Project no. 248828

ADVANCE

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

Resource Management Servers
D25

Due date of deliverable: Sep 30, 2013
Actual submission date: Nov 11, 2013

Start date of project: February 1st, 2010

Type: Deliverable
WP number: WP3
Task number: WP3d

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

Version 1.0 / Last edited by Clemens Grelck / Nov 10, 2013

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

Revision history:

Version	Date	Authors	Institution	Section affected, comments
0.1	25/09/2013	Clemens Grelck	UvA	Deliverable frame
0.2	30/09/2013	Clemens Grelck	UvA	Executive summary
0.3	20/10/2013	Clemens Grelck	UvA	First draft
0.4	01/11/2013	Clemens Grelck	UvA	Various refinements
0.5	06/11/2013	Clemens Grelck	UvA	First complete draft
1.0	10/11/2013	Clemens Grelck	UvA	Final version

Reviewers:
Alex Shafarenko, Clemens Grelck

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
WP3d	Resource Management Servers	UvA*

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

Executive Summary

This document summarises the progress made in Work Package 3 on hardware virtualisation during the 4th and final reporting period of the Advance project.

In the earlier project phases the notion of hardware virtualisation was concretised as a dynamic execution and resource management platform for S-Net streaming networks of asynchronous components and has been implemented and evaluated on a variety of hardware platforms. Progress during the 4th reporting period has primarily been made in three directions:

(a) based on our experience made in and reported after the 3rd reporting period we used period 4 to refine our view on hardware virtualisation and designed, implemented and evaluated the novel FRONT runtime system for S-Net.

(b) we designed and implemented the *resource management server* as demanded by work package 3d for shared memory nodes;

(c) we designed and mostly implemented fully autonomous resource management for (scalable) multi-node systems.

This concludes our work on hardware virtualisation in the context of the ADVANCE project.

Contents

Chapter 1

Introduction

1.1 Overview and Context

Figure 1.1 contextualises Work Package 3 within the ADVANCE project. Core to the work package is the design, implementation and evaluation of the SVP System Virtualisation Platform. During the first two years of the Advance project SVP has been concretised to a dynamic execution and resource management platform for S-Net [8] streaming networks of asynchronous components, where components are implemented either in plain C or (preferably) in the functional array language SAC [9].

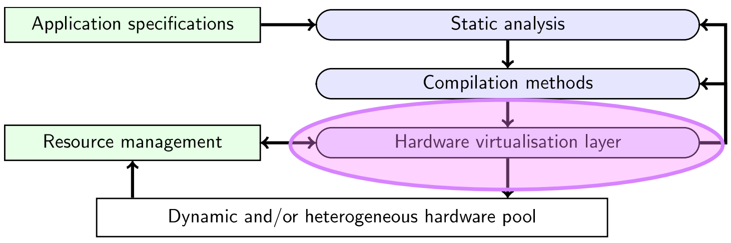
Application specifications
Static analysis
Compilation methods
Resource management
Hardware virtualisation layer
Dynamic and/or heterogeneous hardware pool

Figure 1.1: Positioning of hardware virtualisation in the context of ADVANCE

SVP is the mediator between S-Net streaming networks, box implementations (in C or SAC [9] and the concrete hardware they are supposed to run on. S-Net exposes concurrency and dependencies of computational tasks, but is inherently resource-agnostic and resource-unaware. The system virtualisation platform SVP, in contrast, is aware of the computational resources at hand (compute nodes, processors, cores, hardware threads, memories) and maps S-Net tasks to concrete execution units in an efficient way. Furthermore, it continuously monitors the dynamic behaviour of the streaming network and reports the corresponding information to the upper layers of the Advance technology stack, namely static analysis (WP 4) and compilation methods (WP 5) for refinement of an application's implementation. SVP likewise interfaces with external runtime resource management (WP 6), which makes advanced mapping decisions based on the monitoring data. These decisions are communicated back to SVP, which dynamically implements them by adjusting the task-to-resource mapping accordingly.

1.2 Advance Technology Stack

Fig. 1.2 illustrates the Advance technology stack from a more technical perspective. Going from top to bottom, the S-Net compiler takes an S-Net coordination program and compiles it to the S-Net *Common Runtime Interface (CRI)*. This is a well-defined interface that exposes the structure of an S-Net streaming network as an application-specific call tree of application-agnostic library functions instantiated with application-specific data structures. The library functions of the common runtime interface can be (and have been) instantiated with alternative implementations and thus allow for entirely different technical realisations of S-Net streaming networks.

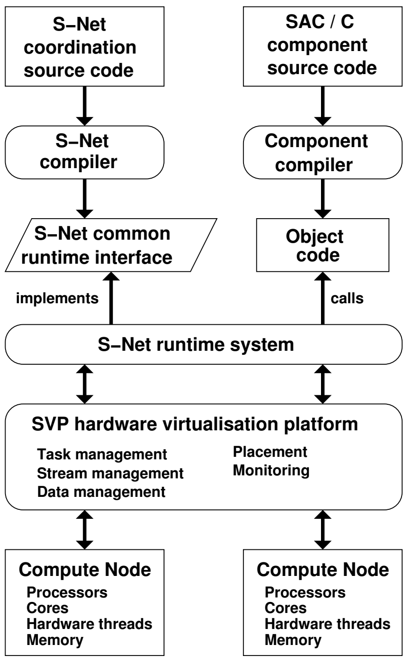

Figure 1.2: Advance technology stack

For the Advance project only one implementation is relevant, which we refer to as *the* S-Net runtime system for simplicity. This runtime system [7] follows an approach similar to communicating sequential processes (CSP). Each S-Net component, including a number of internal components for splitting and merging streams, is instantiated as such a sequential process. Internally, an S-Net component executes an event loop that reads a record from the input stream (potentially blocking on an empty stream) and processes that record, depending on the kind of component and the record data. In the case of S-Net boxes this usually involves calling an external function implemented in a component language and compiled to binary code by the corresponding component compiler.

This may trigger the dynamic instantiation of further streaming network parts (due to dynamic serial and parallel replication) and usually results in one or more records to be sent to the output stream(s). To this end, component execution may block on a full output stream[1]. The process continues until a special input record signals the component to terminate. This can be due to global network shutdown or due to partial network garbage collection [4].

The S-Net runtime system, as described above, is resource-agnostic. The implementation of S-Net tasks and streams and their mapping to a constraint set of resources is the function of the system virtualisation layer. As a part of SVP we use the *Light-weight Parallel Execution Layer (LPEL)* [17] to implement the above S-Net components and the streams via which they communicate. LPEL maps the S-Net components to a given fixed number of kernel worker threads for shared memory execution platforms and organises their orderly interaction.

1.3 Advances in System Virtualisation

During the 3rd reporting period we encountered unexpected anomalies in the execution of S-Net programs on shared memory hardware using our hardware virtualisation approach. These anomalies were reported and intensively discussed in deliverable D24 as well as in [13]. In essence these anomalies can be characterised as states of program execution where not all available hardware resources are effectively utilised by S-Net, despite the availability of work in the system in principle, because this work is blocked in front of busy component executing a long-running box instantiation. Our work on active process migration, also reported in deliverable D24 as well as in [18], can be seem as an attempt to rectify these shortcomings, but the results were not convincing.

We further analysed the situation by making intensive use of the LPEL monitoring facilities [15]. The outcome of these investigations were the insight that the thorough separation between the S-Net runtime system on the one hand and the LPEL system virtualisation layer on the other hand may be useful from a software engineering perspective but proves counterproductive from the perspective of order system behaviour and effective resource utilisation. A this point we made the strategic decision to not cover this observation up and to further attempt to work on symptoms or tweak the existing implementations Instead we gave it a try to design, implement and evaluate a novel runtime system for S-Net with integrated hardware virtualisation.

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

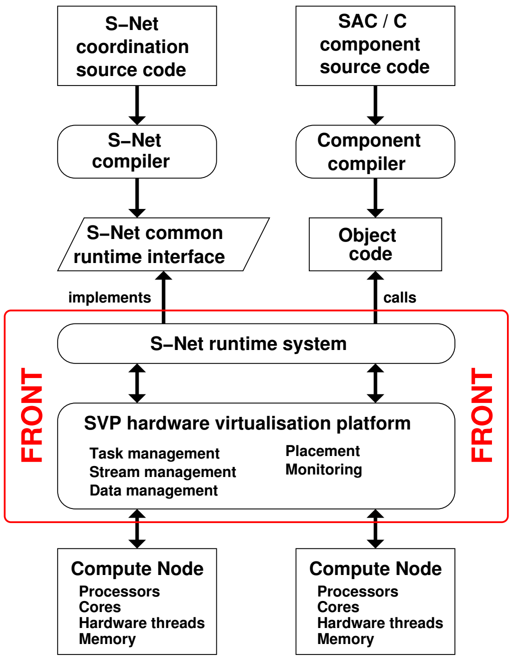

Figure 1.3: Refined ADVANCE technology stack with FRONT runtime system

We coined this new runtime system FRONT; Fig. 1.3 illustrates the refined ADVANCE technology stack. In Chapter 2 we provide a detailed description and some evaluation of FRONT.

1.4 Resource Management Servers

The main theme of task WP3d, our original goal for the 4th reporting period are *resource management servers*. These are characterised as system services that dynamically allocate execution resources to executing programs, more precisely to S-Net streaming networks with sequential or data-parallel components. A particular motivation for this active form of resource management is to control the energy consumption of a running application. In a more general context we aim for adapting the amount of used resources to the actual dynamic needs of the application in order to optimise resource utilisation in multi-user and/or multi-application scenarios.

Following the positive evaluation of the FRONT runtime and system virtualisation system we decided to implement resource management servers in the context of FRONT, rather than the original S-Net runtime system and the separate LPEL system virtualisation layer. As a consequence of this work, FRONT can either be used with a fixed set of resources (processors, cores, hyper-threads) and then makes the best possible use of these resources, or resources can actively be managed by a FRONT resource management server. In the latter case resources are not used in a greedy fashion, but the amount of active computing resources is continuously adapted to the effective level of concurrency exposed by the running S-Net streaming network. In this way, we expect to indirectly control (and reduce) the energy consumption of a system, assuming that the underlying operating system automatically reduces the clock frequency and potentially the voltage of underutilised processors and cores.

Chapter 3 provides more on our notion of resource management servers.

1.5 Virtualisation of Scalable Systems

An issue carried over from the third reporting period was the completion of hardware virtualisation for multi-node system architectures with distributed memory, so-called scalable systems. Given the observed issues with hardware virtualisation on shared memory systems, we temporarily reduced our efforts towards more challenging system architectures. With those issues being resolved we again turned out efforts into this direction, but with very limited resources of time and engineering.

While we still, as originally planned, base our efforts on our experience with Distributed S-Net [5], it is clear from the beginning that a system virtualisation approach requires design decisions to be taken differently. In Distributed S-Net the programmer explicitly specifies which subnetworks are instantiated on which node. The programmer is, thus, fully responsible for effective workload distribution and resource utilisation and Distributed S-Net merely provides the (high-level) tools to do so.

For the full virtualisation of a scalable system we need a different approach. While at least in principle it may be possible to retrieve suitable streaming network annotations in the style of Distributed S-Net by means of static analysis. However, in the presence of very limited resources being left for pursuing any approach in this direction we decided for a fully dynamic computation offloading model that works on the level of individual box instantiations and thus widely ignores the potentially complex network structure around the boxes. In this model the primary compute node executes the S-Net streaming network while all other compute nodes act as remote compute servers that take responsibility for individual computations but not for running the streaming network itself. A cache-only memory architecture that aims at reducing communication requirements complements the picture.

Our approach to extend hardware virtualisation to scalable systems with distributed memory architectures in general and resource management servers in particular is detailed in Chapter 4.

Chapter 2

Advances in System Virtualisation

The design of the novel FRONT runtime system is based on an extensive body of experience with the original S-Net runtime system and the LPEL threading layer. S-RTS was mainly intended as a proof-of-concept for macro dataflow computing in the style of S-Net. The design of LPEL as an underlying system virtualisation layer at the beginning of the ADVANCE project was influenced by the immediate need to provide meaningful profiling data of S-Net streaming networks to our partners for statistical analysis.

The development of the combined S-Net runtime system and system virtualisation layer FRONT is motivated by our wish to achieve high system utilisation and generally good performance in comparison with alternative parallel programming approaches, for now on (large-scale) closely-coupled servers with hardware shared memory. In the sequel we discuss the major design choices in the FRONT system.

2.1 Entity Graph vs Property Graph

Due to the presence of serial and parallel replication combinators in S-Net, the graph of communicating sequential processes is continuously evolving. It grows due to the demand-driven re-instantiation of argument networks, and it shrinks due to network garbage collection under certain circumstances [4]. Any change in the graph must be attributed to overhead that competes for resources with productive box computations. In many situations, evolving the network graph additionally reduces the effectively exploitable concurrency. One of the key design ideas behind the FRONT runtime system is, thus, to replace the dynamically evolving graph of communicating sequential processes by two complementary graphs: the static *property graph* and the dynamically evolving *entity graph*.

We illustrate our first design choice by means of the simple, yet non-trivial example S-Net coordination program shown in Fig. 2.1. The example application implements a dynamic graphics filter pipeline; a graphical illustration of the same program is sketched out in Fig. 2.2.

```
net Example ({Img} -> {Img})
{
  box Pre ( (Img) -> (R,G,B) );
  box fR ( (R) -> (R) );
  box fG ( (G) -> (G) );
  box fB ( (B) -> (B) );
  box Test ( (R,G,B) -> (R,G,B) | (R,G,B,<done>) );
  box Post ( (R,G,B,<done>) -> (Img) );

  net Split
  connect [{R,G,B} -> {R} ; {G} ; {B}];

  net Sync
  connect [|{R},{G},{B}|] * {R,G,B};

  net Pipe
  connect (Split .. (fR | fG | fB) .. Sync .. Test) *
      (cont.){<done>};

} connect Pre .. Pipe .. Post;
```

Figure 2.1: Example S-Net implementing a dynamic graphics filter pipeline

The top-level pipeline consists of a preprocessing step (`Pre`) transforming an abstract image into its red, green and blue colour components, a dynamic filter pipeline (`Pipe`) and a postprocessing step (`Post`) that turns processed RGB image components back into the original image representation.

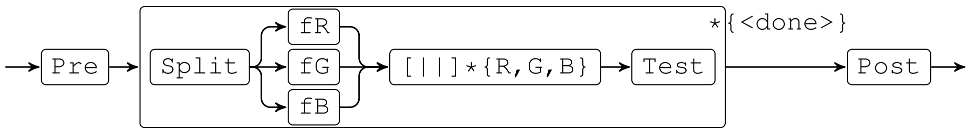

Figure 2.2: Illustration of the S-Net streaming network defined in Fig. 2.1

The dynamically replicated filter pipeline, implemented with a star-combinator in Fig. 2.1 and shown in the central compound box in Fig. 2.2, consists of a splitter that divides an RGB-image (record) into three independent records carrying on the red, green and blue colour information, respectively. These records are routed to three custom filters by means of parallel composition. After the individual processing of colour components, separate red, green and blue records are captured and combined into a single record in the subsequent synchro-cell.

Since we assume a stream of images to be processed by our filter (pipeline) and a synchro-cell only synchronizes a single set of incoming records (see above), we embed the synchro-cell in another serial replication combinator. Consequently, the `Sync` network synchronizes and combines the first red value with the first green and the first blue value on the inbound stream, the second red with the second green and blue value, and so on. On the combined RGB-image we run a simple test whether to continue filtering or to output the image. The decision is signalled by the presence or absence of the tag `<done>`, which is inspected by the star combinator and later removed in the postprocessing step.

At program startup execution of the compiler-generated function call graph (the common runtime interface, see Section 1.2) creates the static *property graph*. In Fig. 2.3 we show the property graph for our running example. It merely contains placeholders for the replication combinators. The property graph contains all information required for running the network, e.g. types for routing decisions, and serves as a template for evolving the entity graph.

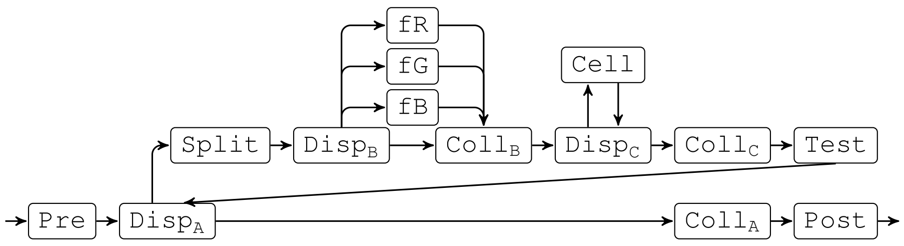

Figure 2.3: The static entity graph of the running example introduced in Fig. 2.1

The entity graph is created strictly demand-driven. Initially, FRONT creates just the first entity, in our example this is `Pre`. Even the creation of the outgoing stream is postponed. The first record to leave `Pre` will detect the absence of an outgoing stream. The `Pre` entity structure contains a pointer to the `Pre` node in the static node graph of Fig. 2.3. This suffices to give the outgoing edge, which in turn gives the destination node. From the destination node the entity type is available (in this case a *star dispatcher*) together with application-specific type parameters (e.g. the termination pattern is `<done>`).

In this case the destination is a dispatcher for a star combinator: $\texttt{Disp}_{A_1}$. This requires special handling, because it has two outgoing connections which ultimately both need to end up at the same collector $\texttt{Coll}_A$. In order to be able to only allocate a stream when it is first needed to output a record, but still guarantee that all incoming streams to a collector use the same destination entity, each entity carries with it a stack of pointers to future collectors. When $\texttt{Disp}_{A_1}$ is created, the collector $\texttt{Coll}_A$ is created as well and pushed onto the stack. Each new entity receives a copy of the stack from its predecessor. When the streams layer detects (according to the static property graph) that it is opening a stream to a collector, then it will take the collector from the top of the stack.

Why is it not possible to do with only a static component graph at runtime? The answer is simple: while boxes are stateless by design, whole networks may indeed be stateful as they may contain synchro-cells. At a synchro-cell it is important to match the right records according to the *semantics* of S-Net, which is based on actual replication.

2.2 Process-centric vs Data-centric View

The process-centric view characteristic for S-RTS implementing each entity by a dedicated thread of control, even if implemented as a logical, cooperative thread in LPEL, turned out to be expensive. Thus we aim at a radical change in the interpretation of streaming networks and abandon the process-oriented design. Instead, we create a fixed set of *worker threads* to be run on different cores at application startup. These workers roam the entity graph in search for work. When a worker finds an entity with a non-empty input stream, it locks the entity using a compare-and-swap (CAS) instruction while it processes that entity. One serious consequence of this design is that a thread can no longer block when the entity writes to an already full stream buffer. As the semantics of S-Net does not bound the number of records a box may emit, streams connecting entities must be unbounded, as well.

The question remains how workers find entities with non-empty input streams. Effectively roaming the entity graph would clearly be inefficient. In particular, collector entities would have to keep track of incoming streams and those have to be examined for non-emptiness. For several reasons, workers cannot just examine the network of entities and test for non-empty incoming streams. Keeping track of entities with work is also inefficient, because then collector entities have to keep track of incoming streams and those have to be examined for non-emptiness. Therefore, the unit of work scheduling in FRONT is the non-empty stream itself. Whenever a worker writes to a stream, it remembers this as a *license to read one record* from that stream for a future invocation at the destination entity. It stores this knowledge in a *stream reference structure*, which contains a pointer to the corresponding stream and a counter for the number of read-licenses it has for that stream. When new read-licenses arrive, the worker looks up the stream reference structure in a hash table which is indexed by a pointer to the stream. To exercise a read-license, the worker first attempts to lock the destination entity. If this succeeds, it decrements the number of read-licenses by one, reads one record from the stream and then invokes the entity. If this was the last read-license, it destroys the stream reference structure and its hash table entry.

Fig. 2.4 illustrates the most relevant aspects of the FRONT runtime system. On the top part we see an entity graph with boxes E, F and G interconnected by streams S holding four records, T holding three records and the currently empty stream U. On the bottom part we see four worker threads that, for example, virtualise the four cores of a quad-core processor. The number of records in the various streams

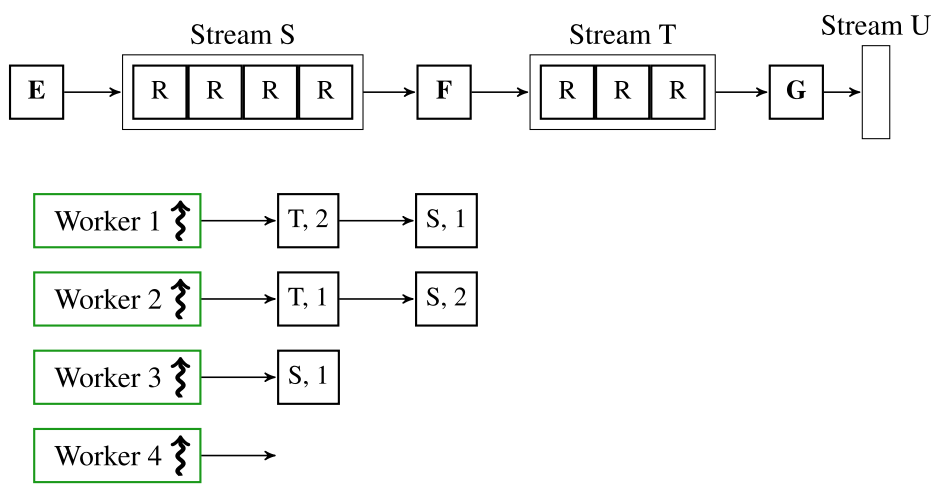

Figure 2.4: Illustration of workers, streams and read licenses

equals the number of read licenses to these streams spread over the four workers' task queues. It is important to note that beyond this numerical equivalence there is no concrete a-priori association between workers and concrete records.

In a potential scenario Worker 1 exercises one of its read licenses on stream T after successfully locking that stream, i.e. it retrieves the first record from stream T and runs box G on it. Worker 2 also has a read license on stream T, but stream T is already locked by worker 1. Therefore, worker 2 traverses its work queue, finds a read license to stream S, which it successfully locks, and runs box F on the first record from stream S. Worker 3 has a read license for stream S, but stream S is already locked by worker 2. Hence, there is currently no work to do for worker 3, but as soon as worker 2 releases the lock on stream S, worker 3 could become active. Worker 4 has no read licenses whatsoever and thus no work to do in the scenario of Fig. 2.4.

Two aspects are remarkable in our example and require our attention. With every record produced by running box G and box F worker 1 and worker 2, respectively, add new read licenses to streams S and T to their local work queues. Obviously, a worker cannot produce read licenses out of nothing, e.g. worker 4 in Fig. 2.4 would always stick to an empty work queue. To properly distribute work among a number of workers we need the additional concept of work stealing. We elaborate on work stealing as well as on how to receive records into the running stream processing system in the first place in Section 2.5.

The other interesting aspect is that in the given example we can only keep two out of the four workers busy at any time. In fact, the level of concurrency in the given example is only 2 as long as we assume standard stream processing. However, given the many more records in the example scenario, it seems we do not fully exploit the concurrency potential. To overcome such limitations we introduce concurrent box invocations in Section 2.6.

2.3 Execution Order

The way workers organize their collection of stream references determines the order in which they are processed. An entity graph can be regarded as a dynamic pipeline with parallel branches, which evolves from an input entity to an output entity. The edges in this graph are the streams which transport the records. In this picture we wish to preferably schedule those non-empty streams, which have the highest probability of quickly producing output. Each output record reduces the memory footprint and also provides the user with results. Another important motivation is to keep the number of non-empty streams as high as possible in order to increase the concurrency which is exposed to all workers. This is best achieved by elongating and widening the entity graph as much as possible.

FRONT stores stream references in a singly linked list per worker; the tail of the list is closer to the input entity and the head closer to the output entity. When a worker searches for a schedulable stream, it traverses this list from the head looking for a stream with a currently unused destination entity. When found, the worker locks the destination entity of the stream, reads one record from the stream and invokes the entity. If the invocation generates new output records which result in a new stream reference structure then these are inserted before the current position in the list. As a consequence, streams closer to the output entity are closer to the head of the list and, therefore, are prioritised.

2.4 Improved Data Locality

We further aim at avoiding the migration of records from core to core to improve data locality in the ubiquitous presence of hierarchical caches. We extend write operations to streams with a flag that identifies the last output record of some entity invocation. In this case the worker thread immediately continues with the follow-up entity and the current record, i.e. it follows the record through the entity graph. This data-centric (instead of entity-centric) solution has the added benefit that we save storing and retrieving read-licenses.

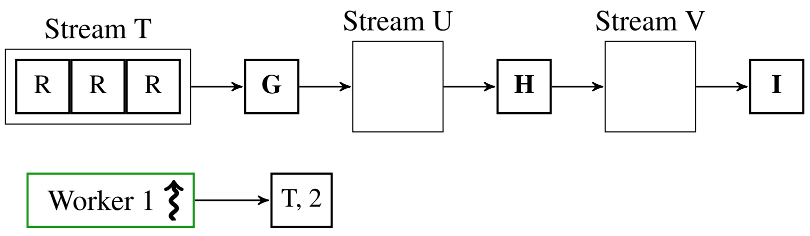

Figure 2.5: Illustration of continued record processing

We illustrate our approach with a simple example shown in Fig. 2.5. Here, we see boxes G, H and I, connected by streams T, U and V. A worker has two read licenses for stream T, which contains a total of 3 records. The worker exercises its first read license on stream T, retrieves the first record in the queue and executes box G on that record. Let us assume that all our boxes in the example are stream transformers that map one input record to one output record, a simple but very common case in stream programming. When box G completes and issues a record marked as described above, the worker does not put the record into stream U, but provided that stream U is empty, immediately continues processing box H with the given record. To do so the worker locks the entity H and at the same time releases the lock on entity G. This allows other workers (omitted in Fig. 2.5) to steal read licenses on stream T from worker 1, but read more on work stealing in the following section.

2.5 Input Control and Work Stealing

When the list of stream reference structures is empty a worker tries to obtain exclusive access to the input entity in order to retrieve records from the input parser. This strategy replaces the concept of back pressure through bounded streams in S-RTS to avoid overloading a streaming network with too many incoming records. Instead, new work is only admitted to the streaming network if workers are still idle.

If there is neither input on the global input stream or another worker has already locked the input entity, idle workers turn into thieve mode and examine the list of stream references of other workers. We store pointers to workers in a global array. Thieves iterate over this array when searching for work. They remember the previously visited victim and continue with the next worker (round-robin). To reduce contention with victims over their stream reference lists at most one thief at a time may visit a victim.

For the same three motivations given previously thieves preferable *steal* read-licenses for schedulable streams which are likely closest to the output. When they find a stream with a lockable entity, they steal half of the number of read-licenses from the victim's stream reference structure. Then they retract from the victim's list and continue with invoking the destination entity for the stolen stream read-licenses. Victims and thieves must exclusively lock a stream reference structure with a CAS (compare-and-swap) instruction before dereferencing a stream pointer or modifying its contents.

It is noteworthy that our variant of work stealing deviates from the standard approach found in most implementations elsewhere. Normally, work lists are doubly linked, and the owner reads from one end while the thieves read from the other end. This model avoids access conflicts between owners (or victims) and thieves as long as there is more than one work item in the list. In the FRONT runtime system we make use of a single-linked list and both victims and thieves read from the head of the work list. This is necessary to ensure that the S-Net network as a whole makes progress towards producing completed records at the global output. In other words, even in the presence of work stealing the FRONT runtime systems aims at computing tasks at the “front” of the streaming network.

Where a worker looks first for more work when its own work list becomes empty is an important design decision. We choose workers to first check global input for more work before trying to steal work from other workers. This choice reduces the overhead created by many workers simultaneously aiming at stealing work that simply does not exist. Moreover, it helps to accelerate the initial ramp up phase of any S-Net network when the number of records in the system is still small and effectively no work exists that could be stolen. As only one worker at a time can lock and thus operate the input entity, new records may enter the streaming network while at the same time other workers aim at stealing work from their peers.

2.6 Concurrent Box Invocation

The S-Net language specifies box functions to be stateless. We can exploit this property to significantly increase concurrency by allowing multiple workers to invoke a box entity concurrently as soon as multiple input records are waiting in the input stream. For the purpose of experimentation, a per-box concurrency limit can be specified for now. As illustrated in Fig. 2.6, we allocate for a box entity an equivalent number of box contexts. Each box context has its dedicated outgoing stream, which ends at a shared per-box collector. The collector merges the incoming streams into one outgoing stream. When a worker invokes a box, it finds an unused box context, locks it and if the number of concurrent invocations is below the limit, it immediately unlocks the box entity, to allow for more concurrent invocations by other workers. The collector entity ensures that despite concurrent box invocations the stream order semantics of S-Net are preserved, i.e. records cannot coincidentally overtake other records.

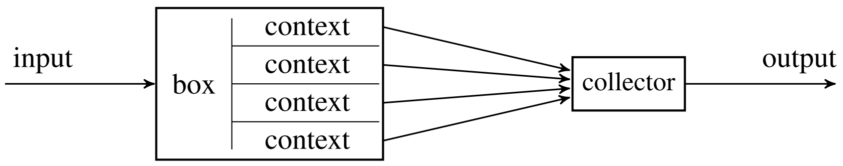

Figure 2.6: Illustration of concurrent box invocations

The ability of the FRONT runtime system to invoke the same box instance multiple times concurrently if multiple input records are waiting to be processed is an important step to fully exploit the concurrency contained in an S-Net specification. Following the macro data flow approach the unit of computation in S-Net is the record, not the box component. Conversely, in a *communicating sequential process* implementation model, as the original S-Net runtime system does, opportunities for concurrent computations are regularly left out whenever multiple records start queuing in the input stream of a busy box component.

We demonstrate the effectiveness of concurrent box invocations for performance and scalability by means of the S-Net MTI-STAP application. MTI-STAP is a signal processing application: Moving Target Indication using Space Time Adaptive Processing [12, 16]; it detects slow moving objects on the ground using an airborne radar antenna. We evaluate the performance of this application to see if concurrent box invocations in the runtime system can improve performance for existing S-Net applications.

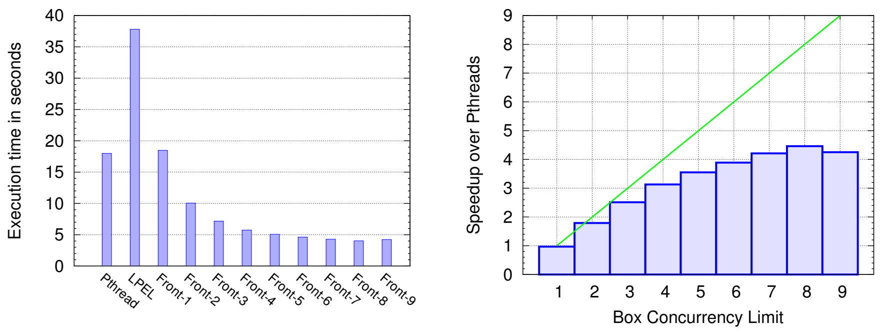

Figure 2.7: MTI-STAP: **(a)** Execution time, **(b)** Speedup for Concurrent Box Invocations.

Fig. 2.7a shows execution times for S-RTS/PTH, S-RTS/LPEL and FRONT. Here FRONT runs with the box concurrency limit set to numbers between 1 and 9 as indicated by the suffix: The label FRONT-1 denotes the default configuration, i.e. no concurrent box invocations, while FRONT-9 denotes the configuration when up to nine workers may invoke a single box landing concurrently. Fig. 2.7b shows the speedup for increasing box concurrency limits relative to the execution time of S-RTS/PTH. This more clearly shows the performance gains by the concurrent box invocations. Of course performance gains by concurrent box invocations are highly application specific. Our implementation merely provides a mechanism for users to increase the exposed concurrency in their applications.

2.7 Optimizing Repeated Synchronization

Another significant optimization is the recognition of the very common combination of a synchro-cell as operand to a star combinator. By replacing this combination with a single entity we quite drastically reduce the required number of memory allocations and deallocations for streams of records which need to be synchronized. The concentration of information in a single entity further allows us to reduce the worst case cost for a single synchronization from $O(N)$ to $O(1)$ when the input stream has strong imbalances in the order of arrival of the types of input records.

2.8 Non-deterministic Feedback

Currently the S-RTS runtime system only provides a deterministic implementation of the feedback combinator where a new record is input into the feedback loop only when all previous record processing activity in the loop has come to an end. In the FRONT runtime system we also provide a non-deterministic feedback combinator which liberally accepts further input records into the feedback loop. This allows for the expression in S-Net of various useful design patterns, such as for flow control and load balancing. It also increases the exposed concurrency and significantly improves performance in some cases.

2.9 Experimental Evaluation

We evaluate FRONT by comparing its performance with that achieved by S-RTS for a variety of applications. More precisely, we compare the following S-Net runtime system configurations:

- S-RTS/PTH: the original runtime system with the PTHREADS threading layer,
- S-RTS/LPEL: the original runtime system with the LPEL threading layer,
- FRONT: the novel runtime system introduced in this paper.

Our experimental system is a 48-core SMP machine with 4 AMD Opteron 6172 “Magny-Cours” processors running at 2.1 GHz and 128 GB of DRAM. Each processor core has 64 KB of L1 cache for instructions, 64 KB of L1 cache for data, and 512 KB of L2 cache. Each group of 6 cores shares one L3 cache of 6 MB. The system runs Linux kernel 2.6.18 with Glibc 2.5. We used GCC version 4.4.6 with the -O3 optimization option to compile all C sources.

We first focus on a fairly small benchmark that deliberately stress tests the runtime system design and implementation through large numbers of records, frequent expansion of dynamically replicated subnetworks and negligible computations within components. At the end of the section we present our findings for a non-trivial (and more representative) S-Net application. A plethora of further experimental data can be found in [2, 3].

2.9.1 Fibonacci Numbers

With the Fibonacci benchmark[1] we compare the performance of the runtime systems in creating and destroying entities and streams as well as the speed at which records are pushed through the entity graph. We do all computing with S-Net

[1] Available at `https://github.com/snetdev/snet-rts/blob/master/examples/fibonacci/fibonacci3.snet`

filters and minimize the influence of input parsing and output formatting by taking only one input record and producing a single output result, i.e. the argument and result of the Fibonacci function. Our implementation follows a divide-and-conquer approach such that all S-Net language constructs are used intensively. The number of created records is proportional to the value of the computed Fibonacci number. Fig. 2.8a shows that FRONT is about 50 times faster than S-RTS for this benchmark; Fig. 2.8b shows the (connected) rate at which records are created and

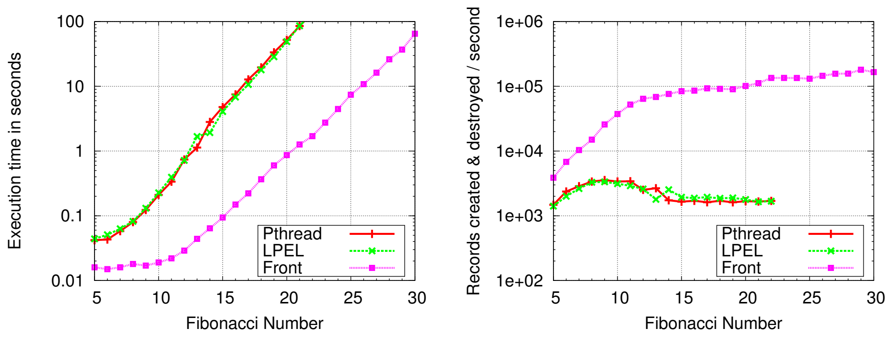

Figure 2.8: Fibonacci benchmark: **(a)** Execution time, **(b)** Record processing rate

destroyed. For FRONT this rate increases strongly up to $Fib(12)$ after which it increases weakly, whereas for S-RTS it diminishes between $Fib(11)$ and $Fib(15)$, regardless of the threading layer.

2.9.2 Cholesky Decomposition

Cholesky decomposition is a linear algebra problem: given a Hermitian positive-definite matrix A, find a lower triangular matrix L, such that $LL^T = A$, where L^T is the transpose of L. We use an implementation by Pāvels Zaičenkovs, University of Hertfordshire, based on the tiled algorithm proposed by Buttari et al [1]. After an initial setup the algorithm repeatedly executes the following seven phases: fan-out, data-parallel computations, fan-in, fan-out, data-parallel computations, fan-in, and sequential consolidation of intermediate results.

We use this application to measure scalability. We stepwise increase the number of processor cores which are available to the runtime system. For this we use the `taskset` program, which permits detailed control of processor core affinity. We incrementally add cores such that they share L3 caches and are part of the same processors and sockets as much as possible. At each measurement step we configure FRONT and S-RTS/LPEL to use a number of worker threads which is equal to the available number of processor cores.

Fig. 2.9a shows measurements 4096 by 4096 double precision floating point matrices using 64 by 64 tiles. This amounts to 32 KB per tile. Hence, two tiles fit into the L1 cache of 64 KB. The FRONT runtime system shows good speedups for

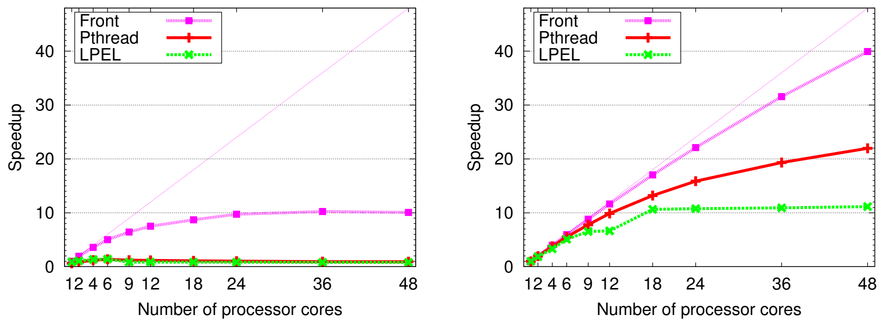

Figure 2.9: Speedup on Cholesky decomposition for two parameter sets: **(a)** Matrix size: 4096 by 4096, tile size: 64 by 64. **(b)** Matrix size: 8192 by 8192, tile size: 128 by 128.

6 cores and diminishing speedups up to 24 cores. The S-RTS runtime shows just minor speedups up to 6 cores and no speedups beyond. Our interpretation of this figure is that the cost to communicate a task to a different core is relatively high compared to the effort required to complete the task. S-RTS suffers from the high overhead to construct threads to execute a task, which involves longer sections of serial code and context switches between different threads of execution. In FRONT no new threads need to be created to execute a new task. The construction of entities which facilitate the concurrent execution of a new task only requires some memory allocation, which therefore is much cheaper. Distribution of a task does not require context switches, but at most one work stealing event. In Fig. 2.9b we increase the amount of data per tile by four while keeping the total number of tiles identical. Hence, the number of S-Net entities remains the same as well; only the time spent in box components increases. Now, S-RTS/PTH also shows reasonable speedups, but less so S-RTS/LPEL. FRONT shows excellent speedup even for 48 cores. Increasing the number of cores from 36 to 48 still improves the performance by 21 percent, which we deem satisfactory considering that the algorithm also contains sequential sections.

2.10 Monitoring

Runtime monitoring is one of the strengths of the LPEL system virtualisation layer [15]. It would be very desirable to combine the FRONT system with these monitoring capabilities in order to analyse the dynamic behaviour of FRONT with the same level of accuracy and scrutiny as we did with the combination of the original S-Net runtime system and LPEL. Unfortunately, we have not had the resources to address this issue within the limits of the ADVANCE project, although we would not expect any issues in doing so beyond the mere engineering effort.

// Chapter 3

Resource Management Server: Single Node

The major contribution of task WP3d, which is to be reported on in this deliverable, are *resource management servers*. These are characterised as system services that dynamically allocate execution resources to executing programs, more precisely to S-Net streaming networks with sequential or data-parallel components. A particular motivation for this active form of resource management is to control the energy consumption of a running application. In a more general context we aim for adapting the amount of used resources to the actual dynamic needs of the application in order to optimise resource utilisation in multi-user and/or multi-application scenarios.

3.1 Approach

Our approach for a single compute node resource server is illustrated by Fig. 3.1. Whereas our initial approaches to hardware virtualisation, both based on LPEL and an FRONT, work with a configurable but otherwise constant number of worker threads, we now relax this restriction and make the number of worker threads variable over the entire program runtime. A dedicated resource server (thread) is responsible for dynamically spawning and terminating worker threads as well as for binding worker threads to execution resources like processor cores, hyperthreads or hardware thread contexts, depending on the architecture being used.

Upon program startup only the resource server thread is active; this is the master thread of the process. The resource server thread identifies the hardware architecture the process is running on by means of the hwloc utility. Optionally, the number of cores or hardware threads to be effectively used can be restricted by the user; this is primarily meant as a means for experimentation, not for production use. Next, the resource server sets up the static property graph, which is to be shared by all worker threads. Once the set up is completed, the resource server launches the first worker thread.

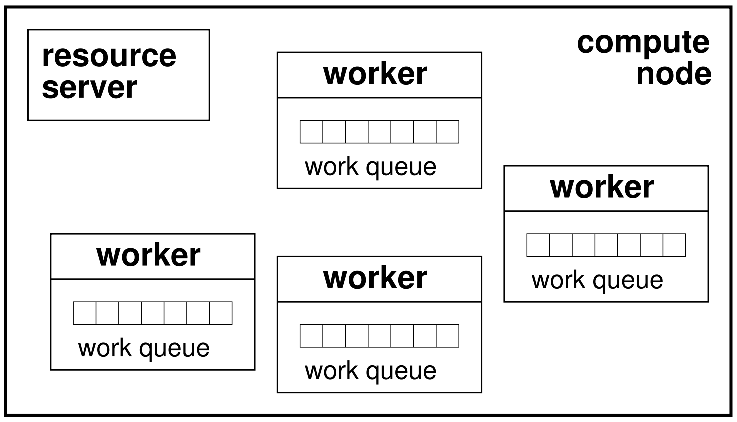

Figure 3.1: Resource server architecture for single node systems

The worker thread executes its standard work stealing procedure. In the presence of an obviously empty work queue it reads the global input stream and, thus, creates the first record in the system. This record is then being processed as described in Chapter 2, which normally triggers the creation of further records to be put into the local work queue.

Creation (and termination) of worker threads is controlled by the resource server making use of two counters, or better *resource level indicators*. The first one is the obvious number of currently active worker threads. This is initially zero. The second resource level indicator is a measure of *demand for compute power*. This reflects the number of work queues in the systems. This is not the same as the number of threads because we have a very special further work queue not associated with any of the workers: the global input of the S-Net streaming network. Thus, the demand indicator is initially set to one. Both resource level indicators are restricted to the range between zero and the total number of hardware execution resources found in the system.

If the demand for computing resources is greater than the number of workers (i.e. the number of currently employed computing resources), the resource server spawns an additional worker thread. Initially, this condition holds trivially. The creation of an additional worker thread temporarily brings the (numerical) demand for resources into an equilibrium with the number of actively used resources. Before increasing the demand the new worker thread must actually find some work to do. In particular during the startup phase of an S-Net streaming network, this usually happens by reading another record from the global input stream. In general, the new thread could alternatively steal existing work from other threads. In any case, once doing productive work, the worker signals this to the resource server, and the resource server increments the demand level indicator, unless demand (and hence resource use) has already reached the maximum for the given architecture.

This procedure guarantees a smooth and quick organisation of the ramp up phase. As a standard scenario we assume the availability of considerable input data on the global input stream as well as a non-trivial amount of initial computation on each of the records. In this case, we effectively overlap (the overhead of) worker thread creation with reading data from global input and its processing. Moreover, only one worker thread at a time attempts to read the global input stream, which avoids costly synchronisation upon accessing this device.

Executing our work stealing model, as described in the previous chapter, potentially leads worker threads to states of unemployment. With the local work queue being empty, no new records on the global input and nothing to steal from other workers, there is nothing left to do for a worker thread. The worker signals this state to the resource server, which in turn reduces the demand level indicator by one. The worker thread does not immediately terminate because we would like to avoid costly repeated termination and re-creation of worker threads in not uncommon scenarios of oscillating resource demand. The worker thread, however, does effectively terminate with a configurable delay following an extended period of inactivity.

3.2 Energy consumption

Effective application-level software control over energy consumption parameters such as clock frequency and voltage is still in its infancy. While such features exist on some architectures, e.g. Intel's Single Chip Cloud Computer (SCC) [10, 14], portability in the availability of features and their control are a matter of the future.

As a consequence, we decided for indirect control over energy consumption and anticipate corresponding support in the operating system for automatic clock frequency and potentially voltage scaling. Most commonly used architectures and operating systems do support this today. Originally motivated by the needs of battery-powered devices like laptops and notebooks server installations likewise use these features nowadays to avoid wasting energy when compute power is temporarily unrequested.

We make use of these facilities by creating worker threads step-wise in a demand-driven manner and bind these threads to run on hardware resources as concentrated as possible. For example, on a dual-processor, quad-core, twice hyperthreaded system we would start at most 16 worker threads. While ramping up the number of active worker threads we first fill the hyperthreads of one core, then the cores of one processor, and only when the number of workers exceeds eight, we make use of the second processor. This policy allows the operating system to keep the second processor at the lowest possible clock frequency or even to keep it off completely until we can indeed make efficient use of it.

While we only ramp up the number of worker threads on-demand as computational needs grow within the S-Net streaming network, we also reduce the number of workers when computational needs decrease. This fits well with our work stealing based runtime system organisation. If a worker runs out of private work, i.e. its work queue becomes empty, it first tries to get hold of the input device and import new records (and thus work) from the global input stream. If that fails, the worker turns into a thief and tries to obtain work from other workers' work queues. If that also fails, it must be concluded that there is at least currently no useful work to do and the worker terminates. By doing so the worker releases the corresponding hardware resource and, thus, gives the operating system the opportunity to reduce its energy consumption by reducing clock frequency and/or voltage or by shutting it down entirely.

While it is fairly straightforward during worker thread creation to incrementally invade the available hierarchical execution resources, worker thread termination as described above is bound to result in a patchwork distribution of active workers over hardware resources over time. This would render the energy-saving capacities of the operating system largely ineffective. To overcome this shortcoming, the resource server continuously monitors the allocation of worker threads to hardware resources and rebinds the workers as needed.

3.3 Multiple Independent Applications

The next step in advancing the concept of resource management servers is to address multiple independent and mutually unaware applications (or instances thereof) running at overlapping intervals of time on the same set of execution resources. Fig. 3.2 illustrates our approach with two applications. The role of the resource management server as introduced in the previous section is split into two disjoint parts: a local resource server per application (process) manages the worker threads of the S-Net runtime system and adapts the number and core-binding of the workers as described before.

The second part of multi-application resource management servers lies with a separate process that we coined *meta resource server*. This meta resource server is started prior to any S-Net-related application process. It is in exclusive[1] control of all hardware execution resources of the given system. Whenever a local resource server has reason to spawn another worker thread, in the current multi-application scenario, it first must contact the meta resource server to obtain another execution resource. The meta server either replies with a concrete core identifier or it does not reply at all. In the former case the local resource server of the corresponding application spawns another worker thread and binds it to the given core. In the latter case the local resource server simply does nothing, which means that the number of execution resources currently occupied by this application remains unmodified.

As said before, the meta resource server is in control of all execution resources and decides which application can make use of which cores. With a single application (instance) the system behaves almost exactly as described in the previous section. The local resource server, assuming that the application exposes ample

[1]We deliberately ignore the underlying operating system here as well as potentially running further applications unaware of our resource management model.

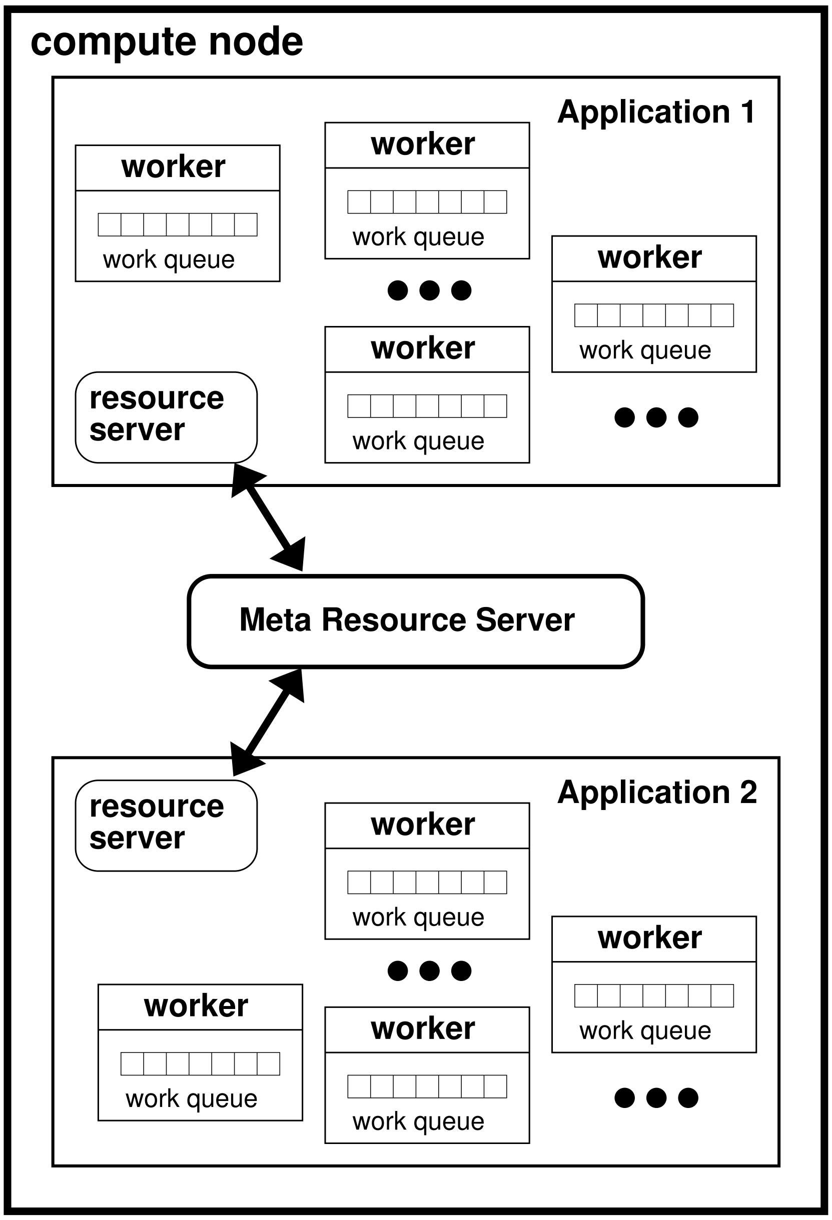

Figure 3.2: Resource server architecture for multiple independent applications on single node systems

```
for (i = 0; i < num_clients; ++i) {
  client = all[i];
  if (client->local_workload >= 1) {
    ++num_positives;
    total_load += client->local_workload;
    portions[i] = 1;
  } else portions[i] = 0;
  remains[i] = 0.0;
}
assert(host->nprocs < total_load);
for (i = 0; i < num_clients; ++i) {
  client = all[i];
  if (client->local_workload >= 2) {
    portions[i] += (client->local_workload - 1)
                 * (host->nprocs - num_positives)
                 / (total_load - num_positives);
    remains[i] = ((double) ((client->local_workload
       (cont.)- 1)
                          * (host->nprocs -
                             (cont.)num_positives))
                          / ((double) (total_load -
                             (cont.)num_positives)))
               - (double) (portions[i] - 1);
  }
  num_assigned += portions[i];
}
while (num_assigned < host->nprocs) {
  p = 0;
  for (i = 1; i < num_clients; ++i) {
    if (remains[i] > remains[p]) p = i;
  }
  if (remains[p] > 0) {
    portions[p] += 1;
    num_assigned += 1;
    remains[p] = 0.0;
  } else break;
}
```

Figure 3.3: Algorithm to divide resources between independent applications proportionally to their resource demand

concurrency, incrementally obtains all available resources on the compute node. Only the additional inter-process communication marginally slows down this process.

Let us look at the more interesting scenario of two applications that both expose sufficient concurrency to make use of the entire compute server by themselves. One is started first and obtains one core after the other until it occupies the entire system.

Now, we start the other application. To do this we must first admit that the meta resource server as well as the local resource servers are scheduled pre-emptively by the operating system. In other words they are not in possession of an exclusive core. And neither are the worker threads. While we guarantee that no two worker threads are bound to the same core at the same time, resource management servers may well interfere with worker execution. With large numbers of cores it may prove more suitable in the future to reserve particular cores for resource management, but the still fairly low core counts representative today, we choose the above solution in order to avoid wasting considerable computing resources. Our general underlying assumption here is that time spent on any form of resource management is negligible compared with the actual computing.

Coming back to our example, all cores are in "exclusive" use by the first application when we start the second application. Hence, we effectively only start the second application's local resource server, which in turn contacts the meta resource server via inter-process communication to ask for a computing core. Since the meta resource server has no such core at hand, it first needs to get one back from another application. To determine the relative need for computing resources the meta resource server compares two numbers for each application:

a) the number of currently allocated cores;

b) the demand for cores, i.e. how many cores the application has asked for.

The quotient between the latter and the former determines the relative need for cores. In our running example and assuming an 8-core system, the first application has a demand quotient of $\frac{9}{8}$ because it currently occupies all eight cores but asked for one more core (we assume ample internal concurrency). The second application has a demand quotient of $\frac{1}{0}$ which we interpret as infinitely high. Thus, a new application that has been started but does not yet have any execution resources has a very high relative demand. The meta resource server goes back to the first application and withdraws one the cores previously allocated to it. The local resource server decides which worker thread to terminate and empties that threads work queue, which is simply appended to another work queue. The worker thread is not preemptively terminated but we wait until it finishes its current box computation. After that the worker thread tries to retrieve the next read license from its work queue, but finds its work queue removed. The thread, thus, signals the local resource server its end and terminates. The local resource server immediately communicates the availability of the corresponding core back to the meta resource server. The meta resource server allocated that core to the second application, which now starts to ramp up its execution.

Assuming that the second application likewise exposes ample concurrency, it will soon ask the meta resource server for another threads. The meta resource server, by means of the demand quotients, step-by-step takes execution resources away from the first application and gives them to the second application until an equilibrium is reached. In order to avoid moving resources back and forth uselessly, the meta resource server makes sure that moving one execution resource from one application to another does not invert relative demands.

If the first application terminates at some point in time while the second is still running, all vacated resources will be moved over to the second application. Fig. 3.3 shows an excerpt of the relevant algorithm to ensure proportional resource distribution among applications.

Chapter 4

System Virtualisation and Resource Management Servers for Multiple Nodes

One of the long-standing goals of the Advance project has been to extend hardware virtualisation to clusters of workstations in a transparent yet efficient way. In the following we describe our approach to extend the per-node resource server described in the previous chapter to multi-node architectures with distributed memory.

4.1 Approach

The resource server approach taken for closely coupled shared memory server nodes, as described in the previous chapter, cannot be extended to loosely coupled, highly asynchronous multi-node systems. Given the strict limitations in time and engineering capacity for this task we strive for a solution that we hoped to be able to realise within the give budget constraints. This approach could best be characterised as an *offloading (software) architecture*, where additional servers essentially act as accelerators that can dynamically be added to and removed from the resource control infrastructure.

On the primary node we essentially run a modified version of the work stealing hardware virtualisation layer described in Chapters 2 and 3. This choice has the first advantage that the global input and output streams of any S-Net streaming network exclusively run on this dedicated node. Other nodes do not actually need to be connected to input and output streams or file systems in any specific way. In other words we do not need to make any assumptions on the input/output capabilities of auxiliary nodes.

In our offloading model the primary node acts as a client; auxiliary nodes as compute servers. While the S-Net streaming network itself runs on the primary node, box invocations are regularly offloaded to compute servers, i.e. instead of actually calling the box implementation function, we send the record alongside an identification of the box (function) to an auxiliary compute server. That compute server actually executes the processing of the given record by the given box (function) and sends the resulting records back to the primary node (or client). The client receives the resulting records and inserts them as usual into the corresponding output stream of the box.

Of course, execution on the primary node does not stall during the offloading of some computation. Otherwise, offloading would be rather useless, both in terms of performance and energy consumption. Having offloaded the computation the runtime system on the primary node continues execution as if the computation were already completed. As S-Net does guarantee certain sequencing rules, offloading creates the need for some reordering capacity to guarantee the orderly behaviour of the streaming network. This, however, is already a solved problem, and we can immediately reuse our solution for concurrent box invocations, as reported in Section 2.6.

Since compute servers must be expected to be moderately parallel internally with multiple sockets and multi-core processors, we must talk about their internal organisation in terms of resource services. Any secondary compute server receives *compute jobs* on a defined network socket. A local resource server dispatches the jobs over the available computing resources. In doing so, it takes into account both load balancing and energy saving concerns, and different policies could be employed that would expose different dynamic behaviour. If energy consumption is a concern, then we use a similar incremental resource utilisation strategy as described in Section 3.2.

Fig. 4.1 illustrates our architecture for distributed system virtualisation and distributed resource management servers.

An important aspect in this design is the management of data. When we said before that records are communicated from the primary node (or client) to one of the compute servers and the records resulting from the corresponding computation are communicated back to the primary node, then we meant records without the corresponding field data. Instead of the actual data referred to by some record, we merely communicate handles to that data. Such a handle consists of the unique node identifier that holds the actual data and the data's address on that node. Before any compute server starts any computation, it first ensures that the data referred to by the record is properly materialised in local node memory. In other words, the data is fetched on demand from the node that owns it. Similarly, data that is created during a computation on one of the compute servers remains where it is, and only the record with handles to that data is effectively communicated back to the primary node. This scheme results in the distributed memories of a collection of compute servers as well as the memory of the primary node to be operated in a form of software cache-only memory architecture. We adopted the general approach from Distributed S-Net [6] and adapted it for our specific needs in the current context.

The cache-only memory architecture creates an avenue towards locality-aware scheduling of computations. A crucial question that we haven't even touched yet

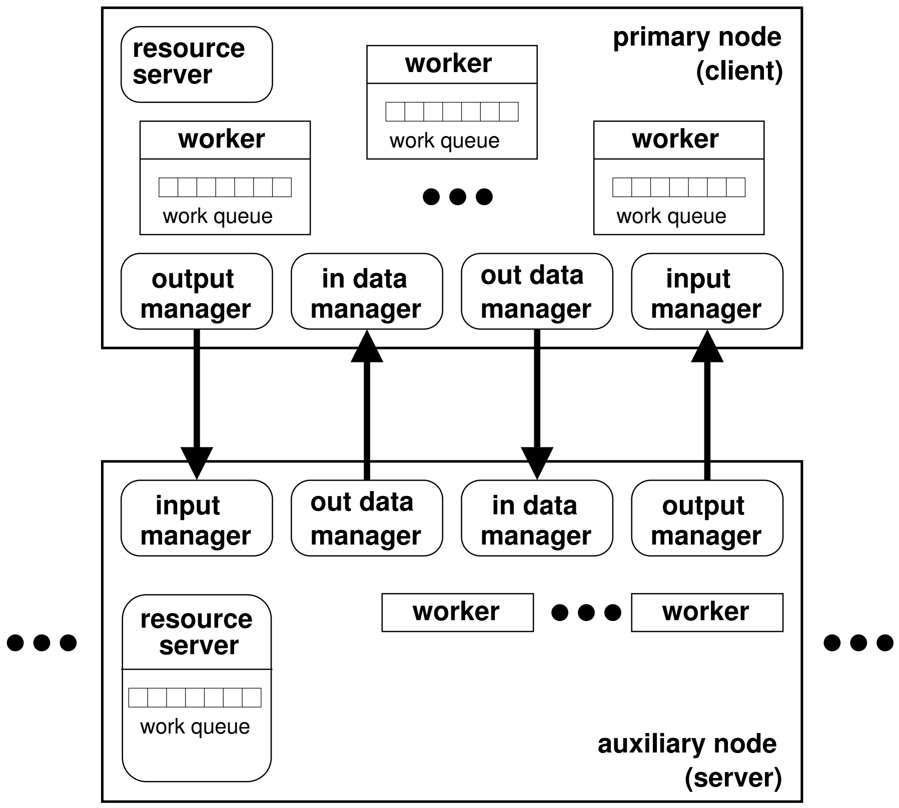

Figure 4.1: Resource server architecture for multiple nodes

is where to offload some computation in the presence of multiple compute servers. In our first approach compute servers inform the primary node's resource manager in certain regular intervals about their load levels. The resource manager in turn aims at a reasonably even workload distribution. Locality-awareness in this context means that the resource server could take into account where the data of the record to be sent off for computing currently is situated. It appears plausible to give some preference to the compute node that already holds significant parts of the relevant data in order to reduce communication requirements. This scenario demands further research into heuristics that find reasonable trade-offs between workload distribution and data locality. Due to limitations in time and resources we have not been able to investigate this direction further.

Our approach opens up another avenue for future research: the combination of the S-Net streaming network technology with data-parallel implementations of boxes, predominantly using SAC. So far, the combination of streaming or data flow style parallelism on the network level with (implicit) data parallelism within individual boxes has suffered from two limitations. Firstly, closely coupled cache-coherent shared memory architectures typically expose only a limited number of concurrent computing resources. In practice, these can typically be utilised in their entirety by either the data flow streaming network level or by only data parallel (non-streaming) implementations. Combining both approaches in this case rather adds technical complexity and, obviously, additional overhead, but cannot generally achieve higher levels of performance. Secondly, the data parallel approach with its inevitable synchronisation barrier at the end requires hardware virtualised computing resources to be available at roughly the same time, which conflicts with the overall streaming model that leads to a rather irregular and independent resource utilisation. With the multi-node offloading approach above we can employ the local resource servers to manage groups of cores and assign them to individual box executions.

4.2 Communication Protocol

In this section we provide the complete protocol for the client-server communication with more details.

4.2.1 Messages from clients to servers

Clients can issue the following message types (here enclosed in double quotes) with the given accompanying parameters:

- `list`
  Requests a space separated list of a computer system identification numbers for the currently managed set of systems, where id zero always represents the local computer system.

- `topology` *`id`*
  Requests a complete hardware topology description of the node/cache/core hierarchy of the system which is identified by number 'id'.

- `resources` *`id`*
  Requests a hardware topology description in HWLOC format (for debugging).

- `access` *`idlist`*
  Communicates to the server that the client can access and make use of the computer resources for all systems in the given space separated list of system identification numbers.

- `local` *`workload`*
  Here workload is a number which represents an estimate for the approximate number of processor cores which could be used to process the local workload.

- `remote` *`workload`*
  Here workload is a number which represents an estimate for the approximate number of processor cores which could be used to non-locally process workload.

- `accept` *`id cores`*
  This is a response to a server grant message which acknowledges to the server that the client will be using the cores on computer system 'id'. The cores are identified by the space separated list of logical core numbers.

- `return` *`id cores`*
  This communicates to the server a list of logical processor core numbers which the client is no longer using on the system identified by 'id'. The list is a space separated list of logical core numbers.

- `quit`
  Asks the server to terminate the connection.

- `help`
  Requests an overview of the available commands and their syntax.

- `state`
  Requests an overview of the resources which have been granted to the client.

- `shutdown`
  Requests the server to close all connections and stop execution.

4.2.2 Messages from servers to clients

Servers can issue the following message types (here enclosed in double quotes) with the given accompanying parameters:

- `systems` *`idlist`*
  Gives a space separated list of machine identification numbers for the currently managed set of computer systems. Zero is used to represent the local computer system.

- `hardware` *`id details`*
  In response to a client topology request this gives a detailed list of information about hardware details for computer system 'id'.

- `grant` *`id cores`*
  This gives the receiving client a license to use the given processor cores on the computer system identified by 'id' to process the client workload. The list is a space separated list of logical core numbers. A client must indicate about each core whether it is going to use it (with an accept message) or not (with a return message).

- `revoke` *`id cores`*
  This requests to the client to stop using the given list of logical processor core numbers. The client should acknowledge this request with one or more return responses.

4.3 Energy consumption

In Analogy to per-node energy control we employ indirect techniques to improve energy efficiency in multi-node and cluster environments. On each node we use the techniques described in Section 3.2. For the multi-node system as a whole we employ a similar*overflow* strategy as between the cores of a single node. Additional nodes are dragged into the execution of an S-Net streaming network on an on-demand basis as computational needs outgrow the already available computing resources. Likewise, complete compute nodes are systematically vacated as computational needs shrink during the execution of a program. This allows the orderly shutdown of remote systems as soon as all data has been migrated away from such a node.

Chapter 5

Summary and Conclusion

This report gives a representative overview on research and development activities in Work Package 3 during the fourth reporting period. As such it deliberately goes beyond the concrete task W3d involving *resource management servers*.

During the (shorter) fourth reporting period we have made major achievements in three distinct but interconnected areas:

1. repositioning hardware virtualisation within the ADVANCE technology stack to the novel S-Net runtime system FRONT instead of two distinct levels for runtime system and hardware virtualisation;

2. developing single-node resource management servers that allow us to actively manage the execution resources of a compute node and by the help of the underlying operating system to save on energy consumption;

3. extending hardware virtualisation to multi-node systems with distributed memories and development of the associated multi-level resource management servers.

The state-of-affairs in the above three focus areas is not the same. Our work on the novel FRONT runtime system, as presented in Chapter 2 has reached maturity and has been validated and evaluated quite intensely. We reported our work to the wider scientific community at the International Symposium on High-Level Parallel Programming (HLPP 2013) and published it in [3].

Our work on resource management servers had been delayed by the developments in the area of FRONT. Our single node solution, as discussed in Chapter 3, has been fully implemented, but performance evaluation on the various ADVANCE industrial use case and beyond is still outstanding. Instead we focused our resources towards hardware virtualisation for distributed memory multi-node systems. We completed the design, as outlined in Chapter 4, but its implementation has been delayed by the end of the ADVANCE project and the corresponding reduction of resources. Completing the implementation and experimental validation of the approach remain future work.

Our progress in the fourth reporting period was also significantly impeded by two organisational issues. Our experienced junior researcher, Merijn Verstraaten, was not available beyond the original end of the project in January 2013. We were able to replace him by a (then) recent MSc graduate, Bert Gijsbers. Unfortunately, he could not be hired until the re-allocation of resources between consortium partners. This was only completed by mid-April, which left us with not more than 5.5 months for effective work. Under these circumstances the reported achievements are (in our opinion) remarkable and were only possible because Bert Gijsbers started working on alternative execution models for S-Net already during his Master research project.

Bibliography

[1] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput.*, 35(1):38–53, January 2009.

[2] B. Gijsbers. An efficient scalable work-stealing runtime system for the s-net coordination language. Master's thesis, University of Amsterdam, Amsterdam, Netherlands, 2013.

[3] B. Gijsbers and C. Grelck. An efficient scalable runtime system for macro data flow processing using s-net. In *International Journal of Parallel Programming, DOI: 10.1007/s10766-013-0271-8*, 2013.

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

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

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

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

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

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

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

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

[12] F. Le Chevalier and S. Maria. Stap processing without noise-only reference: requirements and solutions. *Radar, 2006. CIE '06. International Conference on*, pages 1–4, Oct. 2006.

[13] Kenneth MacKenzie, Philip Hölzenspies, Kevin Hammond, Raimund Kirner, Nga Nguyen Vu Thien, René te Boekhorst, Clemens Grelck, Raphael Poss, and Merijn Verstraaten. Statistical performance analysis of an ant-colony optimisation application in S-Net. In C. Grelck, K. Hammond, and S.B. Scholz, editors, *2nd HiPEAC Workshop on Feedback-Directed Compiler Optimization for Multicore Architectures (FD-COMA'13), Berlin, Germany*. HiPEAC, 2013.

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

[15] V.T.N. Nguyen, R. Kirner, and F. Penczek. Monitoring framework for stream-processing networks. In *HiPEAC Workshop on Feedback-Directed Compiler Optimization for Multi-Core Architectures (FD-COMA'12), Paris, France*, Jan. 2012.

[16] Frank Penczek, Stephan Herhut, Clemens Grelck, Sven-Bodo Scholz, Alex Shafarenko, Rémi Barrière, and Eric Lenormand. Parallel signal processing with S-Net. *Procedia Computer Science*, 1(1):2079 – 2088, 2010. ICCS 2010.

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

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