Conversely, threads can introspect both their selected place and their execution place. The selected place can be further refined into smaller virtual places, and work can be delegated to other leaf nodes of the current selection.)

> **Note**
> How selections interact with the placement of threads during execution is the domain of *mapping*, which is outside of the scope of this note. This will be explored in a later note.

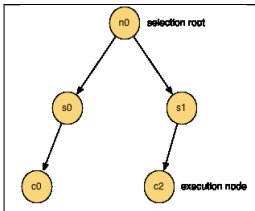
Here is an example. Starting from the following virtual hardware description tree:



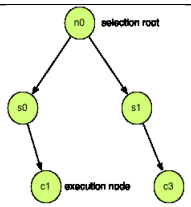
One can select the following two virtual places:



From this point, a thread A can be created at node `c2`, with the orange selection rooted at `n0`; simultaneously, a thread B can be created at node `c1` with the green selection which also happens to be rooted at `n0`.



Selection visible from thread A.



Selection visible from thread B.

While thread A is running, it can further refine its current selection to a virtual place rooted at `s0` and delegate work to `c0` (because both `c0`, and `c2` where it is running, are part of the entire selection where A was created). However, it cannot create a selection that would include `c1` or `c3`.

6.3 Virtual hardware description tree

The hardware tree is unique for a given collection of hardware resources. This tree is *visible* from programs, i.e. it can be introspected/queried by programs running within the system. However, it is not *transformable* by programs. Its purpose is to describe hardware resources: it might evolve dynamically due to changes in the hardware resources, but behaves as a dynamic, read-only data structure for programs running within the system.

The tree is composed of nodes that represent the *programmable structure* of the hardware system. Each node has a *type*. All nodes of a given type have the same *attributes* (set of keys for key/value annotations). There are two categories of node types:

- *processing elements* (PEs), independently addressable units on a delegation network where execution of threads can be carried out, and
- *logical grouping nodes*, which provide information about the structure of the hardware but do not carry out the execution of threads.

PEs can have internal concurrency; that is, multiple threads delegated to one PE might run simultaneously in time. However, internal concurrency within one PE cannot be independently addressed (i.e. one cannot choose which hardware thread of execution within a node executes a given software thread).

> **Note**
> For example, a Microgrid core forms one PE, with further 256 units of internal concurrency. A GPGPU core is such another PE.

PEs can be of either of two types:

- *general-purpose PEs*, where each unit of internal concurrency can carry out a separate programmable computation;
- *specialized PEs*, where each unit of internal concurrency might not be able to carry out separate programs (e.g. SPMD, GPGPU core), or where the programs are fixed (e.g. DSP, FPGA).

We specify that any specialized PE has at least one general-purpose PE as *host*. Delegation of work to a specialized node must always be issued by a thread running on one of its hosts, whereas delegation of work to general-purpose node can happen from any other GP node (restrictions due to *selections* apply, see below).

The other category of node types are "grouping nodes". We do not identify a specific hierarchy of grouping nodes at the SVP level; instead, the differents "kinds" of grouping nodes will be identified by annotations in the virtual hardware tree.

We envision the following concepts will be captured somehow:

- *SMP nodes*: a possibly-heterogeneous collection of GP nodes with possibly-non-uniform access to a single memory; however, the memory is cache-coherent with implicit consistency;

- *nc-SMP nodes*: a possibly-heterogeneous collection of SMP nodes accessing a single memory, with non-coherent caches and/or explicit memory consistency;

- *cluster nodes*: a possibly-heterogeneous collection of cluster, NUMA or SMP nodes, with distributed memory.

As stated initially, nodes are intended to virtualize hardware and provide uniform programming models, as well as locality (either of communication, caches, etc). Within a grouping node, the sub-nodes are ordered linearly and the order should be chosen so that nodes close to each other in this linear order should be local to each other. A "good" linear ordering in a 2D or 3D mesh can be obtained e.g. using a Hamiltonian path. If a network topology does not suggest a trivial ordering (e.g. it is not connected), this might suggest a different hardware tree structure where locality is represented by sub-trees.

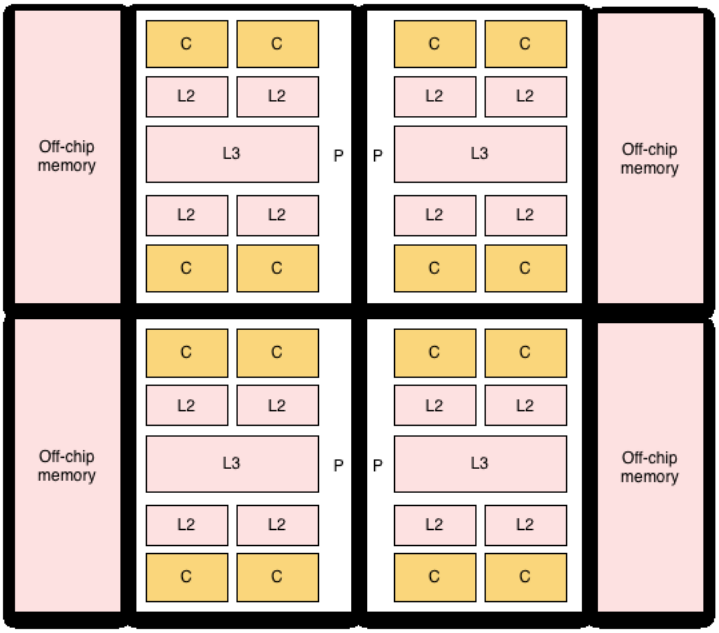
Finally, grouping nodes should also indicate clusters of run-time performance. Within a grouping node, the sub-node should be of similar concurrency granularity and offer similar throughput and latencies in terms of speed and interconnects. Again, if a hardware structure does not suggest homogeneity, multiple grouping nodes should be used to represent sub-clusters where homogeneity can be obtained.

6.3.1 Examples

In this section we illustrate how specific hardware configurations can be described with the proposed system.

6.3.1.1 Small, flat multi-cores

Consider the following schema, describing (approximately) the SunFire X4440 hardware from Sun:

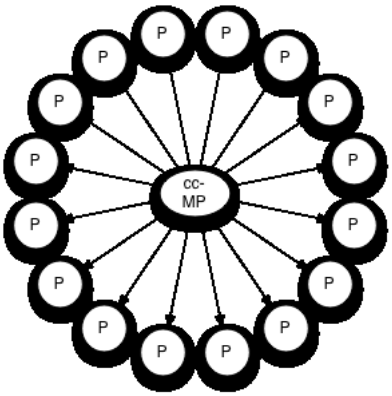


The `cassandra` server at UvA is an instance of this configuration.

It consists of 4 AMD Opteron processors, each with its own memory controller and L3 cache, connected together on a shared memory bus implementing the MOESI coherency protocol (this implements global cache coherency transparently from programs). Each processor contains 4 x85-64 cores, each with its own FPU and L1/L2 caches. Frequency can be scaled independently for each processor down to 50% of the maximum.

Each of the 16 cores is visible as an individual, addressable processor by the operating system.

From the point of view of programs, this system has 16 units of effectively programmable concurrency; which are symmetric with regard to access to memory. A straightforward representation in our description system could be thus as follows:

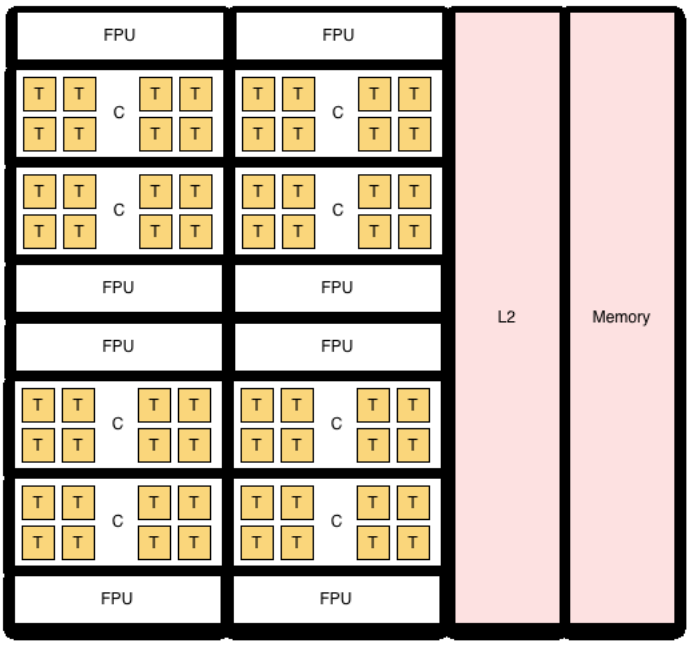


i.e. 16 general-purpose PEs as children of a single SMP grouping node. The linear order of the PEs can be set so that adjacent nodes reflect adjacent cores on chip. The (virtual) attributes of the nodes can be documented as per the hardware

specification.

One could note here that this choice of description does not take into account the memory hierarchy. An additional level in the tree could be introduced e.g. to represent clusters around L3 caches.

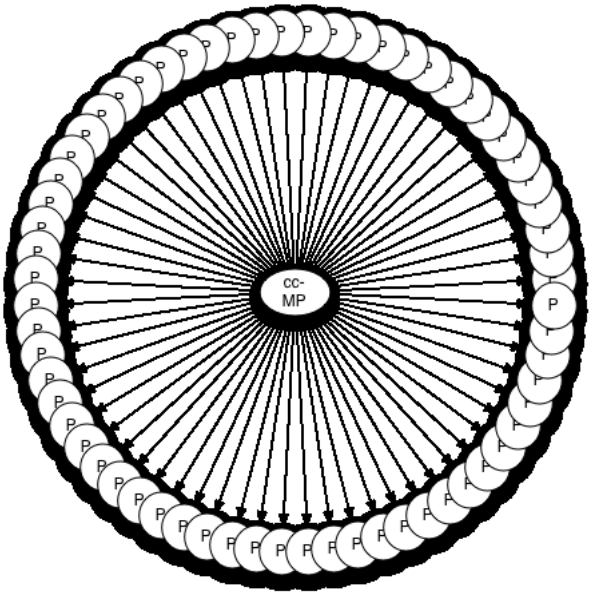
As another example in this category, we take a Niagara T2-based system as per the specifications from Sun:



This hardware consists of a single processor with a single L2 cache and external memory interface. The L1 caches are maintained coherent by an internal, high-speed snoop bus. The processor contains 8 cores, each core with its own FPU. Each core implements 8 hardware threads, scheduled independently from each other.

Each of the 64 hardware threads is visible as an individual, addressable processor by the operating system.

From the point of view of programs, this system has 64 units of effectively programmable concurrency; which are symmetric with regard to access to memory. A straightforward representation in our description system could be as follows:



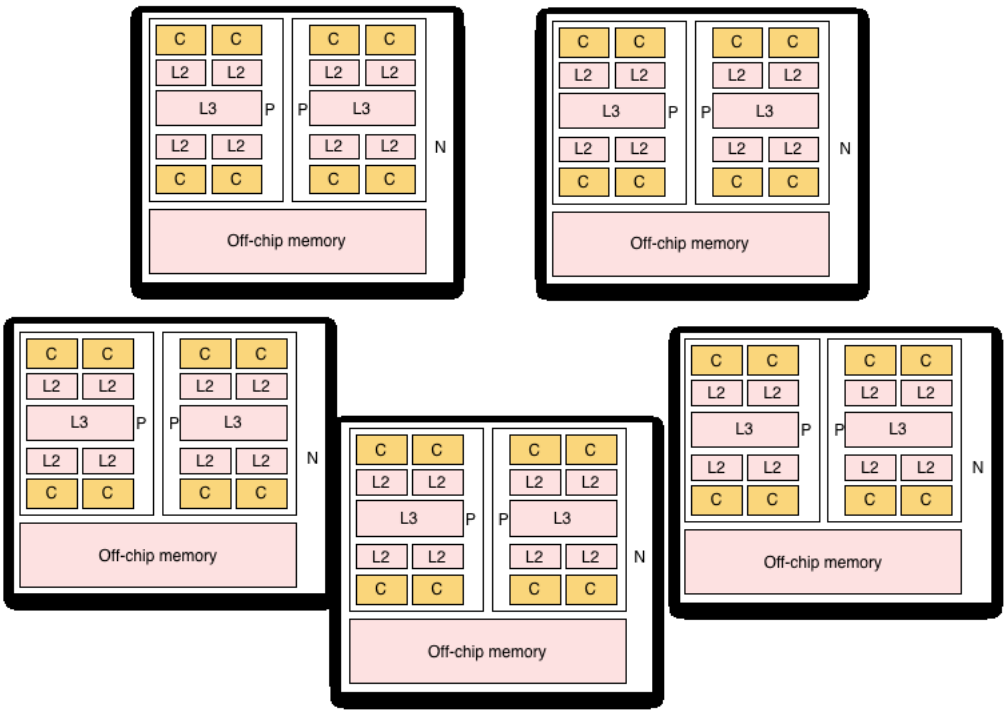
i.e. 64 general-purpose PEs as children of a single SMP grouping node. The linear order of the PEs can be set so that adjacent nodes reflect adjacent threads on chip. Again, the (virtual) attributes of the nodes can be documented as per the hardware specification.

This approach does not consider clustering of cores arounds FPUs. As this could be relevant for performance, it may be desirable to add an additional intermediate level to represent groups of 8 hardware threads (i.e. one group per physical core).

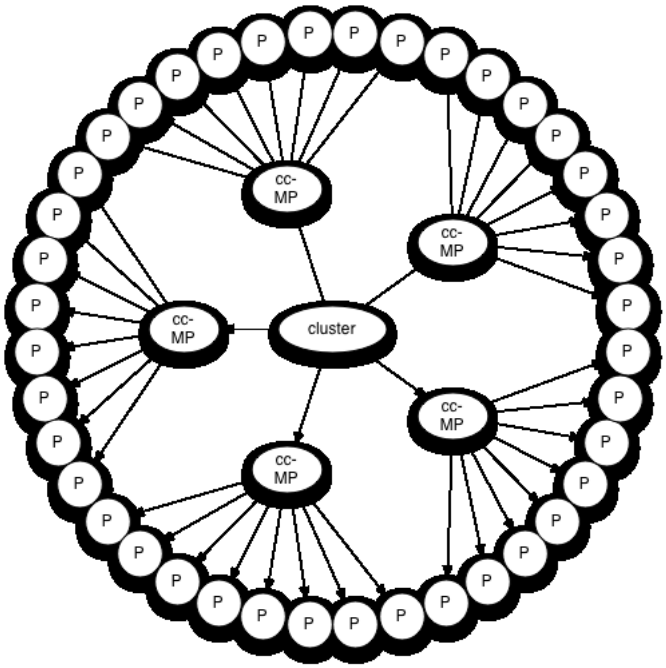
6.3.1.2 Networked multi-cores

The LISA cluster at the SARA computing center in Amsterdam is publicly available for research projects. It consists of homogeneous nodes each containing two Intel Xeon cores.

For this example we consider a selection of 5 nodes in this cluster:



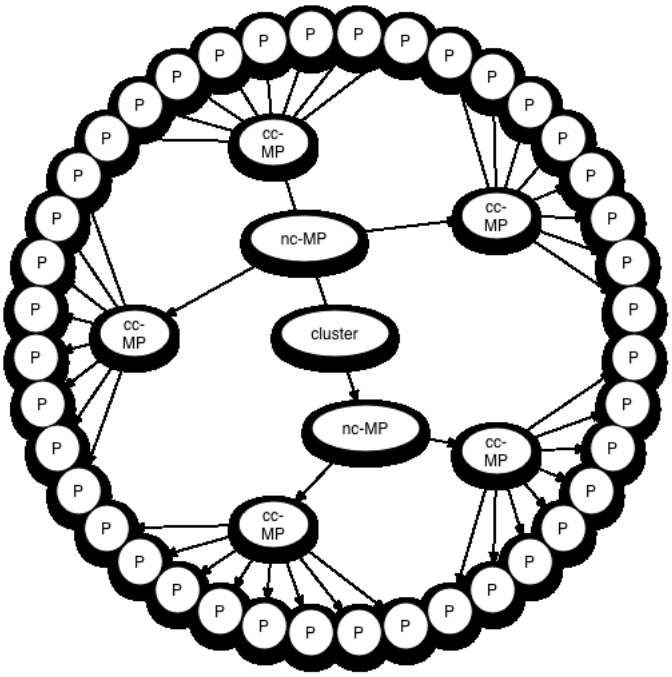
Without further support from software, this (composition of) hardware behaves as a distributed shared memory system; that is, a straightforward representation of this hardware in our system could be:



i.e. a cluster node, with 5 SMP children nodes, with 8 general-purpose PEs as children each.

Now, over the course of the project the work on hardware abstraction at UvA will strive to erase the boundary between distributed memory (disjoint address spaces) and shared memory (shared address space). Assuming some form of run-time software layer is developed that allows to simulate a shared memory over a network of multi-cores, it could then become possible to group parts of a cluster into nc-SMP nodes instead of cluster nodes, to reflect the additional programmatic properties.

For example:

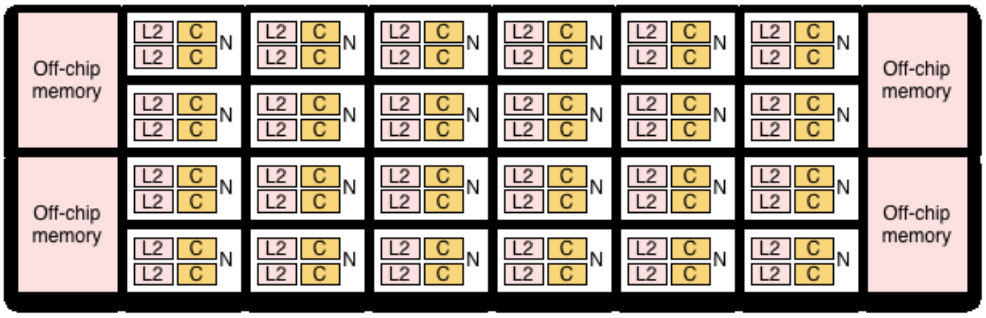


In this example, the virtual hardware description tree reflects that the shared memory virtualization run-time has been configured to provide two shared memory domains, one consisting of 3 nodes, and the other consisting of 2 nodes. The specifics of this configuration is hidden behind the description tree, by describing these domains as “nc-SMP” nodes.

> **Note**
>
> This example does not suggest that it might be *desirable* to implement multiple virtual shared memory domains in a cluster (as opposed to, e.g., only one). It merely shows that our hardware description system covers this possibility.

6.3.1.3 Intel Single-Chip Cloud

The SCC chip from Intel is an experimental on-chip many-core system which is available for use by the project:



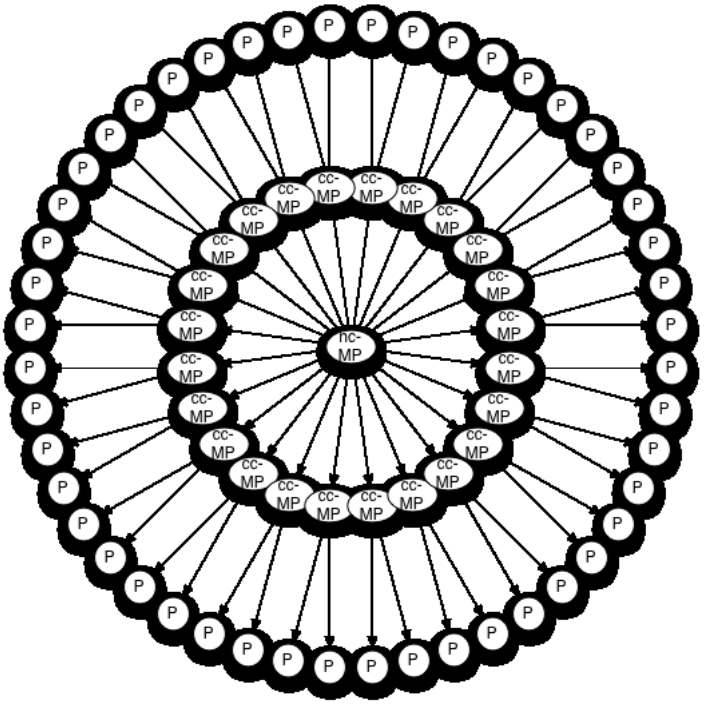
It consists of 24 dual-core nodes with a high-speed interconnect. Each node has two 32-bit x86 cores with one L2 cache per core. There are 4 memory controllers. The mapping of cores to memory can be configured; for example the physical storage can be configured to be visible from all cores, or segmented so that each core sees its "own" memory. There is no on-chip cache coherency protocol implemented in hardware.

Frequency can be configured per node, i.e. per pair of cores. Voltage can be configured per "island" of 4 nodes (8 cores).

The purpose of the research program started by Intel, which makes the SCC chip available to institutions, is to investigate how to best program this chip. It is intended that operating systems and run-time systems explore different ways to expose the hardware to applications. UvA participates in this program with a project separately funded from ADVANCE, but with the intention to overlap research results.

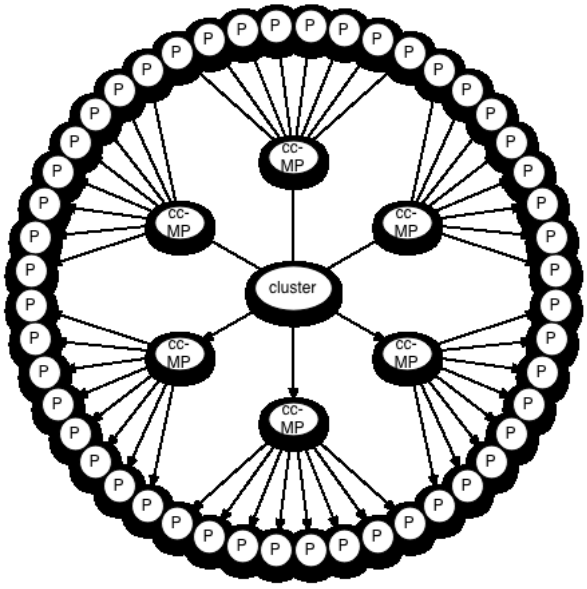
Depending on the approach taken at a system level, this hardware can be described in different ways in our virtual hardware description system. Here are a few examples.

An intuitive approach is to implement (in software) a relaxed coherency protocol between caches, that will provide shared memory to applications with implicit consistency within nodes (pairs of cores) but explicit consistency primitives. This is analogous to virtual shared memory implementations already described in the literature (PGAS). Without considering at first the merits/efficiency of such an approach, the result would be readily describable as follows:



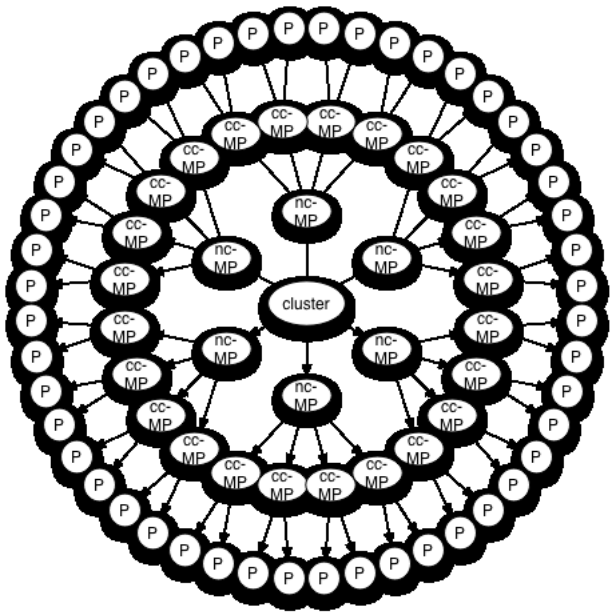
In this description, we reflect that each pair of two cores behaves as a cache-coherent SMP system, whereas the grouping of all nodes is non-coherent and forms a “nc-SMP” node. The system would otherwise be symmetrical with regard to memory access.

Another intuitive approach is to acknowledge that an implicit consistency protocol over the entire chip is expensive; and thus implement a cache-coherency protocol only locally in small groups of cores. The groups would then be mapped to different regions in memory and behave as cluster of distributed memory systems. If such an approach is taken, the resulting system would be readily describable as follows:



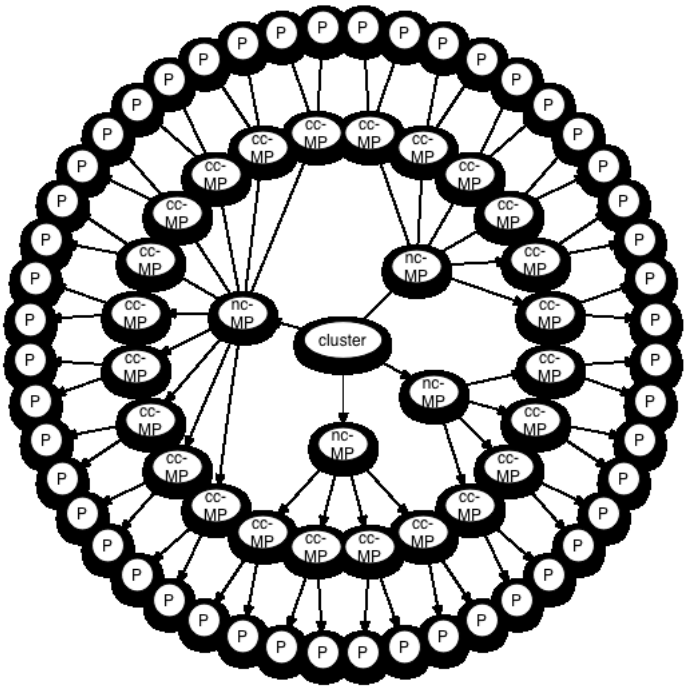
i.e. each group of 8 cores implements a cache coherency protocol for implicit consistency, and is thus described a SMP node. The 6 SMP nodes are then described as part of a cluster node.

One could then combine the two approaches above and implement a relaxed consistency model over the groups of cores. The 6 members of the cluster node would then be "nc-SMP" nodes, each containing the 4 coherent dual-core islands as sub-SMP nodes:



The proposed configurations so far have been relatively homogeneous and symmetric. In a running system however, it is likely that run-time information will allow to tune the description to better support the heterogeneity of usage patterns;

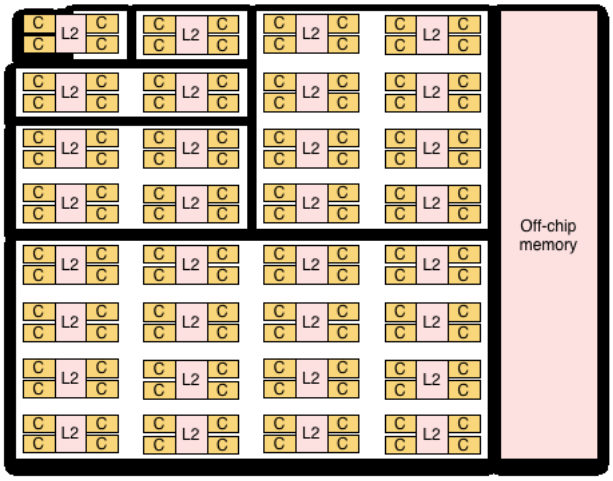
or possibly chip manufacturing defects will need to be represented as well. Regardless, we believe that the description system is general enough that it supports heterogeneous combinations as well, for example:



6.3.1.4 Microgrid

The Microgrid is an experimental architecture researched by the CSA group at UvA. It consists of a composition of many "simple", general-purpose cores with a large amount of internal concurrency (through hardware scheduling of multiple threads).

An example Microgrid configuration from the Apple-CORE project is described thus:

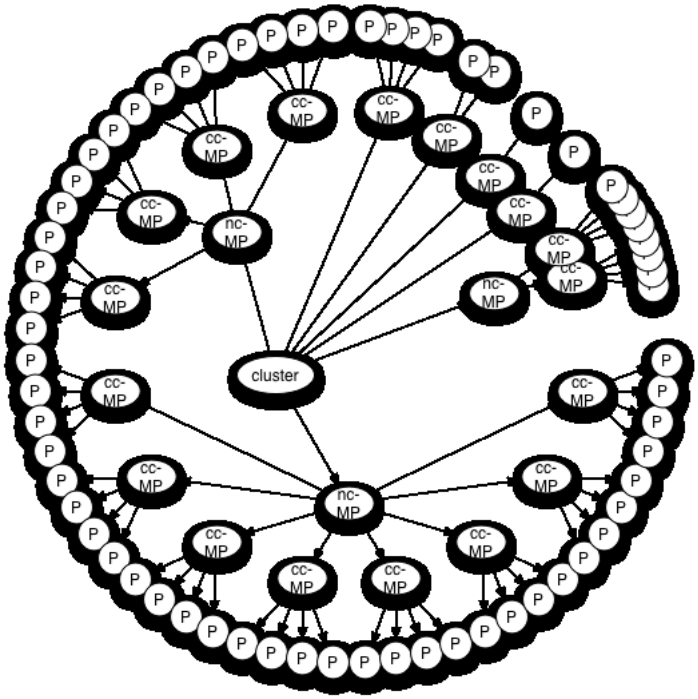


It consists of 128 cores. Each group of 4 cores shares a L2 cache. Memory consistency is ensured below the L2 caches (thus per group of 4 cores), but consistency must be controlled semi-explicitly between L2 caches.

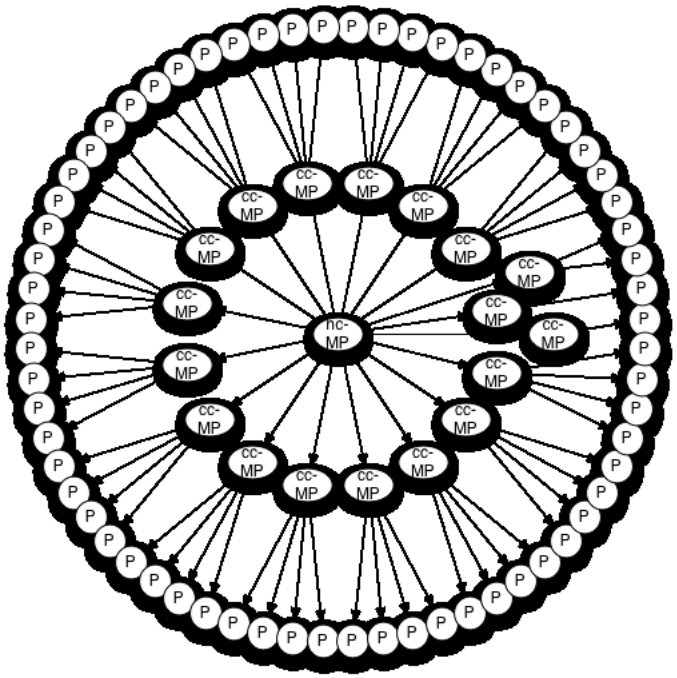
Groups of cores are further grouped in “control rings” over which (parallel) work can be automatically distributed by the hardware. In this configuration, there are 6 such control rings, containing 2, 4, 8, 16, 32 and 64 cores respectively (there are also 2 cores not part of a ring).

Each core can be independently addressed, as well as each ring (through any core in the ring). The hardware threads within each core cannot be independently addressed.

We see two ways to describe this hardware in the system proposed above. One is to consider each control ring as an independent computing resource that can be programmed separately. From a programming perspective the control rings can then behave as nodes in a cluster, corresponding to this description:



One can also disregard control rings from a programming perspective and address each core separately. In which case, what matters most is the boundary between coherent and non-coherent shared memory access. This can then be represented as follows:



6.4 Other properties that can be captured

The motivation for the definition of virtual hardware was driven by the need of the ADVANCE project to isolate statistical behavior of hardware and software components. However, the virtual hardware abstraction can be recycled to capture other aspects of concurrency management:

- *scheduling*: virtual PEs can be configured to provide a specific scheduling policy. The same underlying hardware can be virtualized multiple times when multiple schedule queues are cooperating (e.g. one virtual PE per priority level in a multi-priority preemptive scheduler).
- *process isolation*: if process isolation is provided at a system level (i.e. between applications), the set of virtual PEs available to each application can be a virtualized security domain over the physical hardware resources (with own memory protection, etc).

These topics may or may not be explored in the context of ADVANCE, but the abstraction of virtual hardware / selections give a handle to manage these aspects independently from the expression of program code.

Chapter 7

Integration of SVP with the Kairos on-line resource manager

Key: adv13

Authors: Raphael 'kena' Poss, Timon ter Braak, Robert de Groote

Date: 2010-11-11

Status: Draft

Version: adv13.txt 4135 2010-11-12 01:21:57Z kena

Source: http://notes.svp-home.org/adv13.html

Abstract: This note provides a technical perspective on the integration of SVP with the resource mapping technology provided by TWENTE in ADVANCE. This is the outcome of several discussions between UvA and UTwente over the course of the first reporting period.

7.1 Introduction

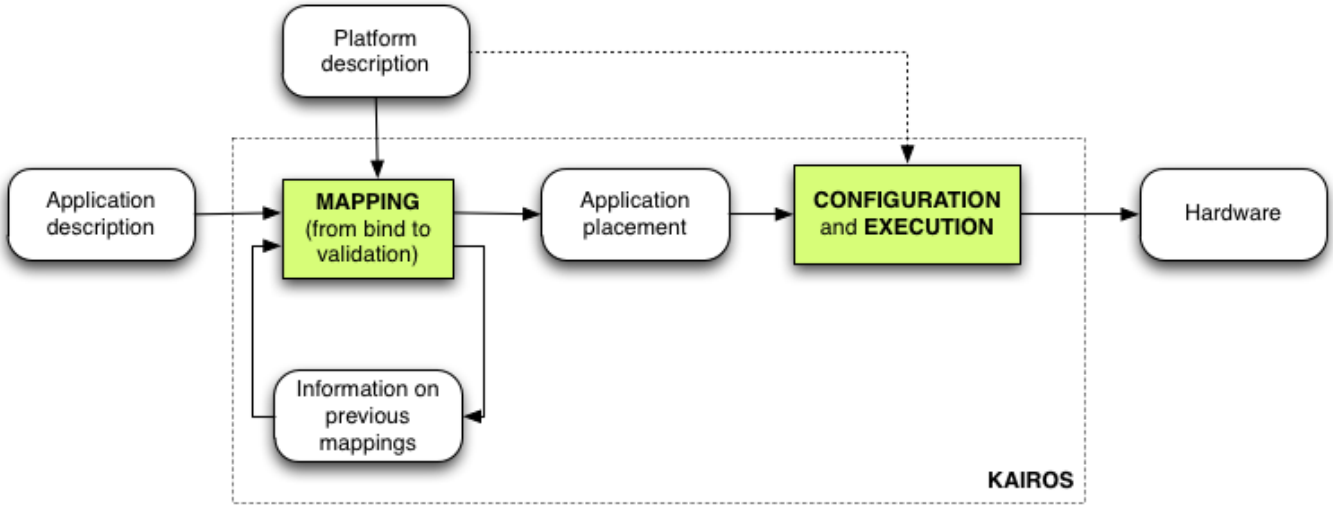
One of the contributions of TWENTE to ADVANCE is an on-line resource mapping engine able to map application task graphs to resources, while satisfying resource and extra-functional requirements. It was documented extensively in [**?**].

This implementation, named Kairos and currently maintained by Timon ter Braak, was originally developped to map synchronous data flow (SDF) graphs to embedded platforms and was extensively tested on the custom CRISP multi-core, multi-DSP architecture also developed by UTwente. While designed for SDFs, it can be extended for use with any task graph where nodes represent persistent activities and edges represent communication requirements.

We give an overview of the function and structure of Kairos in the next section, followed by a strategy to integrate Kairos with SVP in ADVANCE.

7.2 Overview of Kairos from an integrator's perspective

As a functional system, Kairos can be summarized by the following diagram:

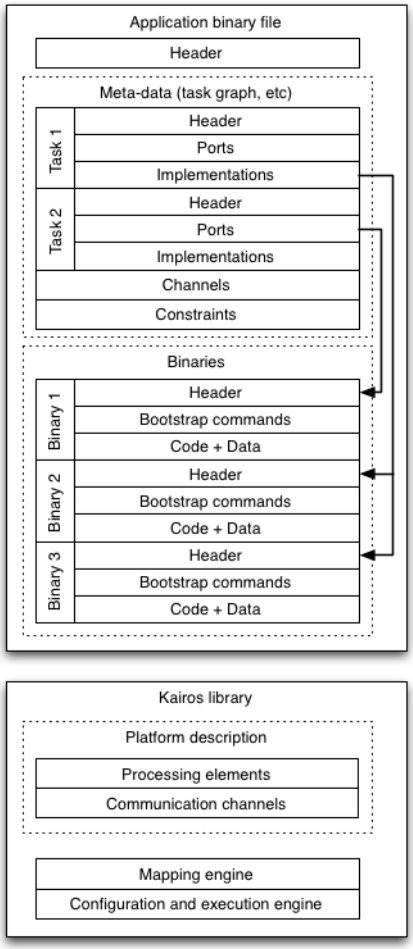


It encompasses two functions:

- a *mapping* function, which takes as input an application description, a platform description, and the state resulting from previous mappings and produces a resolved mapping of application tasks to resource nodes as output.

- a *configuration and execution* function, which takes as input a resolved mapping and performs hardware configuration, loading of programs to the target hardware and execution of the application tasks. The actual hardware access uses hardware-specific handlers outside of Kairos, which can be configured through the platform description.

As a technical framework, Kairos is implemented as a software component that is invoked by applications when they are executed. An example is given by the following figure.



For each application, a single fat executable contains:

- a special task containing the entry point of the application, and which is run on the initial hardware component where the fait binary is loaded and executed. This contains the initial calls to Kairos.
- a binary representation of the task graph, when known at compile time. This is provided by reference to Kairos by the application's entry point. When not known at compile time, the Kairos client must construct a binary representation of the task graph in memory and provide that by reference when requesting a mapping to Kairos.
- the compiled code for all the implementations of all the tasks. Each task can have multiple implementations when different sorts of target hardware require different code.

In the Kairos library, a binary representation of the platform description is maintained. This can be provided either at (Kairos') compile time in the binary

form of the library, or constructed in memory prior to the initialization of Kairos. The platform description references hardware handlers for loading tasks' code+data to specific hardware resources; again these handlers can be either compiled with Kairos or made available dynamically prior to the initialization of Kairos during execution.

7.3 Proposal for an integration with SVP

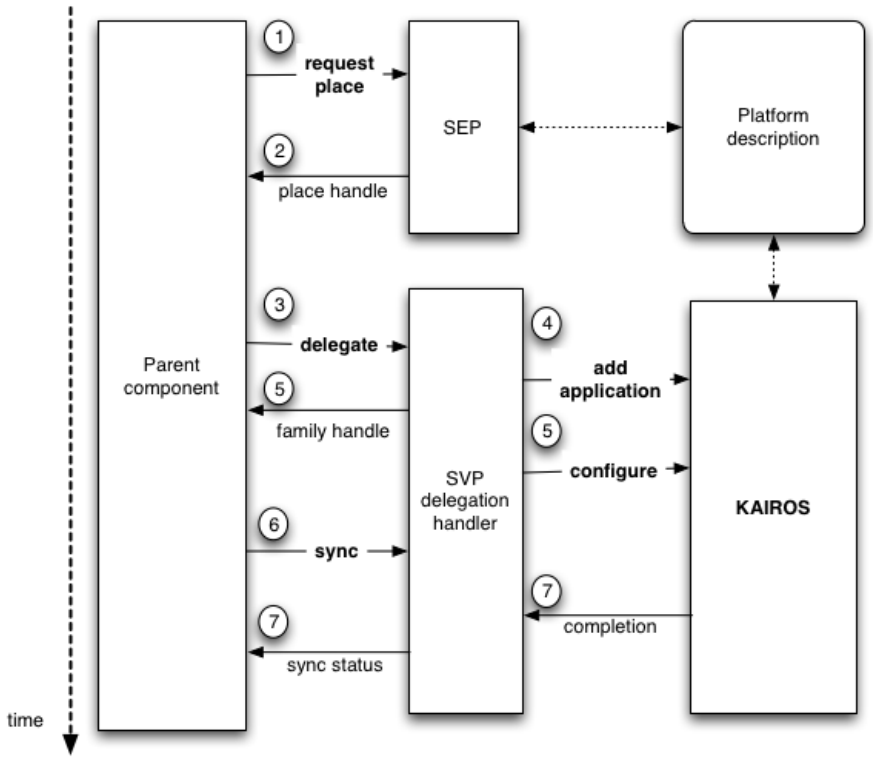
In SVP there are two different ways to approach hardware.

In a first mode described in [**?**], SVP programs request actively during their execution a handle to resources that are not yet visible to them. This is done by sending an *allocation request* to a system service (called "SEP") which computes a suitable allocation based on constraints expressed in the request. The outcome of such an allocation is a handle to the resource and does not entail immediate *use* of the resource. In this mode, hardware properties are used internally in SVP's SEP to match a pool of available hardware to high-level requirements expressed in the program's request for resources, but the structure of the tasks that will be executed on the allocated resource(s) is not visible to the SEP.

The other mode is the *actual use* of (previously allocated) resources. The basic action used by programs is called *delegation* (described in **??**), which creates a family at a named *place* and defines binding between that family and the designated resource. This forces the family to run within the boundaries of that target resource and provides isolation via space partitioning. When a delegation action is issued by a thread, SVP captures this event and translates it to a mapping and execution request for the target place.

Prior to ADVANCE, SVP task graphs expressed during delegation would be mapped across a target resource naively, with a simple round robin or even distribution of threads across the target cores. This was tractable because to this point SVP task graphs were constrained to be linear and homogeneous, and places were also assumed to be composed of homogeneous cores. When either of these assumptions are relaxed (arbitrary task graphs or heterogeneous place structures), family mapping becomes non-trivial. We propose to use the Kairos technology for this purpose.

In the proposed integration, Kairos is used by SVP to support the implementation of delegation. In an example use case, the SVP interactions can be described by the following diagram:



A parent component issues a place allocation request to a SEP with resource constraints. The SEP performs the allocation by partitioning the platform and returns a place handle for a subset of the platform's resources.

At a later stage, the parent component issues a delegation request to SVP. This delegation request carries the place handle allocated earlier, a task description for the family to be created, optional extra-functional requirements, and port information to establish communication between the parent component and the created family.

When the SVP delegation handler receives a request, it propagates the task description and extra-functional requirements to Kairos as a mapping request for a new application. The handle to the resource subset returned by the SEP is also provided, to constrain the resource space where a mapping should be computed. If the mapping fails in Kairos, the SVP delegation request is aborted and an error status is reported to the application. If the mapping succeeds, SVP returns a family handle to the parent component, and asynchronously requests configuration and execution of the application to Kairos.

At a later stage, a parent component may request synchronization on termination of the family. This is resolved by synchronizing with Kairos on termination of the placed application.

7.3.1 Impact on the implementations

The proposed scheme suggests the following adaptations:

- for SVP, a family description provided in a delegation requests must be translated to an explicit task graph data structure, as this is the input required for mapping requests in Kairos.
- for Kairos, the mapping algorithm must be able to constrain the binding process to a subset of the platform, which is partitioned by SVP's SEP.

It is assumed that UVA and TWENTE will coordinate in ADVANCE to achieve this integration.

Chapter 8

Interaction between S-NET and SVP

Key: adv12

Status: Draft

Date: 2010-11-15

Authors: Raphael 'kena' Poss, Clemens Grelck, Chris Jesshope

Source: http://notes.svp-home.org/adv12.html

Version: adv12.txt 4143 2010-11-15 09:14:04Z kena

8.1 Intended cooperation between SVP and S-NET

In its current form advertised by HERTS [**?**], S-NET encompasses both its definition as a *coordination language* and its *implementation* as a run-time system for S-NET programs.

As a coordination language, S-NET introduces a sharp distinction between "coordination", where high-level concurrency is exposed to the S-NET system, and the implementation of algorithm boxes, which is outside of the semantics of S-NET. From a user perspective, S-NET distinguishes between application engineers and concurrency engineers and provides different interfaces to both kinds of users. We highlight here that the *language* S-NET does not mandate a specific execution model for the expressed concurrency, other than "suggesting" distribution in an implementation through the tagged parallel replication operator.

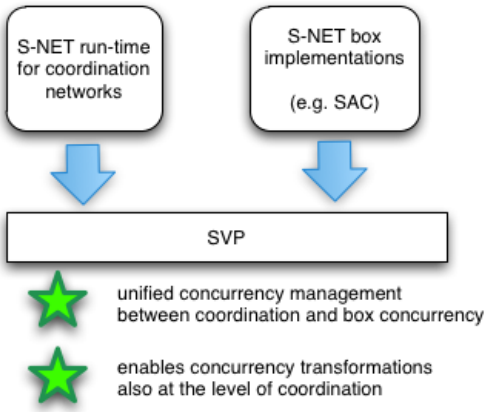
Next to this aspect, one must consider the *implementation* of S-NET. The technology advertised by HERTS and proposed for ADVANCE conceptually maps the S-NET language semantics into an *abstract execution model* where boxes are (sequential) activities communicating through buffered streams. The reification of this model on a shared memory system can, for example, project it onto a CSP network with a single sequential thread per box, and communication through shared

memory buffers and condition variables. This was the reference implementation made available at the start of ADVANCE.

When considered in isolation as implementations of concurrency management systems, a S-NET runtime system and SVP runtime system are largely independent from each other. However, when implemented in isolation, the two systems also exhibit the following shortcomings as the applications grow in size. When S-NET boxes with internal concurrency (e.g. data-parallelism, divide-and-conquer) are added to a network, this is invisible to a CSP-based S-NET run-time system and jitter increases uncontrolled due to oversubscription of resources; this problem is exacerbated as more internal concurrency or more internally concurrent boxes are added. When SVP concurrency trees become deep and unbalanced with work units that operate on large independent data sets, interference can occur in a naive SVP runtime system due to contention on resources used for communication, e.g. memory or network links.

One goal of ADVANCE was thus to use the S-NET *language* and *abstract execution model* and project them onto SVP, to mutually strengthen the two technologies and lessen the specific shortcomings described above. By exposing both high-level concurrency expressed in coordination and internal box concurrency to SVP, it is expected that the S-NET run-time system can achieve better resource usage in presence of internal concurrency of boxes. By introducing component boundaries and streaming semantics into SVP, it is expected that SVP can better map concurrency onto processing resources by using the extra information about communication requirements provided by S-NET.

The envisioned architecture between S-NET and SVP for ADVANCE can be represented as follows:



8.2 Technical strategy for the S-NET integration in SVP

The main execution model for S-NET is to project boxes onto tasks (threads) that communicate through buffered channels. A bounded buffer size allows for back-

propagation of throughput constraints in a network.

In order to express this concurrency in SVP, some mechanism must be designed to implement streams. The existing implementation of S-NET cannot be reused as-is, as it requires first-class condition variables and implicit communication through shared memory, both of which are not directly supported by SVP. We propose a technical solution in this section.

8.2.1 Overview

The existing S-NET implementation assumes the existence of condition variables and implements streams using shared memory as follows:

- each box is implemented by a single thread;
- the thread reads from the buffer, and executes the box's computation for every input record it finds; the computation in turn contains (sequential) calls to the S-NET's `snet_out` primitive which is in charge of queuing produced records to the output stream;
- when the input buffer is empty, the thread suspends on a condition variable that is signalled by the predecessor box/network for the input stream;
- when the output buffer is full, the thread suspends on a condition variable that is signalled by the successor box/network for the output stream.

In SVP, a thread can only suspend on reading its synchronizing data dependencies or waiting for a long latency operation to *terminate*, such as a subordinate family of threads. Moreover, when considering SVP programs that are mapped onto computing resources with multiple (non-shared) memories, the locality of stream buffers must be explicitly controlled.

Our proposed implementation addresses this by requiring a separate thread for each input record of a S-NET box, and placing stream buffers under supervision of an exclusive place. Synchronization between readers and writers is then achieved through synchronization on singleton family termination.

We propose a synthetic implementation of this scheme in the following section.

In addition to enabling S-NET to run using SVP's concurrency model, this approach has the following extra advantages:

- since each input record is processed by a separate thread, it becomes possible to relocate a box to a different resource dynamically by simply changing the box's place identifier used at each family creation;
- since all inter-box communication occurs using named exclusive places, it becomes possible to control explicitly which resources are used for communication;

- in particular, physical concurrency can be exploited without communicating data between places by recognising the independence of S-NET records. Consider the following, three boxes A,B,C in a pipeline. To get physical concurrency from a CSP implementation the output of A must be communicated to the input of B and subsequently its output to C. With a fixed mapping of box to place inherent to CSP (e.g. A->P, B->Q P->R) this requires communication which between P, Q, R, which may be slow and expensive. With our scheme proposed below, A(1), B(1), C(1) can execute on P, A(2), B(2), C(2) can execute on Q and A(3), B(3), C(3) on R, etc. Note that records, A(1), A(2), A(3) are independent of each other and invoke no internal communication. In practice of course the network is more complex but SVP allows for communication to be optimised away more easily through mapping at the record level, while a CSP implementation does not;

- this scheme requires no implicit memory communication between S-NET boxes other than communicating records. Assuming a suitable protocol to communicate records between memory boundaries, the proposed scheme stays valid even when the SVP resources span a distributed memory;

- the scheme is *functional*; that is, one can remove the concurrency constructs from the code and obtain a sequential schedule for the entire S-NET network.

These properties, especially the last one, will be key to enable automatic granularization of S-NET by SVP in later stages of the project.

8.2.2 Proposed implementation

> **Note**
>
> The program code expressed below reuses the syntax and semantics from **??**.

We first define `buffer_t` for stream buffers. This defines a SVP place identifier that must be used for all its accesses.

```
typedef struct record record_t;

typedef struct {
  svp_place_t  xpid;
  size_t       sz;
  size_t       tl, hd;
  record_t*    buf[];
}               buffer_t;
```

Once this type is specified, the following primitives can be implemented to access the buffer:

```
// dequeue() attempts to read a record from the
// buffer, and returns a NULL pointer if the
// buffer is empty.
  record_t* dequeue (buffer_t* b)
 {
   return sync spawn exclusive_at(b->xpid)
          do_dequeue(b);
 }

 record_t* do_dequeue (buffer_t* b)
 {
   /* this runs with exclusive access */
   if (b->tl == b->hd)
      return 0;
   record_t *r = b->buf[b->tl];
   b->tl = (b->tl + 1) % b->sz;
   return r;
 }

 // enqueue() attempts to write a record
// to the buffer, and returns false if
// the buffer is full.
  bool enqueue (record_t* r, buffer_t* b)
 {
   return sync spawn exclusive_at(b->xpid)
          do_enqueue(r, b);
 }

 bool do_enqueue (record_t* r, buffer_t* b)
 {
   size_t nextp = (b->hd + 1) + b->sz;
   if (nextp == b->tl)
      return false;
   b->hd = nextp;
   b->buf[nextp] = r;
   return true;
 }
```

Then we define data types for streams individual S-NET boxes:

```
typedef struct {
  buffer_t*    buffer;
  // ...some definitions omitted...
```

```
 }               stream_t;

typedef struct {
  void          (*box_code)(record_t* input, stream_t* out);
  stream_t      *in;
  stream_t      *out;
  svp_place_t   pid;
}               box_t;
```

We equip the box data structure with a SVP place identifier, with the intent that this is used to instantiate the execution of the box on each input record. We ensure this using the following two coroutines:

```
// "handle_input" represents the instantiation
// of a box for a single input record. This
// should run on the resource identified by b->pid.
 void handle_input (record_t *input, box_t *b)
{
  b->box_code(r, b->out);
  read_next(b);
}

// "read_next" attempts to read another input
// record. If one is available, it tail-recurses
// into handle_input with a new thread creation.
// Otherwise, it simply terminates the thread.
 void read_next (box_t *b)
{
  stream_t *in = b->in;
  record_t *r = dequeue(in->buffer);
  if (r == NULL)
  {
      // the buffer was empty.
       // ... some code omitted here ...
   } else {
      // a record was found, process it.
       // ... some code omitted here ...
       spawn at(b->pid) handle_input(r, b);
  }
  // in all cases, let this thread terminate.
}
```

The two functions `handle_input` and `read_next` mutually recurse using tail recursion as long as there are records to process in the buffer. Tail recursion

guarantees fixed stack usage in the absence of real concurrency during execution. When the buffer becomes empty, the chain simply terminates.

> **Note**
> The implementation of `read_next` above is incomplete. The full version is given below.

From this point, we implement the writer code in S-NET's `snet_out` as follows: if there is no reader yet when `snet_out` is invoked, then the reader thread is created with the provided record. Otherwise if the buffer is non-full, the record is simply queued. If the buffer is full, `snet_out` waits for one reader thread to terminate and tries again.

For this `snet_out` needs to know what box is the designated reader for a stream. We do this by extending `stream_t` as follows:

```
typedef struct {
  buffer_t*    buffer;
  box_t*       reader;
  // ...some definitions omitted...
 }               stream_t;
```

Then we can implement `snet_out` as follows:

```
void snet_out (record_t *r, stream_t *out)
{
  if (!has_active_reader(out))
  {
     // no reader yet, create it.
      make_reader(r, out);
  } else {
     if (!enqueue(r, out->buffer))
     {
        // cannot enqueue the record
         // because the buffer is full.

        // wait for (at least) one
         // reader to terminate...
         wait_on_reader(out);
        // then tail-recurse to self,
         // at this point the buffer
         // should be non-full.
         snet_out(r, out);
     }
  }
```

```
}
```

This implementation requires `snet_out` to be able to introspect whether there is a reader currently running for the stream, and to be able to wait on termination of this reader. For this purpose we introduce the following new data type:

```
typedef struct {
  svp_place_t            xpid;
  reader_t               reader;
  bool                   active;
}                         reader_info_t;
```

This data type, equipped with a SVP exlusive place identifier to serialize its access, provides the required information about reader threads. The field `active` indicates whether a reader is currently running on the stream. The field `reader` stores the synchronization element on which the writer thread can wait. We reveal the exact definition of `reader_t` below.

We use this type to equip the stream data type:

```
typedef struct {
  buffer_t*      buffer;
  box_t*         reader;
  reader_info_t *reader_info;
}                 stream_t;
```

Then we can implement `has_active_reader`, used by `snet_out` trivially as follows:

```
// "has_active_reader" returns a boolean value
// that indicates whether there is a
// reader thread currently computing
// for this stream.
  bool has_active_reader (stream_t *s)
 {
    return sync spawn exclusive_at(s->reader_info->xpid)
           ri_active(s->reader_info);
 }

 bool ri_active (reader_info_t* ri)
 {
    return ri->active;
 }
```

The implementation of `wait_on_reader` is relatively trivial as well:

```
// "wait_on_reader" synchronizes on termination of the currently
// executing reader. This is a no-op if the reader
// has terminated before the synchronization point is reached.
  void wait_on_reader (stream_t *s)
 {
    // retrieve handle to reader thread:
     reader_t f = sync spawn exclusive_at(s->reader_info->xpid)
                        ri_get_reader(s->reader_info);

    // then just wait on the reader to terminate:
     sync f;
 }

 reader_t ri_get_reader (reader_info_t *ri)
 {
    return ri->reader;
 }
```

The implementation of `make_reader` is less trivial. It must create the reader thread and register its handle onto the stream. Intuitively, the two creations could be expressed in this order, as follows:

```
// Create the reader thread
auto f = spawn at(out->reader->pid)
                  handle_input(r, out->reader)

// Then register its handle.
detach spawn exclusive_at(out->reader_info->xpid)
             ri_register(out->reader_info, f);
```

However, we deem desirable to allow `snet_out` to tail recurse into `handle_input` in order to bound stack usage in the absence of real concurrency. This requires the invocation to `handle_input` to occur *after* the registration of the thread handle. This is why we opt to use delayed communication for the reader thread instead, as follows:

```
 void make_reader (record_t *r, stream_t *out)
 {
    void async pending(out reader_t rd) a =
          spawn exclusive_at(out->reader_info->xpid)
                ri_register(out->reader_info)
                  waiting_for(in reader_t);
```

```
    produce to(a) rd = spawn at(out->reader->pid)
                               do_handle_input(r, out->reader)
                                 waiting_for();

    detach a;

}

void do_handle_input (record_t *r, box_t* reader) channels()
{
    // This runs at the designated place
     // for the box.
     handle_input(r, reader);
}

void ri_register (reader_info_t *ri) channels(in reader_t rd)
{
    if (ri->active)
       detach ri->reader;
    ri->reader = consume rd;
    ri->active = true;
}
```

This reveals the exact type for `reader_t`:

```
typedef void async pending() reader_t;
```

With the writer mechanism in place, we need to revisit the reader co-routines. Indeed, the writer side assumes that readers are created in a daisy chain, but may not synchronize on termination of *all* readers; instead it waits on readers only when the buffer is full. However, for the purpose of resource reclamation by SVP, all singleton families that are not synchronized upon must be explicitly detached. This becomes the responsibility of `read_next` as follows:

```
void read_next (box_t *b)
{
  stream_t *in = b->in;
  record_t *r = dequeue(in->buffer);
  if (r == NULL)
  {
      release_reader(in);
  } else {
```

```
      make_reader(r, b);
  }
  // in all cases, let this thread terminate.
}

void release_reader (stream_t *s)
{
  detach spawn exclusive_at(s->reader_info->xpid)
         ri_release(s->reader_info);
}

void ri_release (reader_info_t *ri)
{
  detach ri->reader;
  ri->reader_active = false;
}
```

The collection of all the implementation snippets presented above constitutes our proposed scheme for implementing S-NET with SVP.

Chapter 9

Glossary of SVP for ADVANCE

Key: glo6

Authors: Raphael 'kena' Poss

Status: Draft

Date: 2010-07-06

Version: glo6.txt 4103 2010-11-08 10:32:37Z kena

Source: http://notes.svp-home.org/glo6.html

9.1 Glossary of SVP concepts

9.1.1 Base concurrency concepts

thread A sequential unit of work. All threads are part of a family.

family A group of threads (see thread) related by:

- a single creation point executed by a parent thread;
- an index space;
- a common thread function;
- a pattern of synchronizing data dependencies;
- a designated place, which can be left implicit.

All threads in a family run the same thread function on the same place, but each in a different thread context; in particular their execution can diverge.

singleton family A family of one thread.

creation An operation run by a thread to create a family. Its parameters are:

- the index space for the created family;
- the place where the family execution will occur;

- the thread function to be executed by each thread in the family.

The result of family creation is a family handle that can be used for synchronization on termination on that family.

place Named collection of hardware resources (e.g. cores) where a SVP family can execute.

exclusive place Special type of place where the scheduling guarantees that only one family can run at a given point in time. This implements the "secretary" concept of [**?**] and can be used to implement mutual exclusion and monitors.

delegation The special case where creation specifies the target place explicitly.

synchronizing data dependencies A SVP family and its parent thread are related via a set of synchronizing data dependencies to, from and within the family. These are declared explicitly at the point of creation of a family. Implicit synchronization occurs when they are accessed (they effectively implement dataflow channels or I-variables [**?**]).

synchronization on termination Action that can be performed by a thread to bulk-synchronize on the termination of all threads in a designated family. This operates on a family handle produced during creation of a family.

A given family can be synchronized upon by at most one thread. There is no functional output for this action other than knowledge about the termination of the family.

asynchronous termination An action performed by a thread to terminate a family asynchronously.

This can be either within the family ("break"), which causes the family to terminate without allowing further threads to execute; or by a resource manager ("kill") to re-claim the place where the family is running.

parent The thread that performed the creation of a family.

index A scalar value that identifies a thread within a family. The index is visible to the code run by each thread. The range of index values is defined by the parent thread during creation.

thread function A sequential program to be run by each thread in a family. This produces no return value but may communicate using synchronizing data dependencies.

thread context Entity in an SVP runtime system where thread execution occurs. This captures a thread's machine state including variables, synchronization information and local storage.

family handle Result of the creation of a family. Can be used for synchronization on termination.

9.1.2 Extra terms from the Hydra implementation

consistency domain Collection of hardware resources where threads can communicate using implicit dependencies in shared memory. Such a communication is not allowed by SVP across consistency domains.

memory object A named region in memory that can be made consistent using explicit actions across consistency domains.

For more information see **??**.

9.1.3 Proposed concepts from the SL extension

function A sequential program that optionally returns a value.

spawn The special case of a creation action executed by a thread, when creation defines a singleton family running a thread function that executes a given function and captures its return value.

When a spawn action is used, the corresponding synchronization on termination allows the synchronizing thread to access the return value.

In the revised SL syntax proposed in **??**, spawn is the preferred language construct, and the SL compilers expands this in a variety of creation constructs in the target SVP implementation.

For more information see **??**.

Chapter 10

Implementation work for ADVANCE, Nov 2010

Key: adv16

Status: Draft

Authors: Raphael 'kena' Poss

Date: 2010-11-17

Source: http://notes.svp-home.org/adv16.html

Version: adv16.txt 4158 2010-11-17 23:42:55Z kena

10.1 Summary of efforts required

The following table summarizes the implementation work identified during this reporting period. It also indicates to which task in WP3 they are linked, and gives a rough estimate of the effort required.

Note that this is not exhaustive as the Description of Work implies other efforts in each task. Only the activities identified in this report are listed below.

Description	**Origin**	**Project tasks**	**Effort (est. months)**
Extend the Hydra framework with places: exclusive, named.	**??**	WP3d	2 month
Extend the Hydra framework to support delegation to accelerators.	**??**	WP3b, WP3c	unknown
Implement a new front-end for the SL compiler.	**??**, **??**	WP3c	3 months

... continued on next page

Description	**Origin**	**Project tasks**	**Effort (est. months)**
Implement the annotation system for extra-functional properties and requirements.	**??**	WP2b, WP3a, WP3b, WP3c	3 month (with HERTS)
Implement tools / languages for describing resources	**??**, **??**	WP3a, WP3d	1 month
Extend the SEP protocol / implementation to use the ADVANCE resource system.	**??**, **??**	WP3b, WP3d	3 months
Integrate Twente's Kairos with SVP.	**??**	WP3d, WP6e	5 months (with TWENTE)
Adapt S-NET for use with SVP.	**??**	WP2b, WP2c	3 months (with HERTS)
Implement a process model and monitoring scheme.	XXX	WP3a, WP3b, WP6c, WP6d	unknown (with TWENTE)

FIXME: maybe missing here: describe statistical models for program behavior on SVP resources? Does that even make sense?

10.2 Summary of partners interactions

- With TWENTE: Integration of Kairos for on-line spacial resource mapping. Implement resource monitoring.
- With HERTS: Implement/integrate the annotation system in SAC for extra-functional properties and requirements. Adapt the implementation of S-NET.