FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

A Light-Weight Parallel Execution Layer for Shared-Memory Stream Processing

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Daniel Prokesch
Matrikelnummer 0326495

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Priv.-Doz. Dipl.-Ing. Dr.techn. Raimund Kirner

Wien, 09.02.2011

(Unterschrift Verfasser) (Unterschrift Betreuer)

Daniel Prokesch
Schlachthausgasse 23-29/1/115
1030 Wien

"Hiermit erkläre ich, dass ich diese Arbeit selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe."

Wien, am

Kurzfassung

In den letzten Jahren haben sich zunehmend Mehrkernprozessoren im Bereich der Universalrechner etabliert. Um die Vorteile der zunehmend ansteigenden Anzahl von Rechenkernen nutzen zu können, sind Programmierparadigmen notwendig, welche die Ableitung von Nebenläufigkeiten erleichtern bzw. erst ermöglichen. S-Net ist eine deklarative Koordinationssprache, deren Ziel es ist, die Programmierung von Nebenläufigkeit und die Programmierung von Programmlogik zu trennen. Es wird die Koordination von Netzen bestehend aus asynchronen, zustandslosen Komponenten, sogenannten Boxen, die mittels typisierter Datenströme verbunden sind, definiert. Boxen werden in einer konventionellen Programmiersprache geschrieben und haben einen einzigen Eingangs- und einen einzigen Ausgangsdatenstrom. Netzwerke werden in S-Net durch algebraische Formeln ausgedrückt, bestehend aus vier Kombinatoren, nämlich serieller und paralleler Komposition, sowie serieller und paralleler Replikation.

Das Ziel dieser Arbeit ist es, das bestehende Laufzeitsystem dahingehend zu erweitern, dass es zwei Anforderungen genügt. Erstens soll es ermöglicht werden, Informationen über die Ausführungszeiten der Laufzeitkomponenten oder die Auslastung der Kommunikationskanäle zu sammeln. Zweitens soll es die Möglichkeit zur Kontrolle des Schedulings der Komponenten geben.

Durch Recherche von bestehenden Systemen, welche auf Datenstromverarbeitung basieren, und insbesondere deren Laufzeitsystemen, wurden Konzepte für eine neue Ausführungsschicht erarbeitet. Ein *Light-weight Parallel Execution Layer* (LPEL), d.h. eine leichtgewichtige parallele Ausführungsschicht, wurde entwickelt, welche Komponenten, die mittels gerichteter und gepufferter Datenströme kommunizieren, auf User-Ebene verwaltet, wobei besonderes Augenmerk auf die Anforderungen des S-Net Modells gelegt werden musste. Mechanismen, die dem Stand der Technik entsprechen, z.B. nebenläufige Datenstrukturen und Lock-free Algorithmen, wurden bei der Umsetzung verwendet.

Um die Konzepttauglichkeit nachzuweisen, wurde die Ausführungsschicht als Programmbibliothek in der Programmiersprache C entwickelt, und das bestehende S-Net Laufzeitsystem darauf portiert. Experimente mit der neuen Schicht zeigen eine effiziente Ressourcennutzung selbst bei vielen zu verwaltenden Komponenten. Die zur Laufzeit gewonnene Information kann erfolgreich verwendet werden, um den erforderlichen Rechenaufwand der Komponenten zu ermitteln, sowie um anwendungsspezifische Scheduling- und Platzierungsstrategien abzuleiten.

Schlagworte: S-Net, Datenstromverarbeitung, Prozessnetzwerke, Koordinationssprache, User-level Threading, Nebenläufige Datenstrukturen, Shared-Memory Programmierung

Abstract

In recent years, multicore technology became prevalent in the area of general-purpose computing. In order to facilitate the benefits of the steadily increasing number of cores, programming paradigms are necessary that support or even enable the derivation of concurrency. S-Net is a declarative coordination language which aims to separate the concerns of computation and organisation of concurrent execution by defining the coordination behaviour of networks of asynchronous, stateless components (called *boxes*) and their orderly interconnection via typed streams. Boxes are written in any conventional language, connected to the streaming network with a single input and a single output stream. Streaming networks are expressed in S-Net itself as algebraic formulae built out of four network combinators, namely serial and parallel composition, and serial and parallel replication.

The aim of this thesis is to extend the multi-threaded S-Net runtime system to fulfil two requirements. First, to enable the collection of monitoring information such as execution time of components, or buffer usage along communication paths. Second, to provide a way to control scheduling of components.

By analysing existing stream-processing frameworks with respect to their runtime system implementations, concepts for a new execution layer were devised. A *Light-weight Parallel Execution Layer* (LPEL) has been developed, which manages tasks communicating via unidirectional single-producer single-consumer buffered streams in user-space, with special focus on the requirements imposed by the S-Net model. State-of-the-art techniques, like concurrent data structures and lock-free algorithms, have been employed.

As a proof of concept, the layer is implemented as a separate library in the C programming language, and S-Net is ported onto it. Experiments with the new layer show efficient resource utilisation when having to handle many components. The profiling information can be used to calculate computational costs of components as well as to derive application-specific scheduling and placement strategies.

Keywords: S-Net, Stream-Processing, Process Networks, Coordination Language, User-level Threading, Concurrent Data Structures, Shared-Memory Programming

Acknowledgements

First of all I would like to thank my supervisor Priv.-Doz. Dr. Raimund Kirner who made working on the S-Net project possible for me. Furthermore I would like to thank him for his patience and for always being available for lengthy discussions.

I would like to show my gratitude to Dr. Clemens Grelck who provided me a deep insight into the design ideas of the S-Net runtime system, as well as for proofreading the thesis and his valuable feedback.

I am indebted to Frank Penczek, not only for helping me to get a grip on the S-Net runtime system implementation, for providing access to the benchmark machines, and for his administrative support, but also for encouraging me and helping me to preserve my motivation during the hard times of debugging and benchmarking.

Furthermore I would like to thank Prof. Alex Shafarenko from the University of Hertfordshire and the whole CTCA staff for making the work on the S-Net project such a pleasant experience.

I would like to thank Raphael "kena" Poss for providing valuable feedback and ideas for further improvement.

I would like to show my gratitude to Prof. Hermann Kopetz who provided me a working environment to complete my thesis at the Real-time Systems Group of the Technical University of Vienna.

Last, but not least, I owe my deepest gratitude to my family: my wonderful girlfriend Marlene and our daughter Philine. They have supported me and motivated me all along the way.

Contents

Chapter 1

Introduction

In recent years, multicore technology has become prevalent in the area of general-purpose computing. In order to facilitate the benefits of multiple cores, application programs require explicit parallelization. This is in sharp contrast to the benefits of increase of clock frequency and mechanisms to maximise throughput of sequential instructions, which could speed sequential programs up without any additional effort required from the programmer. But parallel programming in the conventional style is difficult, because computation is intertwined with the organisation of concurrent execution, including the decomposition of the problem or the data, and utilising rather low-level mechanisms for communication and synchronisation. There exist different approaches to tackle the difficulty of developing programs that utilise the available parallelism of the hardware. One of these approaches is to separate the concerns of computation and coordination via so-called *coordination languages*.

S-Net [GSS10] is a declarative coordination language and component technology, which enables turning sequential legacy code written in conventional languages into asynchronous components that interact with each other via a stream-processing network. More precisely, S-Net defines the coordination behaviour of networks of asynchronous, stateless components (*boxes*) and their orderly interconnection via typed streams. Boxes are written in any conventional language, connected to the streaming network with a single input and a single output stream. Streaming networks are expressed in S-Net itself as algebraic formulae built out of four network combinators, namely serial and parallel composition, and serial and parallel replication.

1.1 Motivation

There exists a runtime system [GP10] for S-Net, that utilises PThreads (POSIX Threads, [Ins95]) to provide concurrent execution. In this system, each box is mapped to a PThread, as well as the runtime entities that perform the routing of records through the S-Net network at split and merge points. Communication takes place via bounded buffers in shared memory, for synchronisation the primitives provided by the PThreads interface are used.

The aim of this thesis is to extend the current runtime system to fulfil two requirements:

1. Enable the collection of monitoring information such as execution time of components, or buffer usage along communication paths.

2. Provide a way to control scheduling of components.

The motivation behind this is twofold:

First, there are ongoing efforts to equip S-Net with facilities for self-adaptation and reconfiguration of networks [PSG08]. In order to trigger performance-based reconfiguration decisions, which is the working agenda of an ongoing IST FP7 project, runtime performance measurements are required. Second, the possibility to employ different scheduling policies should make the runtime system open for adaptation to the needs of (soft) real-time applications.

Both requirements can be tackled by implementing a layer that elevates the dispatching of S-Net runtime entities from kernel-space (PThreads as operating system threads) into user-space.

1.2 Contribution

In this thesis, a *Light-weight Parallel Execution Layer (LPEL)* is developed that provides the necessary mechanisms of managing tasks in user-space, with a focus on efficient synchronisation and communication, while diligently utilising the available processing and memory resources in a shared-memory multiprocessor setting. Special effort is made to adhere to a modular design, and to allow the collection of profiling information during the execution, while keeping the introduced overhead for the latter low.

The development includes a thorough investigation of general concepts that are involved with user-level task management. The underlying ideas of S-Net are presented, and the relation to the computational model of Process Networks is outlined. Apart from identifying requirements and constraints that the model of S-Net imposes on task management, existing runtime system implementations of Process Networks and other stream processing frameworks are analysed, to explore the design space and to gather ideas that can be adopted for the design of the LPEL.

The design of the layer incorporates light-weight synchronisation techniques to keep the overhead at a minimum. The layer is not exclusively targeted towards the S-Net runtime system, but it has at least the necessary properties so that the S-Net runtime system can be built upon it.

As a proof of concept, an implementation of the layer is provided as a separate library, and the current S-Net runtime system is ported onto it. A comparison of both the variants of the runtime system demonstrates the performance improvements that can be attributed to the LPEL, even with the extensive collecting of profiling information.

1.3 Outline

The remainder of this thesis is structured as follows.

Chapter 2 provides some theoretical prerequisites to different models of computation, S-Net, and the scheduling of processes.

In Chapter 3 different existing implementations of frameworks for streaming networks and runtime systems for Process Networks are reviewed and the applicability of their concepts to the LPEL is analysed. It also gives a brief introduction to the existing runtime system of S-Net.

The exploration of the design space of LPEL as well as the final design decisions are described in Chapter 4. In addition, the chapter describes the concepts of the core components of LPEL and the synchronisation mechanisms in detail.

An implementation is presented as proof of concept in Chapter 5, also describing the integration with the existing S-Net runtime system.

Chapter 6 contains an evaluation of the LPEL, which involves two aspects. On one hand, the overall performance of the LPEL featured S-Net runtime system is compared against the variant without LPEL. On the other hand, the monitoring information is analysed with respect to its usefulness to identify computational costs and performance bottlenecks.

Chapter 7 concludes this thesis, providing directions for further research.

Chapter 2

Theoretical Background

In this chapter, we provide some theoretical background for different models of computation related to stream processing, in particular Kahn Process Networks and S-Net. The model as such poses certain requirements upon the scheduling of processes, and identifying the relations and differences between them will help in making the right design decisions for the user-space task management layer we intend to develop. Then, the concepts of user-level scheduling and kernel-level scheduling are explained. Concurrent data structures constitute the last part of this chapter.

2.1 Models of Computation

2.1.1 Kahn Process Networks

Kahn process networks (KPNs, [Kah74]) are a distributed model of computation, where deterministic sequential *processes* are communicating solely through unbounded FIFO channels. A process may have multiple input and output channels, to which it reads and writes atomic data elements (tokens). Writing to a channel is non-blocking, i.e., it always succeeds and does not stall the process, while reading from a channel is blocking, i.e., a process reading from an empty channel is stalled and only allowed to continue if data is available again on the input channel. Activation of a process in order to consume newly available data is often called a *firing* of a process. Processes cannot test input channels for availability of data without consuming it, and as processes themselves are deterministic, the whole KPN is deterministic such that the channels always contain the same sequence of tokens, regardless of any computation or communication delays. Note that processes can be arbitrarily connected, i.e., the network may contain cycles and multiple channels between two nodes.

2.1.2 Synchronous Dataflow

In the KPN model, a process is not restricted regarding the number of tokens it might write to a channel or read from a channel (as long as there is data available). *Synchronous Dataflow* (SDF, [LM87]) is a more restricted model that is commonly used for signal processing applications. In SDF, when a

process is allowed to perform its computation (i.e., it *fires*), the consumption and production rates of tokens from input and to output channels are fixed and hence known a-priori. Note that while in a KPN a process might conceptually produce an infinite number of tokens during a single firing, this is impossible with SDF.

2.1.3 Scheduling of Process Networks

The term *scheduling* refers to the timely mapping of processes onto available processors. There exist two different approaches to scheduling: *Static scheduling* means that the schedule is determined at compile-time, whereas *dynamic scheduling* means that the schedule is determined at run-time. The computational model restricts, when scheduling takes place: In the restricted model of SDF, a schedule can be determined statically at compile time, whereas in the general model of KPNs this is impossible to do efficiently [Buc93].

A dynamic scheduling algorithm operates on a set of *ready* processes. Whenever a processor becomes available, the scheduler selects a ready process and assigns it to the processor. Two general approaches can be distinguished for deciding which processes are ready at any given point in time.

Data-driven Scheduling

In *data-driven* scheduling, any process that has tokens available at its input channel (which it wants to read from) is ready. Running processes may produce tokens on their output channels, leading to other processes becoming ready. If a process has no input data available anymore, it is no longer ready. Data-driven scheduling lends itself to parallel execution. But a major disadvantage is that a network can execute in an unbounded way without ever producing output. Because the channels are conceptually unbounded, a process can continue producing tokens without being interrupted. Although other processes may become ready by the produced data of the running process, a naive execution will not make progress towards global output and eventually run out of memory.

Demand-driven Scheduling

In *demand-driven* scheduling, the demand for data is the driving force behind the algorithm. Initially, the process producing the global output is the only process being ready. As it has no data available on its inputs, it is blocked, and the demand for data is propagated to the processes producing the needed input. Demand is propagated through the whole KPN until it reaches the global input, which will eventually produce the requested data. As soon as the data has been produced, the process which demanded the data is executed. This procedure continues, until again the output process is in need of new input data. The demand for data repeatedly propagates from outputs to inputs and back, every time inducing a chain of (possibly expensive) context switches. The major disadvantage of demand-driven scheduling is that it implicitly constructs a chain of sequential process execution that is not very suitable for massive parallelism.

Hybrid Approaches for Scheduling

A promising solution for efficient scheduling lies in hybrid approaches. In an early paper [KM77], Kahn and MacQueen have proposed a parallel mode of execution which basically is demand-driven but with the following difference. A process that has produced demanded data is not suspended immediately, but allowed to execute in parallel with the consumer until it has produced a number of tokens in excess on the output channel. Therefore, each channel c is assigned an *anticipation coefficient* $A(c)$. A producer requested through c is allowed to produce an additional $A(c)$ tokens on c after it has fulfilled its demand before it gets stalled. The anticipation coefficient is set as the channel is passed as an input parameter to a new process.

Another solution, proposed by Parks [Par95], is to bound the size of the channels, and to execute the processes in a data-driven fashion. Processes will not only block upon attempting to read from an empty input channel, but also upon attempting to write to a full output channel. This approach is efficient and easily implementable, yet well-suited for exploiting parallelism. A major disadvantage of this approach is, that bounds on the channels can introduce *artificial deadlocks*, i.e., all processes block, and at least one process blocks on a full channel. It is artificial, because with conceptually unbounded channels, this situation cannot occur. Parks introduced an algorithm to resolve the deadlock by identifying a full channel that causes a producer to block and to increase its size.

A more detailed overview of scheduling approaches of KPNs and a refinement of Parks' deadlock resolution algorithm can be found in [BH01].

2.2 S-Net

S-Net [GSS10] is a declarative coordination language based on stream processing. It achieves a near-complete separation of concerns between computation and communication. Asynchronous, stateless components written in conventional languages interact with each other in a streaming network. S-Net defines the coordination of these components, called *boxes*, and their orderly interconnection via typed streams, while leaving the specification and concrete operational behaviour to conventional languages.

The separation of coordination and communication is achieved by restricting a box to be connected to its environment by two typed streams, a single input stream and a single output stream (SISO). Data on the streams is organised as records of label-value pairs. Operationally, a box is characterised as stream transformer function, mapping a single record from the input stream to a (possibly empty) stream of records on the output stream. Boxes execute fully asynchronously, i.e., as soon as a record is available in the input stream, a box may start computing and producing records on the output stream. In order to allow for dynamic reconfiguration of networks, a box has no internal state that it can carry over between two consecutive activations.

The motivation of SISO boxes comes from separating coordination and computation. This way, the box is relieved from questions to what extent to synchronise on multiple input streams, and to which of multiple output streams records are written to. These questions clearly are in the domain of coordination.

Thanks to the restriction to a SISO stream box interface, whole networks can be described through algebraic formulae. Network combinators are unary or binary operators that take sub-networks as operands and construct a network that again has a single input and single output stream. Hence, network construction becomes an inductive process, with boxes as base cases. S-Net provides a total of four network combinators: Static serial and parallel composition of heterogeneous components as well as dynamic serial and parallel replication of homogeneous components. Routing through the network is achieved by a employing typed streams and records.

As a result, algorithmic aspects are encapsulated in the form of a box function, whereas any communication and synchronisation actions are happening in the coordination language. In the following sections, the concepts of S-Net are explained in more detail.

2.2.1 The type system of S-Net

The type system of S-Net is based on non-recursive variant records with record subtyping. A *type* in S-Net is a non-empty set of anonymous record variants, and each record variant is a possibly empty set of named record entries. Record entries are either *fields* or *tags*. The former consist of a label associated with data, while the latter are plain integer numbers. The data of fields is opaque to the coordination layer of S-Net, while the tags are interpreted as integers in the coordination layer as well as within a user-defined box.

To give an example, an S-Net type looks like as follows:

```
{<rectangle>, x, y, dx, dy} | {<circle>, x, y, radius}
```

In this example, the items enclosed by curly braces are record variants, the whole type is the set of both record variants, denoted by a vertical bar. The record entries within angular brackets are tags, whereas the other entries are fields.

Record subtyping

Record subtyping is based on the understanding that a subtype is more specific than its supertype(s). For example,

```
{<circle>, x, y, radius, colour}
```

is a subtype of

```
{<circle>, x, y, radius}
```

where the type containing a circle record variant with the additional colour field is a subtype of the type containing the circle variant without that field. The position of the fields and the tag is not relevant, it is the set inclusion relationship between the record entries of the two record variants which defines the subtype relation. Another example for a subtype relation is

```
{<circle>, x, y, radius}
```

being a subtype of

```
{<circle>, x, y, radius} | {<rectangle>, x, y, dx, dy}
```

where the type on the left hand side containing only circle variants is a subtype of the type on the right hand side, which encompasses both circles and rectangles.

Type Signatures

Type signatures describe the stream-to-stream transformation performed by a box or a network. A type signature is a non-empty set of type mappings each relating an input type to an output type. The input type specifies the records a box or network accepts for processing, the output type specifies the records that may be produced in response. For example, the type signature

```
{a,b} | {c,d} -> {<x>} | {<y>} , {e} -> {z}
```

describes a network that accepts records that either contain fields `a` and `b` or fields `c` and `d` or field `e`. In response to a record of the latter type, records with field `z` are produced, in all other cases records are produced either containing a tag `x` or a tag `y`.

As boxes and networks accept subtypes of record variants, excess fields and tags of a record entry are handled in a specific way: Any field or tag of an incoming record that is not listed in the input type of the box or the network is bypassed and added to any outgoing record created in response, unless that record already contains a field or a tag with the same label. This mechanism is called *flow inheritance*.

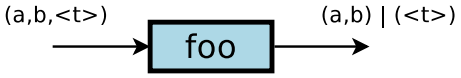


Figure 2.1: An S-Net box has a typed single input and a typed single output stream.

2.2.2 Boxes

Boxes constitute the atomic building blocks of streaming networks in S-Net. Having a single input and a single output stream, they act as stream transformers, i.e., for each record on their input stream, they produce a sequence of records on their output stream, where the length of the sequence is possibly dependent on the input data. Boxes are declared by a unique name and a box signature, for example (cf. Figure 2.1):

```
box foo( (a,b,<t>) -> (a,b) | (<t>) );
```

Box foo accepts records that have fields `a`, `b` and tag `t` and either outputs records containing fields `a` and `b` or records containing tag `t` only. Box signatures are similar to type signatures, except that they only have a single-variant input type, and round brackets are used to show that the order of fields and tags does matter in the context of a box. It is dependent on the actual box implementation how many records are produced and of which of the output variants they are, hence this information can be only determined at runtime.

The box implementation can be in any programming language, it just has to provide an interface to the C language. An example box implementation with the signature above is given as follows, in C:

```
snet_handle_t *foo(snet_handle_t *hnd,
    int *a, mytype_t *b, int t)
{
  /* some computation on a, b, and t */
  snetout(hnd, 1, a, b);

  while( cond(b,t) != 1 ) {
    /* some computation on b, t */
    snetout(hnd, 2, t);
  }

  snetout(hnd, 1, a, b);
  return hnd;
}
```

The parameter `hnd` is an opaque pointer to provide contextual information to the runtime system when making calls to it. Such a call is `snetout` which allows to produce records to the output stream at an arbitrary point during the computation. The second parameter of `snetout` is a number specifying which output type variant is used. Note that producing records possibly could be done within an endless loop, triggered by a single input record, in which case the box would act as data source.

Filter Boxes

For convenience, there exist so-called built-in *filters* which allow simple housekeeping operations for which the knowledge of the field values is not required. These operations include:

- elimination of fields and tags from records
- copying fields and tags
- adding tags
- splitting records
- simple computations on tag values

Synchrocells

Often it is desirable to merge one or more records together into a single one. This cannot be achieved by user-defined boxes, as it would require them to have a state. In S-Net, *synchrocells* are the only stateful boxes, providing the only way to combine records. A synchrocell keeps incoming records which match one of the patterns until all patterns have been matched. The records are then merged into a single one and emitted to the output stream. It provides only storage for one record of each pattern. Records with an already matched pattern are forwarded directly to the output stream. After emitting the merged records, the synchrocell serves as identity function, forwarding all incoming records. If a continuous merging of records is desired, the synchrocell must be placed into a serial replication network combinator. Network combinators are explained in the next section.

2.2.3 Network composition

Complex streaming networks with boxes as atomic building blocks are defined hierarchically in S-Net. This is possible due to the SISO property of boxes and sub-networks. Network combinators either are unary or binary, with sub-networks as operands, and create a compound network that again has a single input and a single output stream. Hence, network composition is an inductive process with boxes as base cases. As a result, whole networks can be expressed by single algebraic formulae rather than by complex wiring lists. Routing decisions are made at split points of the network, they are based upon the type of the sub-networks and the type of the actual record. Furthermore, the networks created by the combinators are acyclic, which eliminates the problem of deadlock due to circular dependencies, which general process networks are prone to. This is a distinctive characteristic of S-Net. Figure 2.2 illustrates the four network combinators.

Serial composition. The binary serial combinator `..` connects two components such that the output of the left operand constitutes the input of the right operand. Serial composition establishes computational pipelines, where records are processed through a sequence of computational steps.

Parallel composition. The binary parallel combinator `|` combines its operands in parallel. An incoming record is sent to exactly one operand, more

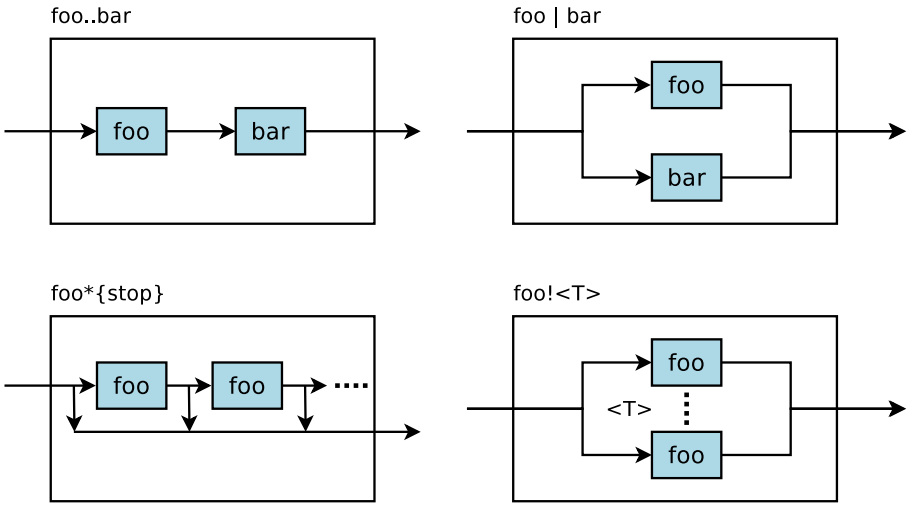


Figure 2.2: Network combinators of S-Net: serial composition (top-left), parallel composition (top-right), serial replication (bottom-left) and indexed parallel replication (bottom-right).

precisely, the operand which type signature matches the type of the record best.

Serial replication. The serial replication combinator `*` replicates its component (left operand) infinitely many times and connects the replicas by serial composition. The right operand constitutes a termination pattern, such that each record that is a subtype of this pattern leaves the replication pipeline through the output stream. Actual replication is demand-driven, hence the network is extended dynamically during runtime.

Parallel replication. The parallel replication combinator `!` replicates its component (left operand) conceptually infinitely many times and connects them in parallel. The right operand is a tag. Each incoming record must have this tag and is sent to the replica with the actual tag value in the record. Again, replicas are created dynamically during runtime.

Deterministic combinators. For all combinators except the serial composition, there exist deterministic variants. Generally, each record that enters a sub-network on the input stream induces a (possibly empty) sequence of records that leave the sub-network on the output stream. The sequences of outgoing records on the output stream, induced by subsequent records on the input stream, are allowed to interleave or be completely reordered. Deterministic variants of the combinators prohibit this interleaving and preserve the order of the outgoing sequences of records. They are denoted as `||` for parallel composition, `**` for serial replication and `!!` for parallel replication.

2.2.4 Relation to KPNs

The computational model of S-Net is somewhat similar to KPNs, considering that the building blocks – boxes in S-Net and processes in KPNs – are not limited with respect to the items they may produce. In S-Net, a box can be executed as soon as there is data available on its input stream. Likewise, KPNs block on input channels if there is no data available.

On the other hand, there are striking differences between these two models:

- A box is invoked once for each record that is present on the input stream subsequently, whereas in KPNs the process itself has to implement a loop that reads subsequent data from its incoming communication channel(s).
- S-Net boxes only have a single input stream and a single output stream. Regular boxes are required to be stateless, whereas KPN processes can have a state. It is important to note that in S-Net split and merge points are in the domain of the coordination language itself.
- Routing of records is managed by the type system of S-Net at split points, not by the boxes themselves.
- S-Net networks are acyclic.
- Parts of an S-Net network are dynamically expanded on demand to facilitate serial and parallel replication.
- S-Net networks are non-deterministic: records that arrive at non-deterministic merge points, are forwarded immediately. This is the source of non-determinism in S-Net compared to KPNs, in which a process is not capable of testing for data availability on its input channels, but can only perform a blocking read (and write) operation on a single channel.

Despite these differences, we will see in Chapter 3.2, that the *runtime system implementation* of S-Net is based on the execution model of Parks [Par95] for KPNs.

2.3 Operating System Threading Facilities

Multithreading refers to the ability of an operating system to support multiple, concurrent paths of execution in the shared address space within a single process. Communication of threads through shared data is simple as a variable or a dynamically-allocated block of memory has the same address in every thread. Thus, pointer-based data structures can be shared between threads without problems, as long as the accesses are properly synchronised.

2.3.1 Kernel- and User-level Threads

There are two broad categories of thread implementation:

Kernel-level threads (KLTs)
Kernel-level threads (or kernel threads) are managed entirely by the operating system. Modern operating systems facilitate the benefits from multiple cores by scheduling multiple threads from the same process simultaneously on the cores.

User-level threads (ULTs)

User-level threads (or user threads) are managed entirely by a user-space threading library or a user-space application, the operating system is not aware of their existence.

KLT vs. ULT

Both types have their advantages and disadvantages. A drawback of a pure ULT strategy (only ULTs in a single process) is that a multithreaded application cannot take advantage of multiple cores, as an OS kernel assigns one process to only one processor at a time and is unaware of the ULTs. Therefore, only a single ULT within a process can execute at a time. As a result, application-level multiprogramming utilises only a single core.

As an advantage, scheduling of ULTs can be application specific. One application may benefit most from a simple round-robin scheduling algorithm, while another might benefit from a priority-based scheduling algorithm. Hence, the scheduling algorithm can be tailored to the application without disturbing the underlying OS scheduler. In a pure KLT approach, the OS scheduler would be ignorant of and as a result disturb the scheduling requirements of the application.

Another benefit of ULTs lies in terms of cost: the transfer of control from one thread to another within the same process does not require a mode switch to the kernel, as with KLTs. The cost becomes even larger if, as common in modern operating systems, KLTs are subject to *preemption*: The operating system may interrupt a KLT at arbitrary points of its execution, typically if it has executed a given amount of time, and reschedule them at a later point. Also, the costs of thread creation and destruction are higher for KLTs than for ULTs.

2.3.2 Threading models

Generally, two possible designs of a *threading model* can be distinguished. In the **1:1** model, each application thread has a dedicated KLT. This situation is depicted in Figure 2.3.

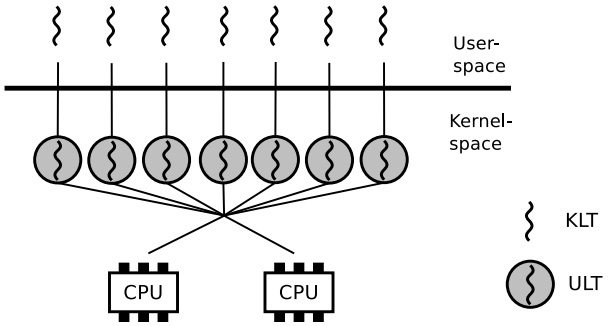


Figure 2.3: 1:1 threading model: Each application thread has a dedicated kernel-level thread (KLT), sketched by a 1:1 mapping from user-level threads (ULTs) to KLTs.

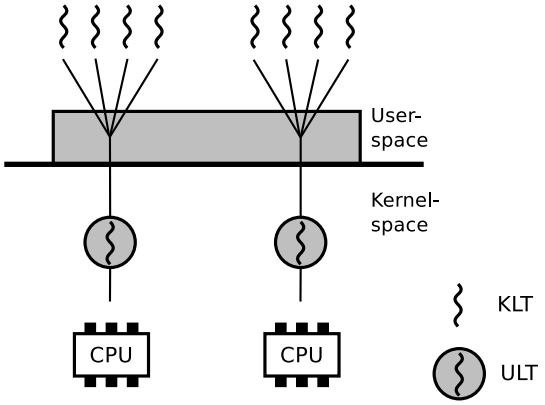


Figure 2.4: m:n threading model: an application is utilising ULTs on top of KLTs, on each KLT a set of ULTs is managed.

To facilitate the advantages of both KLTs and ULTs, an application or a user-space library could handle multiplexing of many ULTs over fewer KLTs, in order to benefit from multiple processor cores. This model is referred to as **m:n** model, illustrated in Figure 2.4. Especially applications with a high level of concurrency, i.e., applications that use a large number of threads, could benefit from such a scheme, as the operating system is not flooded with many KLTs and kept busy with switching between them trying to provide each of them an equal time share of the processors.

Ideally, exactly as many KLTs are utilised as there are processor cores, such that the KLTs are executed in parallel, each on a dedicated processor core. The application itself uses these KLTs to execute ULTs in parallel on top of them. Figure 2.4 depicts the case where an application utilises two KLTs to manage several ULTs on top of each KLT. The ULT management layer is depicted as grey rectangle.

ULTs of an application might also depend in a specific way on each other, such that the user-level scheduler of an application can exploit these dependencies by scheduling ULTs *cooperatively*. As opposed to preemptive scheduling, a cooperative scheduler does not preempt a ULT during execution. Instead, the ULT has to *yield* to allow other ULTs to be executed.

Thus, a disadvantage of ULTs is that if a ULT performs a blocking system call, e.g., an I/O operation where the whole KLT is blocked, as a result, all the other ULTs on that KLT are delayed. One solution is to wrap blocking system calls in helper functions, which issue an asynchronous non-blocking system call, execute other ULTs and periodically check if the call has finished. This technique is often referred to as *jacketing*.

A more comprehensive discussion about OS threading facilities can be found in any actual textbook about operating system design, e.g., [Sta09].

2.4 Concurrent Data Structures

Shared-memory multiprocessors concurrently execute multiple threads of computation which communicate and synchronise through data structures in shared memory. The efficiency of these data structures is crucial to performance, but concurrent data structures are generally far more difficult to design than their sequential counterparts. The challenge arises in designing *scalable* concurrent data structures that continue to perform well as the number of threads accessing these data structures is increasing. Before continuing, we give two definitions in the context of parallel programming:

Speedup

Let T_1 be the execution time of an application when run on a single processor and T_P be the execution time when run on P processors. Then, the speedup of an application when run P processors is given by the ratio

$$S_P = \frac{T_1}{T_P} \, .$$

Ideally one wants *linear speedup*, i.e., when using P processors a speedup of P is achieved.

Scalability

Applications whose speedup S_P grows with P are called *scalable*, i.e., if $T_P < T_{P-1}$, $\forall P > 1$.

2.4.1 Locking

The traditional approach to keep a shared data structure in a consistent state is to use a mutual exclusion *lock* (or *mutex*). At any point in time, the lock is held by at most one thread which is then allowed to read and write the shared data. If a thread wants to access the shared data structure currently *locked* by another thread, it must wait until the lock is released again.

Regarding performance, a lock introduces a *sequential bottleneck*, i.e., at any point in time, at most one operation protected by a lock is doing useful work. The performance impact of sequentially executed parts is illustrated by a simple formula based on Amdahl's Law [Amd67]. Let b be the fraction of the program that needs to be executed sequentially. If the program requires 1 time unit when executed on a single processor, then on P processors the sequential part takes b time units, and the concurrent part takes $(1 - b)/P$ time units in the best case. So for the speedup S_P, following inequality holds:

$$S_P \leq \frac{1}{b + \frac{1-b}{P}}$$

The best achievable speedup is given by the limit $P \to \infty$, such that

$$S_\infty = \lim_{P \to \infty} S_P = \frac{1}{b} \, .$$

For example, if 10% of the application have to be executed sequentially, the best possible speedup if executed on 10 processors is 5.26, and the best achievable speedup is 10. Reducing the sequentially executed code in the context of

locking means to employ *fine-grained* locking schemes, i.e., using multiple locks of small granularity (small number of instructions within the critical section) to protect different parts of the data structure.

Another problem of the lock-based approach arises if the thread holding the lock is delayed, e.g., by being preempted by the operating system. Then, all other threads that want to acquire the lock are delayed as well. This is called *blocking*, and can be avoided by *non-blocking* algorithms, which, by definition, do not use locks.

Two-lock queue algorithm. An example of fine grained locking is the two-lock concurrent queue algorithm of Michael and Scott [MS96]. It is implemented as a singly-linked list, providing an enqueue and a dequeue operation. Instead of using a single lock for protecting the whole queue upon access at either end, two separate locks are employed for the head and the tail, allowing concurrent dequeues and enqueues as never both the head and the tail pointer have to be accessed within one operation. This is achieved by always keeping a "dummy" node at the head, such that always at least one node is contained in the queue. For enqueueing a node, only the tail pointer needs to be accessed: the next pointer of the tail node is set to the new node, and then the tail pointer itself is set to the new node. For dequeueing a node, only the head pointer needs to be accessed, which is the dummy node. If its next pointer is NULL, the queue is empty. Otherwise, the value of the node pointed to by the dummy's next pointer is returned, and the node becomes the new dummy node. The head pointer is set to the new dummy node, and the previous dummy node can be de-allocated. Note that both the head and the tail pointers point to the same node (the dummy node) if the queue is empty. Enqueues and dequeues can be concurrent because dequeues only read the next pointer of the dummy node whereas enqueues only write it. There is no need to handle the special case if the queue is empty before or would become empty after the operation, requiring synchronisation between enqueues and dequeues. Nonetheless, multiple enqueuers have to synchronise upon the tail lock, dequeuers upon the head lock.

2.4.2 Wait-, Lock- and Obstruction-freedom

The defining characteristic of a non-blocking algorithm is that stopping a thread does not prevent the rest of the system from making progress. To put it more formally, various non-blocking progress conditions can be distinguished:

Wait-freedom
: A wait-free operation is guaranteed to complete after a finite number of its own steps, regardless of the timing behaviour of other operations. This is the strongest progress condition.

Lock-freedom
: A lock-free operation guarantees that after a finite number of its own steps, *some* operation completes. Although the operation is subject to starvation, system-wide progress is guaranteed.

Obstruction-freedom
: An obstruction-free operation is guaranteed to complete within a finite

number of its own steps after it stops encountering interference from other operations (live-lock is possible). This is the weakest of the progress conditions.

Atomic Memory Operations

Non-blocking algorithms often are based on atomic memory operations available on the processor's instruction set. The fetch-and-increment, fetch-and-decrement, atomic-swap, and compare-and-swap (CAS) are the most prominent of such instructions. For example, the fetch-and-increment operation can be used to build a concurrent counter: it atomically loads from a memory location, and writes the incremented value back to the location, while the value read from the memory location is returned to the caller. The CAS operation atomically loads from a memory location, compares the value read to an expected value, and stores a new value to the location only if the comparison succeeds. The return value indicates if the substitution succeeded, either by a boolean value or by returning the value read from the memory location. A typical signature of the CAS operation is as follows:

```
boolean CAS(memory_loc, expected_val, new_val)
```

Most non-blocking algorithms involve a loop that attempts to perform an action using one or more CAS operations, and retries when one of the CAS operations fails. A failure of the CAS operation indicates that another thread has changed the value stored at the memory location in the meantime. As many data structures are organised as linked items, changing the data structure involves exchanging pointers. Such data structures are prone to the *ABA problem* as described below.

The ABA Problem

Algorithm 1 sketches a broken lock-free stack implementation. The first operand of the CAS operation denotes the memory location (a pointer to the top of the stack in this case), the second operand the expected value, and the third operand the new value which should be written to the memory location.

Assume the following execution, depicted in Figure 2.5, taken from [AR06]. Initially, the stack contains items A, B, and C. Thread 1 wants to pop an item from the stack, but just before it reaches the CAS operation, Thread 2 pops A and B from the stack, and pushes A back onto the stack. Thread 1 continues, and as it still sees A at the top, it exchanges A with previously read B, which clearly corrupts the linked stack.

The solution to the ABA problem is to never reuse node A. In a garbage-collected environment is is simply a matter of not recycling nodes, i.e., once a node has been popped, it is never pushed again. The garbage collector will check that the memory of node A is not recycled while there are references to it.

Without garbage collection, the solution is to make node A slightly different each time (ABA'). This can be accomplished by appending a serial tag to the reference to A. But this requires the CAS operation to exchange two memory locations atomically, one for the pointer and one for the incremented serial number. Such a CAS operation handling two memory locations is referred to

Algorithm 1 Broken lock-free stack implementation, subject to the ABA problem.

PUSH(n)

```
1  node *top
2  repeat
3      node *tmp = top
4      n.next = tmp
       // if the value in top (memory location) equals tmp,
       // n is written to top and CAS returns TRUE
5  until CAS(&top, tmp, n)
```

POP()

```
1  node *new-top, *n
2  repeat
3      n = top
4      new-top = n.next
5      // ABA problem may arise here!
6  until CAS(&top, n, new-top)
7  return n
```

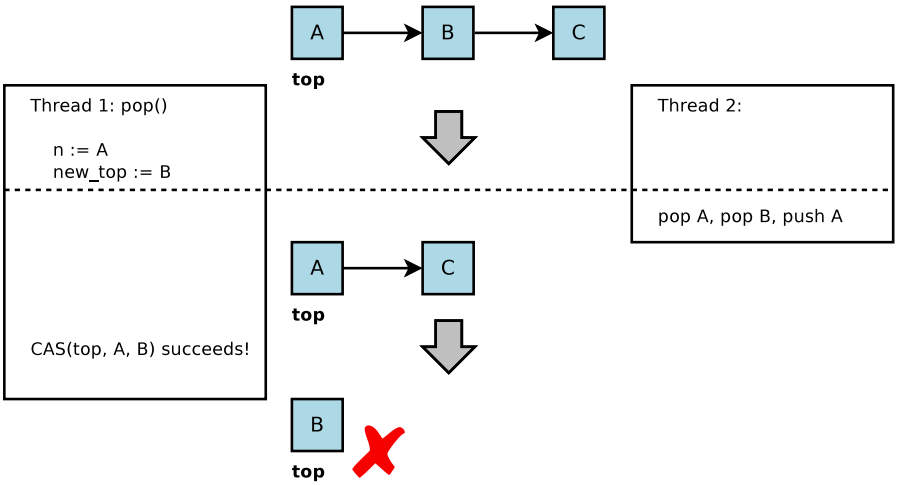


Figure 2.5: Execution of the broken stack, demonstrating the ABA problem. The example is taken from [AR06]

double compare-and-swap, or CAS2. For example, a CAS2 operation might have a signature as follows:

```
boolean CAS2( memory_loc,  expected_val,  new_val,
              memory_loc2, expected_val2, new_val2 )
```

It is similar to the CAS operation, except that two memory locations are atomically exchanged. Only if both memory locations contain the expected values, the new values are written to the memory locations, and the operation returns TRUE. Otherwise, the memory locations are not written to and the operation returns FALSE.

A comprehensive survey of concurrent data structures is given by Moir and Shavit [MS04].

Chapter 3

Related Work

3.1 Existing Process Network Runtime Systems

There exist a variety of runtime systems that implement execution of process networks. In this section, we will describe some in more detail. The selection was motivated by the characteristic features they expose which are of interest for the implementation of the Light-weight Parallel Execution Layer. Therefore, we restrict our investigation to systems which perform scheduling dynamically, i.e., scheduling of processes is done at runtime as it is also done in S-Net.

3.1.1 Nornir

The *Nornir* runtime system of Vrba et al. [VHG+09a, Vrb09] implements the algorithm of Parks [Par95] for the execution of Kahn Process Networks (KPNs). Communication channels between processes are bounded, making execution possible in finite space. Processes are executed if they are ready, i.e., they are not blocked by trying to read from an empty channel or trying to write to a full channel.

An early implementation was built on top of native operating system mechanisms, such that each Kahn Process (KP) was a kernel-level thread. Channels were protected by mutexes and condition variables were used as the sleep and wakeup mechanism for threads. However, evaluation of this approach [VHG09b] showed that facilitating user-level threads instead for KPs executed on top of a few kernel-level threads is considerably more efficient.

Upon startup of the KPN, a specified number of kernel-level threads, referred to as *runners*, are created and scheduled by the operating system onto the available processor cores. KPs are then distributed among the runners. Each runner has a private *run queue* of ready KPs, and is employing a work-stealing scheduling policy. If the private run queue of the runner is empty, it tries to steal KPs from a randomly chosen runner. A run queue is protected by a single mutex, which might be problematic on machines with many cores, as acknowledged in [VHG+09a]. Runners fetch KPs from the head of the private run queue, and as processes becoming ready are enqueued at the tail of the queue, this results in a FIFO order of task scheduling. Other runners attempting to steal ready KPs dequeue them from the tail of the victim's run

queue, and as a result, most recently unblocked KPs are stolen from the owning worker.

Communication and synchronisation between KPs is handled as follows. Each communication channel is protected by a (kernel-level) mutex. As a result, KPs cannot access the channel concurrently. To avoid delaying the whole runner while waiting on the release of the channel mutex, a busy-wait strategy is employed: the KP yields to the scheduler and is put at the end of the run queue. This allows the runner to execute other KPs from the run queue, and eventually the previously descheduled KP will be dequeued and executed by the runner, trying to access the channel again.

KPs can block on communication channels upon a read or a write operation. They yield execution and the runner does not put them back into the run queue. If a KP q unblocks a previously blocked KP p then p will be put onto the run queue where it has been previously executed, which is not necessarily the run queue of q's runner. Also, if a p and q are on different runners, then p is inserted at the head of the owner's run queue. Hence, the only means a KP can migrate to another worker, is only by being stolen. This strategy avoids too frequent process migration among runners and achieves active load distribution instead of making runners to look for more work.

In order to resolve artificial deadlocks, which arise from the problem of bounded communication channels and cyclic networks, Nornir facilitates a centralised deadlock detection and resolution algorithm. The algorithm is invoked each time a KP would block on another KP on a send or a receive operation. The centralised graph structure is protected by a single mutex, and KPs that cannot acquire it, yield to the runner like with the communication channels.

To conclude, Nornir is an efficient runtime system for the execution of KPNs. But the communication channels and run queues, which are protected by single locks, are subject for further optimisation.

3.1.2 YAPI

YAPI [dES+00] is a programming interface aimed towards modelling signal processing applications as process networks. YAPI is not a pure KPN implementation, as it extends KPN semantics by a channel selector, which allows a process to block on a (limited) set of channels. The select operation returns the channel on which the next communication action can be completed. This introduces non-determinism, as channel selection is performed dynamically at runtime. There exists a runtime library written in C++, but the paper [dES+00] gives very few implementation details. The library is downloadable from `http://y-api.sourceforge.net/` (Jan 2011), but unfortunately it does not make use of hardware parallelism and is not maintained anymore.

3.1.3 FastFlow

FastFlow [ADM+09] is a parallel programming framework for multi-core platforms based upon non-blocking lock-free synchronisation mechanisms, particularly targeted to the development of streaming applications. The authors observe that especially in shared-cache multicore architectures traditional programming models limit scalability severely. These limitations arise from atomic memory transactions, that exhibit a rather high latency and tend to pollute the

shared memory hierarchy. They argue, that those operations are not strictly required when concurrent threads operate in a pipeline fashion, because data can be streamed from one-stage to the next using fast lock-free single-producer single-consumer (SPSC) queues, as described by [GMV08]. The FastFlow framework extends this lock-free SPSC queue approach from simple pipelines to complex streaming networks, providing low-level constructs such as multiple-producer multiple-consumer (MPMC) queues and a parallel lock-free memory allocator (MA), and higher-level constructs like programming skeletons for task-farms, or the master-worker pattern.

FastFlow builds upon the threading facilities of the operating system, using kernel-level threads for its application threads. This could have a negative impact on scalability for very large streaming networks.

3.2 The S-Net Runtime System

As the S-Net runtime system will be the experimental platform for the Lightweight Parallel Execution Layer, it is described in more detail in this section.

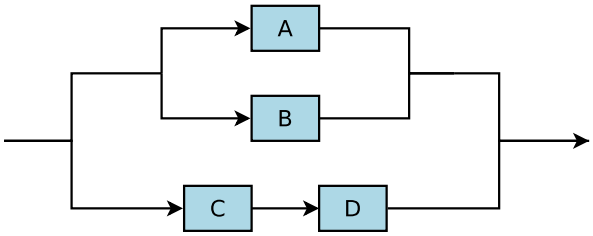
The current multi-threaded runtime system implementation [Pen07, GP10] of S-Net is targeted at shared-memory architectures and provides means to distribute parts of an S-Net network over a network cluster of machines. Latter is based upon an extension of the S-Net core language and is called *Distributed* S-Net [GJP09], but this extension is not discussed here.

The runtime system is implemented in the C programming language and makes use of kernel-level threads via the POSIX Threads (PThreads [Ins95]) programming interface to provide multiple application threads of execution. In the runtime system, components of a streaming network are mapped to threads, such that not only each box operates in the context of a thread, but also each split- and merge point is operating as separate thread, performing the routing of S-Net *records*. More precisely, there exists a set of *entities* such that each instance of an entity is executed in its own kernel-level thread.

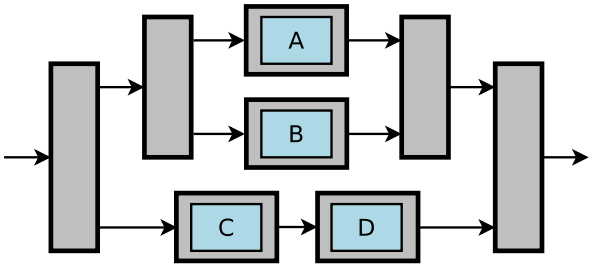
3.2.1 Mapping of S-Net Components onto Threads

Figure 3.1 gives an example of a mapping of an S-Net network onto runtime system threads. Note that in this simple example for the mapping of four boxes twice as many threads are created. Threads (instances of entities) communicate through unidirectional *streams*, which are implemented as bounded circular buffers. Through these streams, references (pointers) to memory locations in the shared address space are passed, where the actual records are stored. Synchronisation takes place through the primitives provided by the PThreads API, namely *mutexes* and *condition variables*. The buffers are accessed mutually exclusive, and a thread is suspended if it performs a read operation on an empty stream or if it performs a write operation on a full stream, by waiting on a condition variable. If a thread writes to an empty stream or reads from a full stream, it signals the appropriate condition, possibly making suspended threads ready again. This pattern constitutes a solution for the producer-consumer synchronisation problem (with bounded buffers) with the concept of *monitors*, as described in, e.g., [Sta09].

Apart from *data records* containing the records as perceived by the S-Net coordination layer and described in Section 2.2, there exist other types of *con-*



(a) Network as described by the algebraic formula `(A|B)|(C..D)`.



(b) Manifestation in the runtime system.

Figure 3.1: Mapping of an S-Net network onto runtime-system entities.

trol records that are communicated by entities. Among theses are records that allow for deterministic combinators, registration of new input streams (e.g., at merge points of dynamic replication components), or orderly termination of the streaming network. Entities also have to incorporate mechanisms to support flow inheritance for data records.

Each entity executes, after some initialisation code, a main loop, in which it reads a record from its input stream, performs some action dependent on the type of the record, and writes record(s) to its output stream(s). The *Collector* constitutes a special case because it acts as a merge point and as such does not only have a single input stream but a set of input streams, as described below. The set of different entities is fixed and can be classified by the number of input and output streams.

Single-input single-output (SISO) entities are:

Box. A box entity has a single input stream and a single output stream. It is a wrapper to the user-supplied box function such that upon arrival of a data record, the box function will be called with the relevant record data.

Filter. A filter performs simple functionality like splitting or duplicating records, or adding, stripping or duplicating fields or tags.

Synchrocell. This entity is responsible for merging two or more records. After successful merging (synchronisation) and writing the compound record to the output stream, the synchrocell sends a synchronisation record to its output, containing a reference to its input stream. Afterwards, it terminates.

There exists only a single type of merge point, with multiple inputs and a single output:

Collector. The collector entity has to listen for newly available data on its input streams. Dependent of its mode of operation, i.e., whether is part of a deterministic sub-network or not, it either has to listen to all of its input streams or just a subset of them.

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

Parallel. The parallel entity implements the router part of the parallel composition operator in S-Net.

Star. For serial replication, at the beginning and after each stage of the serial replication pipeline there is a star entity, that either routes records to the collector for leaving the pipeline or to the next instance (which is also created if it does not exist). Newly created instances are registered at the collector, i.e., a record containing the address of the output stream of the created instance is sent to the collector. Hence, for a serial replication pipeline of depth N, for a single box $2N + 2$ threads are created: N box entities, $N + 1$ star entities and one collector.

Split. Parallel replication is handled by the split entity. Dependent on the specified tag value of an incoming data record, it sends the record to an existing instance matching this value, or, if it does not exist yet, to a newly created instance for that value. The split entity also is connected to the corresponding collector directly to be able to register new instances like the star entity.

3.2.2 Execution model

Although the S-Net model is quite different from the KPN model, the runtime manifestation of a S-Net resembles an extended, acyclic and dynamically reconfigurable KPN executed with bounded buffers according to the proposal of Parks [Par95]. The extension lies, like with YAPI (cf. Section 3.1.2), in introducing non-determinism by allowing for testing on input channels. As networks are acyclic, deadlock detection and resolution due to bounded buffers is not an issue. Although S-Net boxes do not have a state, the runtime system entities are stateful, like KPs.

3.2.3 Desired Extensions and Adaptations of the Runtime System

As S-Net networks potentially can grow very large due to replication, the excessive use of kernel-level threads will surely impact scalability and overall system performance. Furthermore, scheduling is completely within the scope of the operating system, which is undesirable, for example, for applications with real-time constraints. Also, the fact that each application thread is executed as kernel-level thread makes profiling execution traces difficult. Execution time of components can hardly be measured in a convenient way as the operating system can preempt the threads at arbitrary points in time. If each thread outputs its profiling information in a separate resource, e.g., file, the system

is flooded with concurrent I/O requests, which have to be serialised. On the other hand, if a file is shared, mutual access also has to be ensured. Both options result in a sequential bottleneck, severely affecting scalability.

Hence, extensions to the S-Net runtime system are desired, that

(a) allow for profiling execution of S-Net networks, and

(b) enable application-specific scheduling policies

The Light-weight Parallel Execution Layer as described in the next chapter tackles the challenge to provide these extensions.

Chapter 4

The Light-weight Parallel Execution Layer

In this chapter, the design of the Light-weight Parallel Execution Layer (LPEL) is outlined. First, the requirements are gathered, and the design space is characterised. Then, the overall architecture and each of the core components of the LPEL is described in detail.

4.1 Requirements and Design Space

The main purpose of the LPEL is to provide an efficient and flexible execution platform for stream processing applications in a shared-memory setting. Although it is primarily targeted towards the S-Net runtime system, it can be used without S-Net as a basis for other streaming applications, e.g., process networks, as well. The LPEL at least provides the necessary properties such that the S-Net runtime system can be built on top of it. Hence, most of the properties of the S-Net runtime system execution model are adopted. For example, communication will also take place by asynchronous message passing through bounded buffers. In Chapter 5, the required modifications of the runtime system to fit on top of the LPEL are discussed.

The desired extensions of the S-Net runtime system are tackled by providing a layer for user-space thread management, similar to the work of Vrba et al. [VHG+09a]. First, by using user-level threads for S-Net entities, control of their scheduling is elevated from the operating system to the application. Scheduling must remain adaptable to the needs of an application. For example, scheduling requirements for an application with real-time constraints are different than for scientific simulations. Therefore, using a fixed scheduling policy is not an option.

Second, user-space thread management makes profiling executions of S-Net networks much easier. As the knowledge of when execution of a user-level thread is started and stopped is available to the layer, time-stamping these events will provide reasonable execution-time estimates. In addition, stream communication is required to be monitored. As profiling is intrusive to the execution, care must be taken to introduce as little overhead by monitoring as possible. It is also desired to be able to switch off monitoring completely.

Of course, the LPEL must provide good scalability and make use of the

availability of processor cores. Hence, multiple kernel-level threads must be employed, leading to an **m:n** threading model (cf. Section 2.3). Scalability must be ensured by employing concurrent data structures and lock-free techniques where possible and beneficial. This applies to mechanisms regarding synchronisation between user-level threads as well as synchronisation between kernel-level threads.

As the resulting layer will be similar to the runtime system for KPNs described in Section 3.1.1, the ideas and experiences of that project can be used and improvements, e.g., for synchronisation mechanisms, incorporated. Additional requirements of the LPEL include:

- Support for non-determinism by testing of availability of new data on input channels (required at merge points)
- Dynamic (de-)construction of the streaming network during runtime
- Provide the possibility to adapt the scheduling policy to the needs of the application.

What is out of scope of this work, is implementing elaborate scheduling and placement strategies. The architecture and modules are provided, but only simple policies are used that do not use any information about the application.

4.1.1 General Concepts

Like the existing multi-threaded S-Net runtime system, the LPEL builds upon the scheduling model of Parks [Par95] for process networks: Tasks are connected by (uni-directional) streams, which are modelled as bounded buffers. They are suspended from execution upon reading from a full stream and writing to an empty stream. This model allows for an easy implementation and lends to parallel execution.

Bounded buffers provide a mechanism for back-pressure, but can lead to artificial deadlock in circular networks. As S-Net networks are strictly non-circular, a deadlock resolution like the one described in [Par95, BH01] is not necessary for them.

The main difference of the LPEL compared to the existing multi-threaded S-Net runtime system lies in the threading model. While the latter incorporates a 1:1 threading model (cf. Section 2.3), in the LPEL the tasks are not directly executed as operating system threads, but executed as user-level threads in the context of a *worker*. As LPEL tasks only require specific synchronisation constructs, task management is tailored to these constructs which can be implemented efficiently.

Architecture

The architecture of the LPEL is depicted in Figure 4.1. The bottom layer contains all lower-level mechanisms and services that the LPEL builds upon. These include kernel-level threading, assigning a KLT to a specified core (setting the thread's *core affinity*) atomic memory operations and the ability to perform context-switching from user mode.

On top of the architecture stack is the S-Net runtime system. It uses the facilities that the LPEL provides for (user-space) thread management and

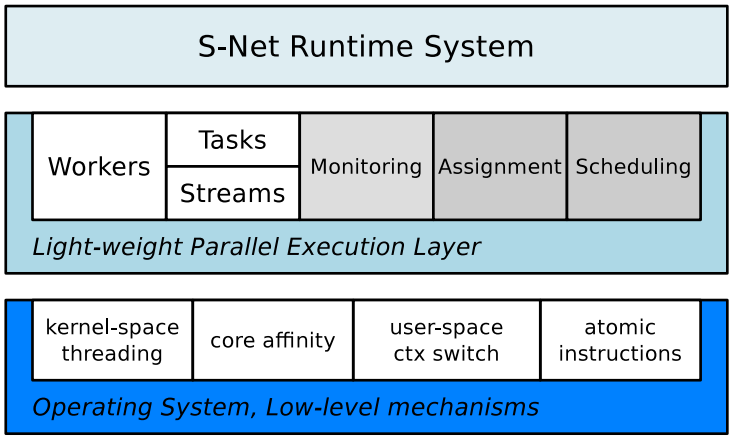


Figure 4.1: The architecture of the LPEL. It builds upon the low-level mechanisms for threading that the operating system provides as well as hardware-specific atomic operations. The LPEL is the substrate for executing process networks, particularly S-Net.

communication between them. Also configuration and adaptation of various LPEL modules is possible from the uppermost layer.

The middle layer depicts the LPEL with its core modules. These are described in detail in the following sections.

4.2 Workers

One of the key ideas of the LPEL is that there exist *workers* which run in parallel and manage disjoint sets of *tasks* in user-space. Tasks are the processing nodes of the stream processing network to be executed. The number of workers depends on the processing resources, i.e., ideally for each processor one worker exists. Workers themselves are scheduled by the operating system, but in an ideal setting the operating system is restricted to always execute a specific worker on the same processor. We say, the worker is *pinned* to a core. Keeping a worker and related data on the same processor helps to preserve cache locality and avoid the cost for thread migration. If there exist as many pinned workers as processors, we have a bijective mapping from workers to cores. In order to keep the synchronisation overhead between workers low, special care is taken to have as little data as possible shared or concurrently accessed by the workers.

In the most general of such a workers-tasks pattern, the tasks can be arbitrary chunks of work. They can be stateful or stateless, have their own execution stack or use the same stack as the worker, etc., all which depends on the requirements of the environment. Although the specific characteristics of LPEL tasks are described in the next section, it can already be mentioned here that each LPEL task has its own execution stack. As tasks are managed entirely in user-space within an operating system thread, switching between tasks does not involve expensive context switching to and from the operating-system kernel. This effectively constitutes a two-level scheduling scheme in

which the OS kernel schedules the worker threads (in kernel-space) and the workers schedule their tasks (in user-space).

One advantage of such a two-level scheduling scheme is, besides reduced context-switching overhead, that the workers have control when and how often to dispatch their tasks, and this flexibility allows us to assign priorities to tasks, dependent on a given scheduling policy (see Section 4.2.3).

Another advantage is that the operating system is not swamped with kernel-space threads that might be all active at the same time, as it would be the case if a task is executed directly as a kernel-space thread. Instead, the number of workers effectively defines the level of concurrency exploited by the execution layer, potentially utilising the available parallelism automatically, if each worker runs on a processor core and the workload is distributed on the workers equally.

The fact that each worker manages its own set of tasks and all task-sets are disjoint helps to achieve a good caching behaviour, as tasks are not migrated too frequently between processor cores. Task migration has to be done explicitly as opposed to fetching tasks from a global task-pool, where a task could be executed on a different processor core each time, and hence has to be loaded into the core's cache each time. From the scheduling perspective, explicit task migration allows mid- and long-term planning for long-lived applications that might have to satisfy (soft) real-time constraints. On the other hand, a diligent assignment of tasks to workers and load-balancing strategies are essential to compensate for inevitable load-imbalance.

Figure 4.2 depicts the organisation of workers. Each worker runs on a kernel-level thread pinned to a core. Tasks are illustrated as circles labelled T. Execution of tasks is based on ready-sets, a worker fetches a task from the ready set and dispatches it. Communication between and with workers is accomplished by sending messages to their mailboxes. These concepts are explained in more detail in the next sections.

4.2.1 Execution of tasks

After initialisation, each worker operates in a loop that performs following steps:

1. Pick a task eligible for execution (a task that is *ready*)
2. Switch execution context to the context of the task (the task is *dispatched* by the worker)
3. Execution is passed back to the worker (context switch)
4. Output collected profiling (=monitoring) information for that task
5. Fetch and process messages from the mailbox
6. Go to step 1.

Workers execute tasks in a co-operative manner, i.e., a task is not interrupted by its (or another) worker but has to give up execution by itself. It can do so by either finishing its computation, or by being suspended during calling a *blocking subroutine*. The latter puts the task in a *blocked* state as it waits on an event triggered by another task, and gives execution back to the worker.

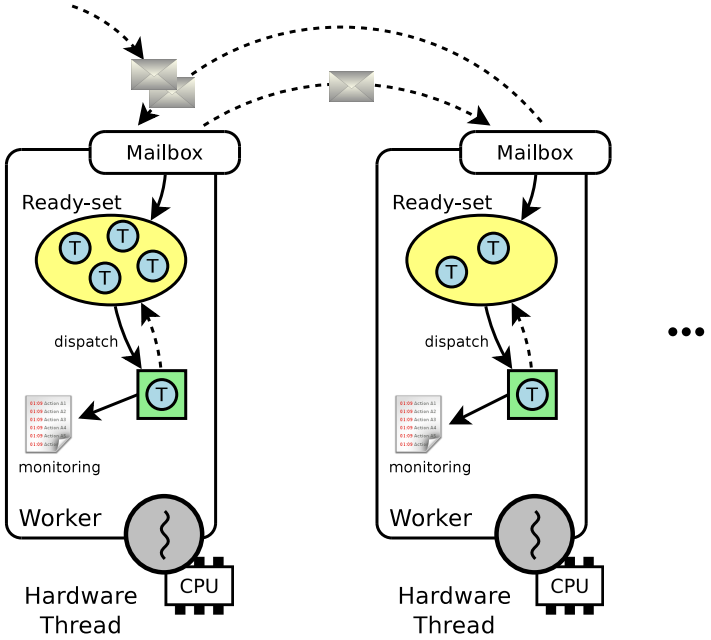


Figure 4.2: Each worker runs as a pinned kernel-level thread on a separate core and manages its own set of (user-space) tasks. Mailboxes facilitate communication among workers.

To be more precise, as in the stream processing network the only means how tasks can communicate and synchronise upon each other are streams, the only blocking functions are certain stream operations. For example, if a task calls `StreamRead()` to read an item from a stream, the task is suspended if the stream is empty. Streams are described in detail in Section 4.4.

If in step 1 of the worker loop no tasks are ready for execution, it either could instantly poll the state of its tasks, or it could suspend itself and rely on an external notification if a task becomes ready. The latter is clearly the better choice if one wants to avoid wasting CPU-cycles.

4.2.2 Communication over Mailboxes

To allow notification of workers, either from outside of a worker or among workers, each worker is equipped with a mailbox, and communication takes place by message passing. As the workers access the mailboxes of each other concurrently, with a possibly high contention, special efforts are required to keep the synchronisation overhead low at this point. Therefore, *concurrent data structures* (see Section 2.4) are used.

The mailbox of a worker basically consists of a message queue, in which messages are enqueued by other workers and dequeued *only* by the owning worker. The number of messages a worker can receive is not bounded a-priori, therefore fixed-size arrays are not desirable. Hence, a variation of the two-lock queue algorithm of Michael and Scott [MS96] is used.

In the original two-lock algorithm, the queue is implemented as a singly

linked-list that improves the naive single-lock approach by having separate locks for the head and tail pointers. This allows an `enqueue` operation to execute in parallel with a `dequeue` operation, and is achieved by always keeping a "dummy" node in the queue, at the head (cf. Section 2.4.1). Enqueuers have to synchronise upon the tail-lock, dequeuers upon the head-lock.

In our case, only the queue-owning worker dequeues messages from the queue, and this multiple-producer single-consumer (MPSC) scenario effectively makes the head-lock superfluous. Hence, for the `dequeue` operation, no atomic operation is involved at all, for the `enqueue` operation, two pointer assignments have to be protected by a lock.

Note that because a dummy node is always kept at the head of the queue, the `dequeue` operation requires the message to be copied, as the node holding the received message becomes the new dummy node. The previous dummy node can be de-allocated.

As mentioned in the previous subsection, a worker thread suspends its execution if it has no ready tasks to execute. To facilitate this behaviour, the mailbox M is equipped with a semaphore $M.sem$. Before dequeueing a message node in the RECEIVE operation, a P($M.sem$) (wait) is issued, and after enqueueing a message node in the SEND operation, a V($M.sem$) (signal) is issued. As a result, if the mailbox contains no messages, a receive operation will lead to the worker thread to be suspended upon a receive operation. In addition to the send and receive operations, an operation HAS-INCOMING is provided. This enables the worker to check if there are incoming messages. This is required, because in step 5 of the worker loop a worker fetches all incoming messages, if there are any, before starting its next loop iteration. Only if there are no ready tasks in step 1, the worker calls RECEIVE on its mailbox, which potentially suspends its execution until a new message arrives.

Algorithm 2 displays the pseudo-code for the mailbox operations. Each message node n has a pointer to the next node $n.next$, and a message field $n.msg$. A mailbox M has, besides the semaphore $M.sem$, a pointer to the head $M.head$ and the tail $M.tail$ of the queue, and a tail lock $M.tail\text{-}lock$.

Free-pool of message nodes

To avoid permanent reallocation of memory for list-nodes (=messages), each mailbox employs a pool of free messages. The procedure for sending and receiving a message is as follows. Assume, a message is sent from worker A to worker B. The whole operation involves following steps:

1. A obtains a message node m from the free-pool of B. If the free pool is empty, allocate a new node for m.
2. A writes the contents of the message to m.
3. A performs an `enqueue` operation on B's message queue.
4. B eventually obtains m by a `dequeue` operation on its message queue.
5. B reads the contents of m for further usage.
6. B puts m back to its own free-pool.

Algorithm 2 SEND, RECEIVE, and HAS-INCOMING operations of the worker mailbox.

SEND(M, msg)

Place a message msg into mailbox M.

```
1  node *n = alloc-node()  // Allocate a new message node
2  n.msg = msg             // Set message field

3  lock(M.tail-lock)       // Aquire tail lock
       // Critical Section
4      M.tail.next = n     // Link node n at the end of the list
5      M.tail = n          // Swing tail to node n
6  unlock(M.tail-lock)     // Release tail lock

7  V(M.sem)                // Signal the semaphore
```

RECEIVE(M)

Fetch a message from the mailbox M.
The worker will be blocked if there is no message available.

```
1  P(M.sem)                    // Wait on the semaphore

2  node *n = M.head            // Read head, points to dummy node
3  message msg = n.next.msg    // n.next exists,
                               //    message is copied
4  M.head = n.next             // Swing head to next node,
                               //    it becomes the new dummy node
5  free-node(n)                // Free the old dummy node

6  return msg
```

HAS-INCOMING(M)

Test, if the mailbox M has incoming messages.

```
1  if M.head.next ≠ NULL
2      return TRUE
3  else
4      return FALSE
```

Algorithm 3 PUSH and POP operations of the lock-free stack of free message nodes.

PUSH(S, n)

Push node n onto stack S. $S.top$ points to the top node of the stack, or is NULL if the stack is empty.

```
1  node *tmp
2  repeat
3       tmp = S.top
4       n.next = tmp
5  until CAS(&S.top, tmp, n)
```

POP(S)

Pop a node from stack S. $S.out\text{-}cnt$ is the out counter to avoid the ABA problem. Returns NULL if the stack is empty.

```
1  node *tmp, int oc
2  repeat
3       tmp = S.top
4       oc = S.out-cnt
5       if tmp == NULL
6            return NULL
7  until CAS2(  &S.top, tmp, tmp.next,
                &S.out-cnt, oc, oc + 1   )
8  tmp.next = NULL
9  return tmp
```

Note that this way message nodes do not change their mailbox. If a worker picked a free node from its own free-pool, message nodes would migrate from one worker to another, causing two potential problems. On one hand, if message flow is unbalanced, i.e., worker A sends messages to worker B more frequently than vice-versa, A's free-pool will permanently be empty, causing allocation of new message nodes. On the other hand, as a number of message nodes is pre-allocated in the initialisation phase, the nodes are located in proximate memory locations. This will presumably lead to better performance, when a worker fetches all incoming messages from its mailbox.

This results in the free-pool of being accessed in a multi-consumer single-producer (MCSP) way: Only the receiver ever puts the node of the consumed message back to its own free-pool.

The free-pool is implemented as lock-free stack, like the one first described by Treiber [Tre86]. It requires a single Compare-and-Swap (CAS) instruction for the `push` operation and a single Double-Compare-and-Swap (CAS2) instruction, as described on page 20 in Section 2.4.2, for the `pop` operation. CAS2 is required, as the lock-free stack is subject to the ABA problem. An *out counter* is updated along with the pointer upon each successful CAS2 operation, indicating the number of successful pop operations on the stack. Algorithm 3 lists

the stack operations of the lock-free stack S. The fields $S.top$ and $S.out\text{-}cnt$ are accessed concurrently and subject to change during the stack operations.

4.2.3 Scheduling

One of the major design decisions was to decouple the order in which tasks become eligible for execution on a worker from the actual order in which they are dispatched by that worker. In this context, the term *scheduling* means choosing a task from the ready set to be dispatched next (step 1 of the worker loop).

Decoupling the order of execution of tasks from the order induced by their state transitions to READY makes it possible to perform priority-driven scheduling. For example, shorter tasks could be preferred over computational-intensive tasks. Or, taking the topology of the stream-processing network into consideration, tasks of a sub-network containing a critical path of an application could be preferred over tasks not located on that path.

If a stream-operation within a task t causes another task u to become ready, the owning worker of t acts dependent on the location of u. Let T be the owning worker of task t and U be the owning worker of task u. Then, following case distinction is made:

- If $T = U$, t is put immediately into the ready set of T.
- Otherwise, T sends a task-wake-up message containing u to U. U eventually receives this message and puts u in its own ready set.

So, waking up tasks located on other workers is handled by notification over mailboxes. This way, the set of ready tasks is only accessed by the owning worker of these tasks, and due to the lack of concurrent access on the ready sets, mutual exclusion mechanisms and or concurrent data structures are not required for their implementation.

For the scheduling module, the implementation is hidden mainly behind two functions:

PUT-READY(T, t)
: Worker T puts a task t into its ready set.

$t =$ FETCH-READY(T)
: Worker T fetches a task from its ready set, storing the result in t.

Although a worker may only access its own ready set, it is possible for each worker to have its own, application-specific scheduling policy.

4.3 Tasks

In this section we first describe the characteristics of an LPEL task, imposed by the underlying computational model. Afterwards, we describe how they are actually designed in the LPEL.

4.3.1 Stateful vs. stateless

As outlined in Section 2.1.2, for Synchronous Dataflow networks (SDF) the consumption and production rates of each process are known a-priori. Hence, also the demands of input data as well as the resource demands of the outbound communication channels are known a-priori, which makes static scheduling possible, such that processes can successfully complete their computation upon each firing. In the more general model of Kahn Process Networks (KPN), the consumption/production rates are not known a-priori. Due to the use of bounded communication channels, it is possible that a process cannot complete its computation as a write to a full buffer at an outgoing channel fails. This requires the current state of the process to be saved, such that it can be resumed if the outgoing channel has available space again.

As the LPEL is designed with the requirements of the S-Net runtime system in mind, we must consider how S-Net boxes are handled in the runtime system and what that means in the perspective of a S-Net user, i.e., box implementor.

Most important of all is the fact that S-Net boxes are *stateless*. They are not allowed to have any permanent internal state, which allows for constructs like serial replication. From an S-Net users perspective, a box is executed (fired) each time a record is available on its input stream (remember, boxes are also SISO). The user has no notion of the streams between the boxes to be bounded, as a consequence the suspending of a box is completely transparent to the user.

Another relevant aspect of the S-Net model is that the box is not restricted on how many records it is allowed to emit upon each firing. The box could act as an emitter, e.g., triggered by a single input record, it could instantly produce output records in a possibly endless loop. Operationally, the box is a stream transformer function T. It maps each single record from its input stream to a possibly empty stream of records on the output stream:

$$T : a \mapsto b_1, b_2, b_3, \ldots, b_n$$

where n is an input-data dependent function $n(a)$.

The S-Net runtime system implements box handling as a simple function call to the user-written box function, performed within a box-entity. The internals of the box function are not accessible from within the box-entity, in fact it can be pre-compiled, only a reference to the box function must be supplied to the box-entity. The box-entity executes a loop in which it reads records from the input stream, and calls the box function each time the input record contains data for the box (there also exist records that are only used within the runtime system and therefore are invisible to the user, e.g., which are responsible for handling of deterministic S-Net combinators, synchronisation in synchro-cells and proper termination, cf. Section 3.2). The box-entity passes an additional parameter to the box function, constituting of a (from the user's perspective) opaque handle. Within the box, this handle is passed to the `SNetOut()` function, which is provided by the runtime system and used for writing records to the outgoing stream. It is not needed to be specified where to write the record, as each box has only a single output stream. The handle contains, besides S-Net specific type information of the box, a pointer to the outgoing stream. The `SNetOut()` function uses this information to properly write the record to

the outgoing stream. As this write operation could encounter a full buffer, the whole box-entity must be suspended.

It is important to emphasise the fact that while the box function itself *must not* contain any state, the box-entity *does require* a state in the form of an execution stack, due to the possibility of being suspended during the execution of its box function.

Currently, all the entities of the multi-threaded S-Net runtime system are mapped to a separate (kernel-space) thread, but we want to map these entities to (user-space) LPEL tasks. Of course, one could argue that the necessary state saving upon a stream operation encountering a full or empty buffer within runtime system entities could be performed by the runtime-system explicitly, removing the need of an execution stack. Dispatching an entity would result in a plain function call, passing the previously stored state as additional parameter. In fact, apart from the box-entity, every other entity of the S-Net runtime system could be rewritten to express this behaviour. But for the box entity, the execution stack of the box must be saved in any case.

One could make the distinction of tasks with and without a separate execution stack, but this would require a far more difficult task wake-up handling, and for the sake of simplicity, we decided to save the execution stack for every LPEL task.

To summarise, LPEL tasks necessarily do have a state, in the form of an execution stack, hence comprising a user-space thread.

4.3.2 Task Control Block (TCB)

Like operating-system processes, which are managed with a data structure called process control block (PCB), also LPEL tasks are managed with a similar data structure, the task control block (TCB). The TCB of an LPEL task contains the following information:

TID Newly created tasks get assigned a unique task identifier (TID) in ascending order.

STATE State of the task. Can be one of CREATED, RUNNING, READY, BLOCKED or ZOMBIE.

PTR Pointers for linking tasks together in a list.

WCTX Reference to the worker context of the worker the task is currently assigned to.

SYNC Data needed for synchronisation of the task with other tasks, currently only needed for polling stream sets (cf. 4.4.5).

SCHED Task-specific scheduler data, whose actual contents are dependent on the scheduling module.

FLAGS Flags specifying which information to collect for accounting during execution.

ACCNT Accounting information: start/stop timestamps of last dispatch, a dispatch counter, and a list of “dirty” streams, i.e., streams on which a stream operation was performed.

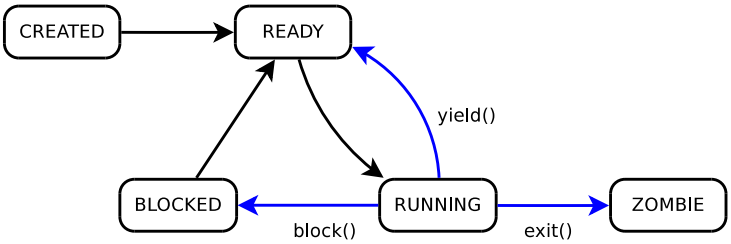


Figure 4.3: Life-cycle of an LPEL task.

STACK The hardware context and execution stack.

4.3.3 Life-cycle of a Task

The life-cycle of a task is depicted in Figure 4.3. Initially, a task request message is sent to a specified worker, containing the task function, stack size of the execution stack and flags. Upon receiving a task request, the worker creates the task control block, assigning a unique task identifier (TID), setting the STATE to CREATED, storing a reference to its worker context (WCTX), and initialising the other remaining structure. After this initialisation phase, its state is set to READY and the task is put into the ready set of the worker.

A READY task is eventually dispatched by the worker, putting it into the RUNNING state. As mentioned before, tasks are dispatched cooperatively, i.e., they are not preempted in their execution, but have to give up execution voluntarily. There are three possibilities for a function to call from within a task to give up execution:

exit()
: The task has finished its computation. State changes to ZOMBIE.

yield()
: The task interrupts its execution for no particular reason, just to avoid monopolising the worker. State changes to READY.

block()
: The task has to wait for an event before it can resume its computation. This function usually is called from within a blocking stream operation. State changes to BLOCKED.

The transitions from and to the RUNNING state involve a task context switch. Upon dispatching a task the context is switched from the worker to the task, and the task resumes its execution after the last instruction of its previous dispatch. If a task gives up execution, the context of the worker is restored to the point where it has dispatched the task. Then the worker decides what to do with the returned task, depending on its state upon return. If the task's state is ZOMBIE, the TCB of the task is destroyed. But instead of freeing the memory, the TCB structure is stored in a local list such that it can be reused upon arrival of a new task request. If the state is READY, the task is put in the worker's ready set immediately. Nothing is done if the state is BLOCKED: events that cause a task to become ready again will place the task

Function call	Description	State changes to	Reaction of worker
`exit()`	The task has finished its computation	ZOMBIE	Destroy TCB
`yield()`	The task interrupts to avoid monopolising the worker	READY	Put into ready-set
`block()`	The task has to wait for an event	BLOCKED	Do nothing

Table 4.1: A task gives back execution to the worker by calling a blocking function. The worker reacts depending on the state of the returning task.

into the ready set or send a task-wakeup message, dependent on the location of the task (see Section 4.2.3).

The transitions back from the RUNNING state are summarised in Table 4.1.

4.3.4 Monitoring of Tasks

For monitoring purposes, the TCB contains an ACCNT structure.

Upon each dispatch of a task, a dispatch counter d is incremented and stored in the ACCNT structure. Also, a timestamp $t_{\text{start},d}$ is captured before the switch to the task's context, and a timestamp $t_{\text{stop},d}$ after the switch back to the worker's context.

Additionally, during the execution of the task, information about activity on streams is collected, which is described in detail in Section 4.4.4. The worker outputs (if desired) the monitored information for that task after each dispatch d.

From the execution times at each dispatch d, already some basic measures can be obtained. The difference $T_d = t_{\text{stop},d} - t_{\text{start},d}$ gives the execution time of that dispatch d. From this information the total execution time is simply derived as

$$T_{\text{total}} = \sum_{d=1}^{d_{\max}} T_d,$$

and the average execution time of a task is derived as

$$T_{\text{avg}} = \frac{T_{\text{total}}}{d_{\max}}.$$

The ACCNT structure also contains a timestamp of the task creation, t_{creat}, which enables to calculate the task's alive time

$$T_{\text{alive}} = t_{\text{stop},d_{\max}} - t_{\text{creat}},$$

and based on this the idle time

$$T_{\text{idle}} = T_{\text{alive}} - T_{\text{total}}.$$

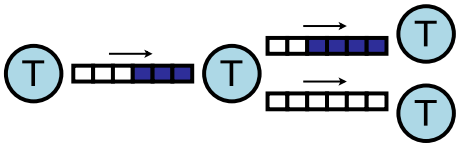


Figure 4.4: Tasks communicate over uni-directional single-producer single-consumer streams, implemented as bounded buffers.

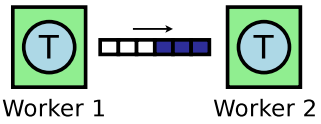


Figure 4.5: Tasks communicating over a stream, being dispatched in parallel.

4.4 Streams

One of the most important components of the LPEL are the uni-directional communication channels between the tasks, the realisations of streams. They provide the only means of communication between tasks, and stream operations encapsulate the necessary synchronisation mechanisms to block tasks and make them ready again. Thus, as a stream basically is designed as bounded first-in first-out (FIFO) buffer (Figure 4.4), a task t trying to write to a full stream or to read from an empty stream needs to be blocked. A task u reading from the full stream or writing to the empty stream will put the previously unsuccessful task t in ready state again (i.e., u wakes t up).

It must be pointed out that the problem of synchronising tasks upon the stream operations lies in user-space, i.e., tasks are suspended, whereas the worker continues running its main loop and will pick another ready task for execution. Still, two tasks communicating over a stream can execute the write and read operations truly in parallel, by being located on different workers on different cores and being dispatched at the same time, as shown in Figure 4.5. This must be kept in mind for the design and verification of the synchronisation mechanisms devised.

As explained in Section 2.2, the S-Net model allows for non-deterministic behaviour at merge points, which subsequently requires the runtime system to be able to test the availability of data in streams at collector-entities. This is done by providing a Peek operation besides the Read and Write operations on streams. As we will see, also a Poll operation is required, to allow a task to wake up if any of a given set of streams has data available that can be read.

In the next subsections we gradually refine the synchronisation mechanisms provided by streams, starting with the currently employed solution in the multi-threaded S-Net runtime system, which is suitable for multiple producers and multiple consumers.

Then we will consider the underlying buffer for the stream data and see that we can access the items without using mutual exclusion given there exists only a single consumer and a single producer. Afterwards, we derive the mechanism to suspend and re-activate tasks blocked on streams, again exploiting the fact that we only have a single producer and a single consumer.

Algorithm 4 Monitor-based solution for the producer-consumer problem.

Initially, $B.count = B.nextin = B.nextout = 0$

WRITE(B, x)

Write an item x into the buffer of monitor B.

```
1  lock(B.door)          // Enter monitor B
2      if B.count == SIZE          // Buffer is full, avoid overflow
3          cond-wait(B.notfull, B.door)
4      B.buf [B.nextin] = x        // Place the item in the buffer
5      B.nextin = B.nextin + 1  mod SIZE
6      B.count = B.count + 1
7      cond-signal(B.notempty)     // Resume any waiting consumer
8  unlock(B.door)      // Exit monitor B
```

READ(B)

Read an item x from the buffer of monitor B.

```
1  lock(B.door)          // Enter monitor B
2      if B.count == 0             // Buffer is empty, avoid underflow
3          cond-wait(B.notempty, B.door)
4      item x = B.buf [B.nextout]) // Retrieve the item from the buffer
5      B.nextout = B.nextout + 1  mod SIZE
6      B.count = B.count - 1
7      cond-signal(B.notfull)      // Resume any waiting producer
8  unlock(B.door)      // Exit monitor B
```

4.4.1 Monitor-based solution

The (bounded buffer) producer-consumer problem is a well-known and studied multi-process synchronisation problem. In the multi-threaded S-NET runtime system (cf. Section 3.2), the problem of synchronisation between the (kernel-space) threads is solved with the concept of *monitors*, i.e., by the usage of condition variables and mutexes, which the PThread API provides.

Algorithm 4 depicts the monitor-based solution for the producer-consumer problem. Entering and leaving the buffer-monitor B is explicitly done by acquiring and releasing a lock $B.door$ in the stream operations READ and WRITE. The condition variables $B.notempty$ and $B.notfull$ are used for suspending and signalling producers and consumers. Upon waiting on a condition, the lock $B.door$ is released, and re-acquired automatically when the thread is resumed. The counter $B.count$ keeps track of the number of elements contained in the actual buffer $B.buf$, which has a capacity of SIZE items. The counters $B.nextin$ and $B.nextout$ point to the position in the buffer $B.buf$ where the nested item is stored to or loaded from, and they are incremented each time modulo the buffer size.

The solution works for multiple producers and multiple consumers. But,

Algorithm 5 Lamport's lock-free circular buffer implementation.

ENQUEUE(*item*)

Enqueues *item* in the buffer and returns TRUE, if there is space left in the buffer. Otherwise, FALSE is returned.

```
1  if head + 1  mod SIZE == tail
2       return FALSE

3  buffer [head] = item
4  head = head + 1  mod SIZE
5  return TRUE
```

DEQUEUE(*item*)

If the buffer contains items, the procedure dequeues one, stores it in *item* and returns TRUE. Otherwise, FALSE is returned.

```
1  if head == tail
2       return FALSE

3  item = buffer [tail]
4  tail = tail + 1  mod SIZE
5  return TRUE
```

as a stream conceptually is uni-directional and connects exactly two tasks, a specific solution for the single producer and single consumer case is sufficient.

In order to develop refined synchronisation mechanisms, a close look on the underlying data structure, the circular bounded buffer, is taken.

4.4.2 Single-Producer Single-Consumer FIFOs

In the situation of a single producer and a single consumer, Algorithm 4, imposes following restriction: Even when the consumer is reading an earlier enqueued element, the producer cannot enqueue an element into a different buffer slot. Due to the monitor-lock, only one of them could access the buffer at the same time.

Lamport's Concurrent Lock-free Queue

Lamport [Lam83] proved that, under *sequential consistency*, a circular buffer can be implemented without locks in the single-producer single-consumer case, as depicted in Algorithm 5. Items are enqueued at the head and dequeued at the tail. To distinguish the buffer empty from the buffer full case, one buffer slot always remains free. Note that the resulting queue is even *wait-free*: The producer/consumer cannot prohibit the other from progress.

Sequential consistency requires, that a consumer sees the operations for writing to the buffer and updating the head in the same order as executed by

Algorithm 6 The FastForward circular buffer implementation.

```
ENQUEUE(item)

1  if buffer[head] ≠ ⊥
2      return FALSE

3  // Write-Memory-Barrier here, on weak consistency models,
   //    if references to external data are transferred
4  buffer[head] = item
5  head = head + 1  mod SIZE
6  return TRUE

DEQUEUE(item)

1  item = buffer[tail]
2  if item == ⊥
3      return FALSE

4  buffer[tail] = ⊥
5  tail = tail + 1  mod SIZE
6  return TRUE
```

the producer. Otherwise, it could happen that a consumer reads stale and thus invalid data from a buffer location. The dual case is similar, possibly resulting in an overwritten, yet not consumed item.

The next subsection describes a buffer implementation that avoids this problem. We will use that data structure for the buffers in the LPEL streams.

FastForward Queue

While there is no explicit synchronisation between the producer and the consumer at the algorithmic level, there still exists an implicit synchronisation between them at the memory layer as the control data (i.e., head and tail) is still shared. Thus, on modern cache-coherent systems, the queue still results in cache-line thrashing across caches. Furthermore, to support weaker memory consistency models, potentially expensive memory barriers are required to ensure correct ordering between the data writes (buffer) and the control writes (head/tail).

The FastForward queue [GMV08] eliminates this separation of control and data by tightly coupling control and data into a single operation. This is done by employing a known value $\perp$ for an empty buffer slot, in order to use the buffer itself to indicate full and empty conditions. Thus, the head and tail are not shared and can remain cache resident (provided that they are located in separate cache-lines). Also, all buffer slots can be occupied by items. Algorithm 6 depicts the FastForward implementation.

Note that the FastForward queue assumes that every item can be trans-

ferred in a single write operation. This is no limitation, as for large items a pointer to an external memory location can be transferred. In the S-Net runtime system, only references to the records are transmitted through the buffer anyway. In this case, as $\perp$ element, the NULL-pointer suffices. But passing references requires a write-memory-barrier for architectures with a weak memory consistency model, where stores to memory are not necessarily seen in program order at remote processors. Without a barrier, it could happen that the write of the reference to the data to the queue buffer is visible to the consumer before the update of the actual data, effectively causing the consumer to read stale data. Memory barriers are an unavoidable cost for weak consistency models and must be paid with any communication mechanism, whether based on lock-free or locking data structures.

For the LPEL, the choice of FastForward as the underlying implementation for stream buffers also makes the implementation of the operation for testing the availability of new data at the consumer side very convenient: Simply return $buffer\,[tail]$. If the buffer is empty, NULL is returned, which might be expected for references as items.

4.4.3 Writing and Reading to and from a Stream

Having presented a buffer data structure that allows for concurrent access of a single consumer and a single producer without explicit synchronisation, the problem of putting it into a context where tasks are blocked and waken up upon the transitions to and from empty/full buffers still remains open.

For the producer-consumer problem, a solution based on semaphores exists that is simple and perfectly fits the problem. Algorithm 7 lists this solution, already in the context of a stream S, by assuming a single producer and a single consumer, and that enqueue and dequeue are operations of a FastForward queue. Semaphore $S.e$ keeps track of the number of empty slots, and semaphore $S.n$ keeps track of the number of occupied slots.

As in the LPEL stream operations are called from within a task and tasks are going to be managed by workers, the semaphores used in this algorithm differ from those used in Algorithm 2 on page 33: They do not suspend the worker (kernel-space) thread, but only block the (user-space) task. Hence, the semaphore primitives provided by the operating system cannot be used, but own primitives have to be developed.

Semaphores usually consist of a counter variable and a queue. A P() (i.e., wait) operation decrements the counter, and, if it falls below zero, blocks the calling process and puts it on the semaphore's queue. A V() (i.e., signal) operation increments the counter, and if it still is less than or equal to zero, then a process is removed from the queue and unblocked. The operations of a semaphore must be atomic and protected from concurrent access. In a multi-threaded environment this is achieved by employing *spinlocks*.

Assigned consumer and producer. We want to further exploit the fact that there is only a single producer and a single consumer assigned to a stream S, such that there are exactly two tasks that operate on s. Assume task p is the producer and task c the consumer of S, and S contains a reference on each of them, $S.prod$ and $S.cons$. Then, only p will ever block on $S.e$ and only c will ever block on $S.n$. As a result, the queue of the semaphore becomes superfluous

Algorithm 7 Semaphore-based solution for the (bounded buffer) producer-consumer problem.

Initially:
semaphore $S.e = \text{SIZE}$
semaphore $S.n = 0$

WRITE(S, x)

Write an item x to the stream S.

```
1  P(S.e)              // Wait if number of empty slots ≤ 0
2  enqueue(S.buf, x)
3  V(S.n)              // Signal occupied slot
```

READ(B)

Read an item x from the stream S.

```
1  P(S.n)              // Wait if number of occupied slots ≤ 0
2  dequeue(S.buf, x)
3  V(S.e)              // Signal empty slot
4  return x
```

and the semaphore operations can be reduced to atomic fetch-and-increment and fetch-and-decrement operations of the counter variable. Algorithm 8 shows this realisation of the READ and WRITE operations on a stream. The wake-up function will, dependent on the location of the task in the argument, either put that task in the ready set of the currently executing worker directly or notify its owning worker, as explained in Section 4.2.3.

Wake-up before block. Reducing the semaphore operations to atomic counter manipulation can lead to following interesting situation. Assume a producer p encounters a full buffer by fetching and decrementing $S.e$ in line 1 of the WRITE operation, reading 0. Before p reaches line 2 where it blocks, i.e., it returns execution back to the scheduler, a consumer c fetches and increments $S.e$ in line 4 of the READ operation on the same stream S, reading -1. A value of -1 indicates that the producer has to be unblocked (wake-up), hence the c executes line 5 of READ, even before p blocked on line 2 of WRITE. This situation would be a problem, if p is executed on a worker before it reached line 2, but this is avoided, due to the fact that tasks are not preempted. Referring to Section 4.2.3, if p and c are located on the same worker, p is put into the ready set. The worker does not fetch p from the ready set until p blocks and returns execution to the worker. If p and c are located on different workers, the owner of p is notified by a message in its mailbox. As fetching messages happens between the execution of tasks, the worker eventually receives the message only after p has blocked.

Although the use of atomic variables might not exploit all the good cache-

Algorithm 8 Realization of the stream operations with atomic fetch-and-increment and fetch-and-decrement operations.

Initially:
    semaphore $S.e = \text{SIZE}$
    semaphore $S.n = 0$

WRITE(S, x)
Write an item x to the stream S.

```
1  if fetch-and-dec(S.e) == 0
       // block currently executing task, i.e., the producer
2      block()

   // Enqueue item x in the buffer
3  enqueue(S.buf, x)

4  if fetch-and-inc(S.n) == −1
       // consumer is blocked: wake up
5      wake-up(S.cons)
```

READ(B)
Read an item x from the stream S.

```
1  if fetch-and-dec(S.n) == 0
       // block currently executing task, i.e., the consumer
2      block()

   // Dequeue an item from the buffer and store in x
3  dequeue(S.buf, x)

4  if fetch-and-inc(S.e) == −1
       // producer is blocked: wake up
5      wake-up(S.prod)
6  return x
```

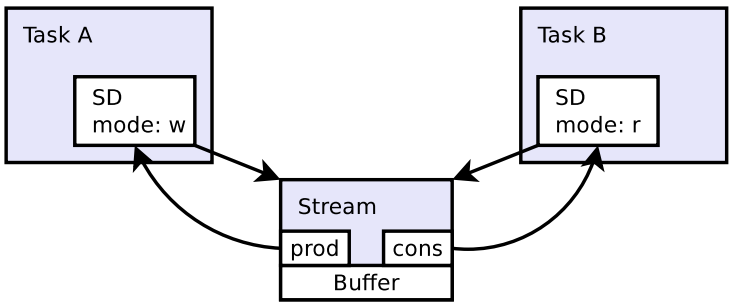


Figure 4.6: Two tasks connected by a stream.

preserving properties of the FastForward queue (atomic variables have to be shared among threads), it still presents a wait-free solution.

Besides WRITE and READ, an operation PEEK on a stream is provided for the consumer task. It does neither block the task, nor consume an item from the stream. It allows to check for available data by returning the next item in the stream without removing it from the stream, and returns NULL if there is no item available.

4.4.4 Stream Organisation and Monitoring of Stream Activity

In order to be able to use a stream, a task has to *open* it beforehand. A stream can be opened either for read or for write access, and the OPEN operation returns a stream descriptor which needs to be used for successive operations accessing the stream. After the task has finished using the stream, it has to *close* it with the stream descriptor, which becomes invalid then.

Figure 4.6 depicts two tasks A and B connected by a stream. A stream descriptor is private to each task, i.e., tasks communicating over a stream have distinct stream descriptors for accessing the stream. It contains, besides a reference to the accessed stream and the access mode, accounting information for the task's usage of that stream. This information includes the number of items a task has read/written from/to that stream, its state (OPENED, INUSE, CLOSED or REPLACED), and flags for indicating if the task has been waken up by the task on the other end of that stream or if it is blocked on it.

The accounting information is collected for a single dispatch of a task, and output by the worker after the dispatch. For this purpose, each stream descriptor *sd* contains a *sd . dirty* field. If this field is NULL then there was no activity on the stream on behalf of the task, otherwise the *sd . dirty* field is used to chain together all stream descriptors that were used for stream operations during the dispatch in a list. A pointer in the ACCNT field of the TCB points to the head of the "dirty-list", which is initially a constant representing the end of the list, e.g. $\perp$.

Let *dirty-head* denote the head of the dirty-list. Upon a stream operation, the dirty field *sd . dirty* of a stream descriptor *sd* is examined:

- If $sd\,.\,dirty \neq$ NULL, *sd* is already on the dirty list and nothing needs to be done.

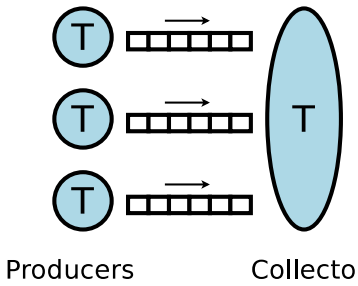


Figure 4.7: A collector polls a set of streams. If all streams are empty, the collector blocks. If an item is written to any of the streams by a producer, the collector is woken up again.

- Otherwise, the value of *dirty-head* is stored in $sd\,.\,dirty$ and *dirty-head* is set to sd.

Upon return from a dispatch of a task, its dirty list is traversed. After the monitored information of each dirty stream descriptor sd has been outputted, its state is updated and the flags are cleared. The $sd\,.\,dirty$ field is reset to NULL, and finally the head pointer in ACCNT of the task's TCB is reset to $\perp$.

Stream Sets. A stream descriptor sd also contains a $sd\,.\,next$ field, for the organisation of stream descriptors in disjoint sets within the task code. Therefore, functions for adding and removing a stream descriptor to and from a stream set, as well as for iterating through the members of a stream set, are provided. This makes it, for example, easy to broadcast data to all streams in a streams set by using the iterator. But, apart from convenience aspects for the task programmer, stream sets are needed to implement the POLL operation.

4.4.5 Polling a set of streams

The collector-entities of the S-NET runtime system implementation, which comprise the merge points of an S-NET, require the functionality of testing input streams for new data. A collector has to observe a set of streams for the availability of items on any of them, and if a new item arrives upon an empty stream, it has to react accordingly (Figure 4.7). In the following discussion, we refer to the collector task also as consumer, as for each of the streams' perspective it is on the consuming side.

Instantly iterating through the set of streams, performing a PEEK operation on each stream is no viable solution, as it results in busy-waiting and even deadlock. A task also could yield its execution after each iteration, but, as of remaining in ready state, would be dispatched by the worker regardless of the availability of new data. The result would be computational and organisational overhead: the task unsuccessfully iterates over the stream set, while the worker keeps busy dispatching the collector task despite of its lack of progress.

Therefore, an operation is required which puts the polling task in blocked state if every stream of the set is empty, such that it is unblocked again by the first producer of a stream contained in the set writing a new item to the stream. With respect to this behaviour, the term "polling" might be misleading. But

from the task's perspective, the term fits well as the POLL operation accepts a set of stream descriptors as parameter and returns a stream descriptor from this set which refers to the stream with items available, without any notion of the underlying synchronisation mechanisms.

The POLL operation is implemented by employing three additional shared variables:

Poll-Flag
A shared flag is located each stream S shared between a producer and the polling consumer *cons*, which is referred to as $S.\mathit{is\text{-}poll}$. As the name suggests, it is used to indicate the producer that the stream is currently polled, and that it might have to wake up the consumer.

Poll-Token
The second shared variable is a flag located in the SYNC field of the consumer's TCB, shared by the consumer and all producers, which is referred to as $\mathit{cons}.\mathit{poll\text{-}token}$ in the following explanations.

Wakeup-SD
The third shared variable is also located in the SYNC field, used for indicating the consumer which stream descriptor points to the stream with available items. It will be referred to as $\mathit{cons}.\mathit{wakeup\text{-}sd}$.

The idea of the POLL operation is as follows. If the consumer is blocked, multiple producers have to compete for unblocking the consumer. In the TCB of the consumer, there exists a single poll-token. Only the first producer which can grab the poll-token will unblock the consumer.

Algorithm. Upon a poll operation, the consumer places the single poll-token in its TCB. Then it iterates through all of the streams in the stream-set to check whether there is data available or not. In each iteration (a single stream of the set is considered), following case distinction is made:

- If there is no data available in the stream, the consumer sets the *is-poll* flag in the stream which causes the producer to try to grab the poll-token the next time it writes data to the stream, and the consumer proceeds to the next stream. We say, the producer is *flagged*.
- If there is data available, the consumer tries to grab the poll-token itself.
  - If it succeeds, then no flagged producer has unblocked the consumer yet in the meantime, and the consumer does not need to block at all. It can exit iterating through the streams of the stream-set immediately and return from the POLL operation.
  - If the poll-token was already grabbed, the consumer will exit iterating through the stream set but has to block afterwards, as a producer registered in one of the previous iterations successfully grabbed the poll-token.
- The operation returns the stream descriptor pointing to the stream with newly available data, which was either set by the consumer itself or by the producer that unblocked the consumer.

Algorithm 9 lists the pseudo-code for the POLL operation. Note that for this operation, the same that was explained regarding wake-up before blocking for the read and write operations holds true: A producer can wake-up a consumer before it has blocked in the POLL operation. But this is not harmful as the worker will notice that the consumer is ready again only after the consumer has returned from execution (cf. page 45).

Adapting the WRITE operation to support polling. To enable polling, also the WRITE operation of a stream has to be modified accordingly. Therefore, each time before a producer writes an item to a stream S, it has to check if the stream is currently polled by a consumer, i.e., if the flag $S.\mathit{is\text{-}poll}$ is set (the producer is flagged). If the flag is set, the producer clears it and tries to grab the poll-token. Following invariant holds within the POLL operation: If $S.\mathit{is\text{-}poll}$ is set, then the atomic counter variable for the full slots $S.n \geq 0$. Informally, a consumer that polls a stream, cannot be blocked in a READ operation. Hence, if a producer unblocks a consumer, it either does so by reading -1 in $S.n$, or by successfully grabbing the poll-token.
Algorithm 10 shows the pseudo-code of the modified WRITE operation.

Until now, we have ignored the fact that checking if the stream is empty and setting the *is-poll* flag at the consumer side, resp. writing to the stream and checking the *is-poll* flag at the producer side, have to be performed atomically to avoid a race condition. For this purpose, a lock is associated with the stream that has to be acquired within the POLL and the WRITE operations. But as only at most two tasks compete for the lock within these operations, i.e., a producer writing to that stream and a consumer polling that stream, the overhead caused by this lock is expected to be low. Note that the more frequent WRITE and READ operations on the same stream are not serialised by that lock.

Algorithm 9 Pseudo-code for polling a set of streams.

```
POLL(S)
    Poll a set S of stream descriptors. The calling task is cons.
    Returns a stream descriptor sd ∈ S, whose stream is containing
    at least one item.

 1  local do-block = TRUE
 2  cons.poll-token = TRUE        // Place the single poll token
 3  foreach sd ∈ S
 4      stream S = sd.stream
        lock(S.lock)
 5          if PEEK(S) ≠ NULL     // Check availability of data
                // Try to grab the poll token
 6              token = atomic-swap(cons.poll-token, FALSE)
 7              if token == TRUE
                    // Got it, no need to block
 8                  do-block = FALSE
 9                  cons.wakeup-sd = sd
10              unlock(S.lock)
11              break             // Exit the loop
            else
                // Register stream as being polled
12              S.is-poll = TRUE
13      unlock(S.lock)

14  if do-block == TRUE
15      block()

16  return cons.wakeup-sd
```

Algorithm 10 Modified WRITE operation to support polling of stream sets.

WRITE(sd, x)

Write an item x to the stream pointed to by the stream descriptor sd.

```
 1  local poll-wakeup = FALSE
 2  stream S = sd.stream
 3  if fetch-and-dec(S.e) == 0
        // block currently executing task, i.e., the producer
 4      block()

 5  lock(S.lock)
        // Enqueue item x in the buffer
 6      enqueue(S.buf, x)
 7      if S.is-poll == TRUE
 8          poll-wakeup = atomic-swap(S.cons.poll-token, FALSE)
 9          S.is-poll = FALSE
10  unlock(S.lock)

11  if fetch-and-inc(S.n) == -1
        // consumer is blocked: wake up
12      wake-up(S.cons)
13  elseif poll-wakeup == TRUE
14      update(S.cons.wakeup-sd)
15      wake-up(S.cons)
```

Chapter 5

Implementation

As a proof of concept of the LPEL design, an implementation is provided. This chapter gives an overview and some details about it.

The LPEL has been implemented as a standalone library, in the (ISO-) C programming language and targeted towards POSIX operating systems. As a first prototype, the Linux operating system was targeted. The LPEL library contains about 4000 lines of documented code. Library functions are provided for initialisation, where, for example, the number of workers can be specified, for the creation and assignment of tasks, and for the use and organisation of streams within tasks.

For atomic memory operations, the LPEL facilitates the atomic built-ins of the GNU C Compiler (GCC, [GFSF11]). If GCC is not available, and for Double Compare-and-Swap (CAS2), inline assembly is provided for the Intel x86 and x86-64 instruction set architecture (ISA).

The multi-threaded S-Net runtime system has been ported to the LPEL, to provide a platform available for experiments.

5.1 Kernel-level Threads

The POSIX Threads (PThreads) API is used to create operating system threads as LPEL workers. Either by using PThreads API extensions or by the means of operating system calls, these kernel-space threads are pinned to the available cores, such that each core is assigned a kernel-space thread with a worker executing on it. These functions are:

```
int pthread_setaffinity_np( pthread_t thread, size_t cpusetsize,
                            const cpu_set_t *cpuset);
```

This PThreads extension accepts the thread handle as first argument, which is obtained upon thread creation. The `cpuset` argument is a bitmask which specifies on which cores the thread is allowed to be scheduled by the operating system. The i-th bit correspondents with the CPU with id i. For worker threads, only one bit is set in the bitmask, as a worker is allowed only to run on a single CPU.

```
int sched_setaffinity( pid_t pid, size_t cpusetsize,
                       cpu_set_t *mask);
```

This is the Linux operating system call for providing the same functionality. Instead of the thread handle, the Linux process identifier is required as first argument. Each Linux thread has such an identifier, although often referred as thread identifier (tid).

For synchronisation of kernel-level threads, mechanisms provided by the operating system through the POSIX API are facilitated. These are:

Spinlocks which cause a thread to spin in a busy-wait loop if it cannot acquire the lock. They are used to protect the short critical sections of the tail-lock in the worker mailbox, and the lock in streams used for supporting the polling operation. Currently, the spinlock provided by the PThreads Realtime Extension is employed. Alternatively, usual PThread mutexes can be used, but care must be taken as they deschedule the thread if the lock cannot be acquired, causing additional context-switching overhead.

Semaphores for worker mailboxes, used to keep track of the number of messages in the mailbox and making it possible for a worker to block waiting for new messages.

5.2 Atomic Operations

The LPEL facilitates atomic operations, i.e., operations for atomic memory access, for its synchronisation mechanisms. Since version 4.1.2, GCC provides built-in operations for atomic memory access that are compatible to Intel's ICC built-in atomic operations [GFSF11]. These are built-in functions of the compiler that are translated to atomic instructions of the target architecture during compilation. If GCC built-ins cannot be used, because, e.g., a different compiler that does not support them is used, inline assembly containing the instructions for the x86 and the x86-64 instruction set architecture is facilitated. Most instructions require a special instruction called "lock prefix" before the actual instruction, to ensure that the operation is performed atomically. Table 5.1 lists the atomic operations which are required by the LPEL, how they are used, and how they are implemented by a GCC atomic built-in function and the x86(-64) instruction set architecture.

A special case is the compare-and-swap-2 (CAS2) function, which has no GCC built-in implementation. Here the CMPXCHG8B on architectures with 32 bit pointer width and CMPXCHG16B on architectures with 64 bit pointer width is employed. Rather than operating on distinct memory locations, these instructions operate on adjacent memory locations. This requires that for the free node stack of the mailbox, the top pointer and the out counter are stored in adjacent memory locations (cf. Section 4.2.2).

For other architectures than x86(_64), a fallback solution exists that emulates atomic operations by employing a PThread mutex for each atomic variable. Of course, it is desirable to provide inline assembly for the architectures that are targeted in the future.

Atomic operation	Semantics	Usage in LPEL	GCC built-in	x86 instruction
fetch-and-increment	atomically reads an integer from memory and writes the incremented value back	generating unique, identifiers, read/write operations on streams	`__sync_fetch_and_add`	XADD (with lock prefix), argument 1
fetch-and-decrement	atomically reads an integer from memory and writes the decremented value back	read/write operations on streams	`__sync_fetch_and_sub`	XADD (with lock prefix) argument -1
atomic-swap	atomically exchanges the contents of a memory location with the contents of a register	poll/write operations on streams (stream-sets)	`__sync_lock_test-_and_set`	XCHG
compare-and-swap (CAS)	like atomic-swap, but exchanges only if value equals an expected value	push operation of the free message node stack of a worker mailbox	`__sync_bool_compare-_and_swap`	CMPXCHG (with lock prefix)
compare-and-swap-2 (CAS2)	like CAS, but compares and exchanges two memory locations atomically	pop operation of the free message node stack of a worker mailbox	N/A	CMPXCHG8B CMPXCHG16B (with lock prefix)**

** see text for explanation.

Table 5.1: Atomic operations used in LPEL and how they are implemented by GCC atomic built-ins and the x86 instruction architecture.

5.3 User-space Context Switch

Context switching in user-space is achieved with the *GNU Portable Coroutine Library (PCL)*, [Lib11]. It can be downloaded from `http://www.xmailserver.org/libpcl.html` (v1.12, accessed Jan 2011). It provides basic low-level mechanisms for context-switching and has functionality to be used in a multi-threaded (PThreads) environment. PCL is easily portable on almost every Unix system and on Windows. Mainly four functions are used from the library:

co_create for creating a new context upon creation of a task,

co_current for retrieving the context of a worker at initialisation of the worker,

co_call for performing an actual context switch, and

co_delete for destroying a context, after a task exited.

5.4 Porting S-Net to LPEL

Porting S-Net to LPEL required adding LPEL initialisation to the startup process. This includes specification of the number of workers and their creation. Each entity was modified to make use of the LPEL streams, i.e., first opening a stream and retrieving a stream descriptor before writing and reading to and from a stream.

5.4.1 Handling of Blocking System Calls

One question which arose was how to handle blocking system calls. This is especially important for the global input and global output stream of a network. As the kernel-level threads for reading and writing to these top-level streams facilitate blocking system calls, e.g., reading from stdin and writing to stdout, creating task for them and putting them on an ordinary worker could block the whole worker and prevent it from dispatching its tasks (see Section 2.3).

Our solution was not using asynchronous system calls, but putting these tasks onto separate "wrappers" that provide a proper worker context, but are not pinned to a particular core and where each wrapper only executes a single task. Wrappers either run on one or more processor cores dedicated only to wrappers, i.e., some cores are reserved for executing wrapper threads instead of workers, or are distributed among the cores that are executing worker threads. Latter option is beneficial when only a small number of cores is available and one does not want to spare cores solely for wrapper threads. In either case, creating tasks for threads containing blocking system calls is necessary because streams can only be used in the context of a task.

The same technique was used for being compatible to the Distributed S-Net extension. Distributed S-Net builds on top of MPI [For94], which is a message passing interface for communication within a cluster of workstations. The Distributed S-Net extension employs synchronous message passing, i.e., send and receive operations block until the communication partner has received the message (send) or the message has not arrived yet (receive).

In Distributed S-Net different parts of the streaming network can be located at different workstations. Streams between subnetworks are split conceptually in two endpoints such that at the producer location a kernel-level thread is employed that constantly reads records from the stream and sends it with an MPI send function. At the consumer site a kernel-level thread constantly receives records with an MPI receive function and forwards it into the sub-network located on that site. Like with global input and output streams, these threads are replaced by tasks that are put on a wrapper kernel-level thread each. Generally, each additional thread is transformed to a task that is executed on a wrapper.

5.4.2 Placement and Scheduling

Placement and scheduling are separate modules which have to be developed according to yet to be defined requirements, which is out of the scope of this thesis. In the prototype implementation, very plain and simple placement and scheduling strategies are employed.

Assignment. The assignment module performs a static assignment of tasks to workers. For S-Net, a simple abstraction is used: box-tasks (tasks which are an instance of a box-entity) are assumed to be computationally intensive, whereas all other tasks (split/merge-points, synchrocells and filters) are assumed to have almost no computational cost. Hence, boxes are distributed among workers in a round-robin fashion. All other tasks are placed onto the worker to which the next box task will be assigned to. The assignment module produces a file where it outputs the mapping from task-ids to box names or entity types. This information is useful for the interpretation of the monitoring logs.

Scheduling. The scheduling module is implemented in a similar, plain way: workers dispatch their tasks in FIFO order. A call to PutReady puts a task at the end of a single task-queue, a call to FetchReady returns a task from its front.

5.4.3 Monitoring

Monitoring can be configured regarding the amount of monitored information. In the prototype implementation, there are several monitoring levels:

a. Only output information for box-tasks, and only their execution time.

b. Only output information for box-tasks, both execution time and stream activity.

c. Output information for all tasks, both execution time and stream activity.

d. Like c, but additionally output debug information of the workers, mostly wait-count and wait-time, i.e., how often and for how long they were waiting on a new message.

For each worker, a separate monitoring log file is produced. A log file contains log records, each log record is on a separate line. A log record contains all information related to a single task dispatch. Following example shall describe the contents of a log record (the line is wrapped only in this document):

```
248954687985216 tid 191 disp 3 st Z et 7473
    creat 248954684456346 [246,w,C,2,-!*;245,r,C,2,--*;]
```

`248954687985216`
: Timestamp when the task returned to the worker. Note that timestamps are taken with the Linux operating system's `clock_gettime` function, and the unit is nanoseconds, although the actual precision is implementation specific of the clock used.

`tid 191`
: Unique task identifier of the task.

`disp 3`
: Dispatch counter for that task, in this case task 191 has been dispatched 3 times in total.

`st Z`
: State of the task. Can be one of Z=zombie, R=ready, Bi=blocked on input, Bo=blocked on output, Ba=blocked on any (in poll operation).

`et 7473`
: Execution time of the dispatch, i.e., the difference between the timestamp in the first field and a timestamp taken before the dispatch of the task. Hence, in this example, the start time of the task can be computed as $248954687985216 - 7473 = 248954687977743$.

`creat 248954684456346`
: This field is only output if the state is Z. It returns the timestamp taken when the task was created.

`[246,w,C,2,-!*;245,r,C,2,--*;]`
: List of streams the task used during the last dispatch, and the activity on each stream. Streams are separated by a semicolon.

The stream activity field is explained using the example `246,w,C,2,-!*`:

`246`
: Unique stream identifier. Only two tasks can communicate over a stream with this id.

`w`
: Mode. Can be either 'w' or 'r'.

`C`
: State. One of O=opened, C=closed, or I=in use.

`2`
: Total items written (or read, if mode='r') to that stream.

`-!*`
: Activity flags. If no flag is set, the pattern is '`---`'.

The first flag ('?' if set) indicates that the task is blocked on that stream. The second flag '!' indicates, that reading/writing (dependent on mode) unblocked the task on the other side of the stream.
The third flag '*' indicates that items have been read/written to the stream.

For a worker debug log entry, the format is a timestamp, followed by three asterisks and the worker debug message, for example (the line is wrapped only in this document),

```
248955428585140 *** Worker 1 exited.
    wait_cnt 16, wait_time 0.923745482
```

This message indicates that the worker exited, and during its execution it was blocked 16 times waiting for a new message to arrive at the mailbox, with a total waiting time of about 0.92 seconds.

The format of the monitoring log makes it human-readable and easily parsable. For evaluation, scripts can easily be employed to analyse the monitoring logs of an execution. The time for writing the monitoring log could be reduced by using a binary format. This is easily achieved as the code for emitting the monitoring data is located in a separate module. Note that the time for writing the monitoring data for a task dispatch is not included in the measured task execution time.

In the next chapter, some experiments conducted for evaluation of the prototype implementation are described. It will also be shown how the monitoring information can be used for analysing executions.

Chapter 6

Evaluation

In this chapter we present the results of some experiments with the prototype implementation of the S-Net runtime system on top of the LPEL. In all experiments, we compare the original pure kernel-level threaded (KLT) S-Net runtime system implementation with the new LPEL-based implementation.

In the first part of this chapter, we conduct an experiment regarding the performance with many tasks under constant data throughput. Another experiment is targeted on testing performance when tasks are constantly created and destroyed. Also, the impact of monitoring on execution time is tested.

In the second part of this chapter, we conduct an experiment to assess how the information provided by monitoring can be used to derive more advanced strategies for assignment and scheduling.

6.1 Performance Benchmarks

The performance benchmarks have been performed on two different platforms. The first platform was a Pentium U4100 dual-core notebook with 2x1.3 GHz and 4 GB of RAM, a Ubuntu Linux 10.04 distribution with kernel version 2.6.32-28 SMP x86_64.

The second platform was a node in a network cluster, a 4-processor, 48-core system with 256 GB of RAM (4 sockets x 12 cores, 2.2 GHz each, Opteron 6174), QDR Infiniband, and and a Fedora Core 13 Linux distribution with kernel version 2.6.35-rc4 SMP x86_64.

6.1.1 Deep replication pipeline.

Setup

The first benchmark is an S-Net network only consisting of a box within a serial replicator. The network definition is as follows:

```
net star {
  box foo((A) -> (A) | (B));
} connect foo*{B};
```

A stream of 1000 records is sent through the network, with each record containing a field `A` with an integer value. The box reads the integer value, and

if it is greater than zero, it decrements the value and writes it to the output stream. If the value equals zero, a record containing a termination pattern `B` is emitted, which then leaves the replication pipeline and the network. So the record passes as many pipeline stages as specified by its value field.

For each input sequence of 1000 records, the maximum pipeline depth (i.e., the maximum number for the value field) was specified. Then for each record of the sequence, a number between 0 and the specified maximum depth was chosen randomly. This sequence was then used for each experiment for that pipeline depth. The reason for choosing random values for each record is to force the collector to read records from more than one of its input streams, in a random order, hence stressing the poll operation.

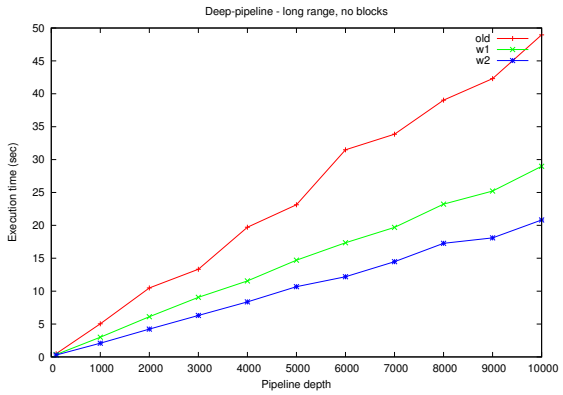
The benchmark has been executed on both platforms, the dual-core notebook and the 48-core cluster node. With the default assignment strategy, the box-tasks are assigned to the workers in a round-robin fashion. As a result, in each pipeline stage communication among workers is necessary. Another assignment strategy has been tried which assigns *blocks* of contiguous pipeline stages instead of single box-tasks to the workers, to reduce the communication overhead among them. The benchmark has been performed with various numbers of workers. For the LPEL-based implementation, also the impact of using different monitoring levels was investigated. The aim of this benchmark was to test the performance with a large number of tasks and constant throughput.

In all experiments, the wall clock time was measured, i.e., the overall time from start to the end of the computation, including initialisation, as perceived by the user. As a baseline, the execution time of the pure KLT S-Net runtime-system implementation was considered, which is labelled as "old" in the diagrams, and drawn with a thicker line. For both implementations, a fixed buffer size of ten records was used.

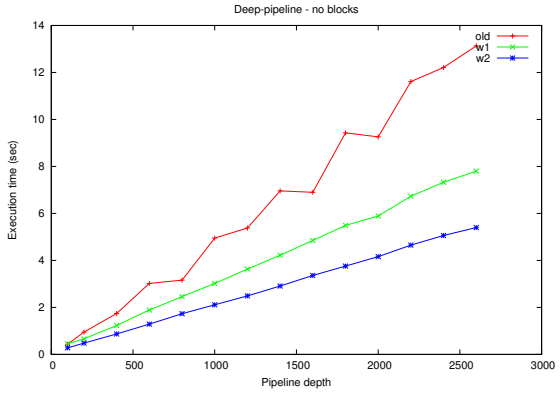
Results

Dual-core. Figure 6.1 shows the results for the deep replication pipeline benchmark on the dual-core machine. (a) shows experiments up to a pipeline depth of 10000. The LPEL variant performs better than the pure KLT implementation even with a single worker. Two workers lead to another speedup of $\sim 30\%$, compared to one worker. A speedup of 50% is not achieved due to the communication overhead between the workers. But the diagram shows that with two workers the execution is more than twice as fast as with the pure KLT based implementation.
(b),(c) show the situation for a shorter range. Grouping tasks in blocks does not cause a significant improvement. But what can be seen is that the graph of the old implementation is not necessarily monotonically increasing. This variability can be attributed to the operating system's scheduling effort.

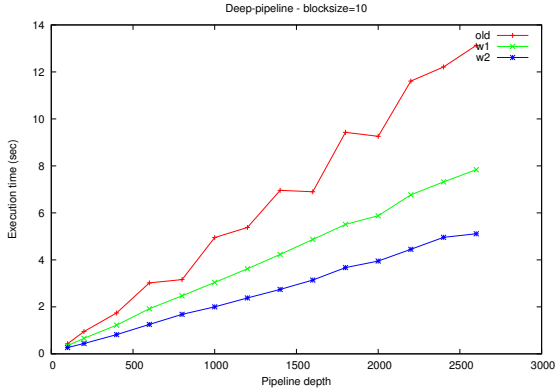
The overhead caused by monitoring is depicted in Figure 6.2. The monitoring levels of the diagram (0,m2,m3,m4) correspond to the monitoring levels (a,b,c,d) described in Section 5.4.3. It can be seen that the proportions of the overhead are preserved, with one and two workers. Using the highest monitoring level with one worker performs still better than the old S-Net implementation.



(a) Long-range, no blocks

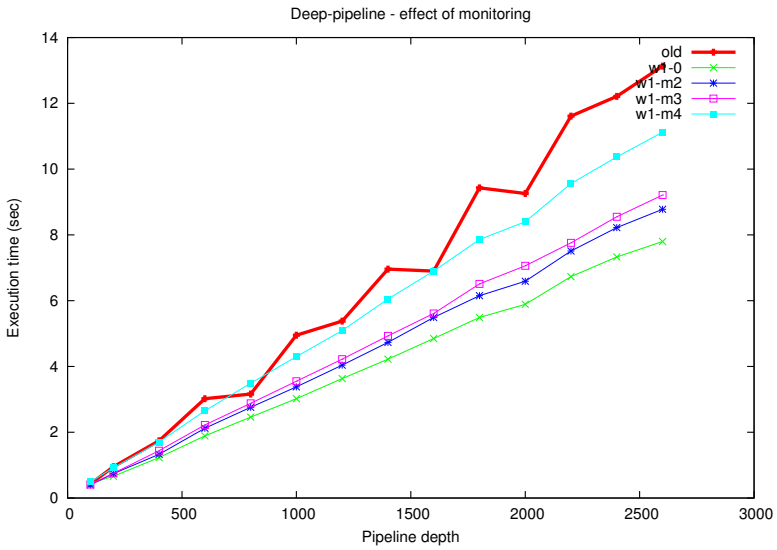


(b) Short-range, no blocks

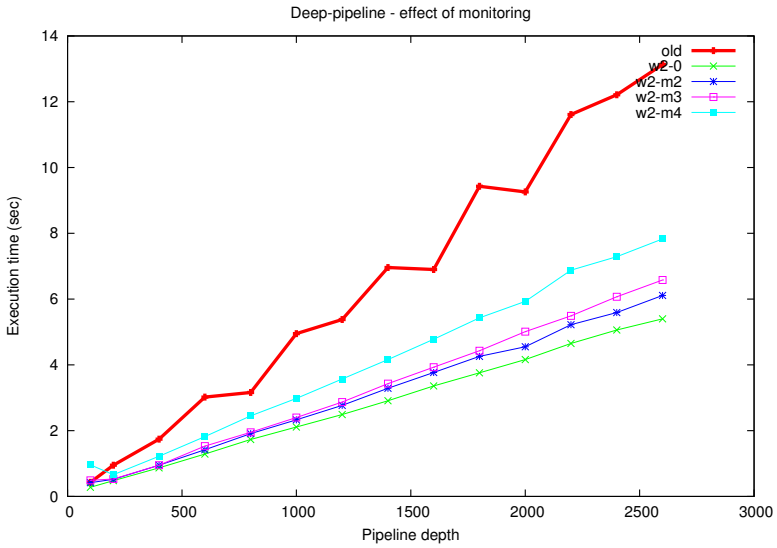


(c) Short-range, blocksize=10

Figure 6.1: Results of the deep replication pipeline benchmark, on the dual-core machine.



(a) Different monitoring levels, one worker



(b) Different monitoring levels, two workers

Figure 6.2: Impact of monitoring on the execution time, on the dual-core machine.

48-core. The results for the experiments on the 48-core machine are depicted on Figures 6.3, 6.4 and 6.5. The diagrams show executions with different numbers of worker threads. The performance improvements are not as outstanding as on the dual-core machine. Furthermore, only few executions, with a specific number of workers, can compete with the old implementation. On a short range, with no blocks, no execution with the LPEL was better than the old implementation. With only two workers, the no-blocks assignment had the worst performance, it was even twice as slow as with only one worker. The situation improves when employing the block-assignment strategy, towards a large number of tasks. Choosing a large blocksize comes with a penalty for shorter pipelines.

It is also notable that the best executions were performed with a number of about 16 or 24 workers, which presumably results from increased communication overheads among the workers or adverse caching effects as each four processing cores share their L3 caches. Executions with more workers lead to worse performance. What cannot be seen on Figures 6.3 and 6.4 is, that the baseline of the execution of the old implementation is "smoothed". Figure 6.5 (a) shows several single executions of the old implementation in one diagram. With an increasing pipeline depth, the execution times expose a big variability, whereas the execution times for LPEL executions tend to be more predictable with a number of workers of up to 32.

For a pipeline depth greater than 2500, the old implementation suffers scalability as Figures 6.5(b,c) show, whereas the LPEL-based implementation can cope with the situation very well.

Figure 6.6 shows the effect of different monitoring levels on executions with 16 and 24 workers on the 48-core machine. Here, monitoring seems to have a greater impact. A significant part of it can be attributed to the fact that the workers outputting the monitoring logs compete for writing to the file-system, and the more workers exist, the more synchronisation on behalf of the operating system and the IO facilities is required.

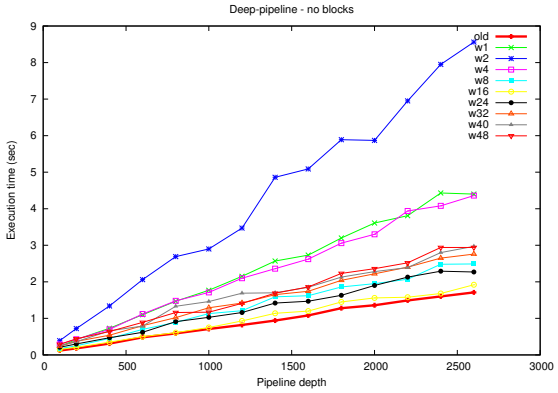
6.1.2 Synchronisation Pipeline.

Setup.

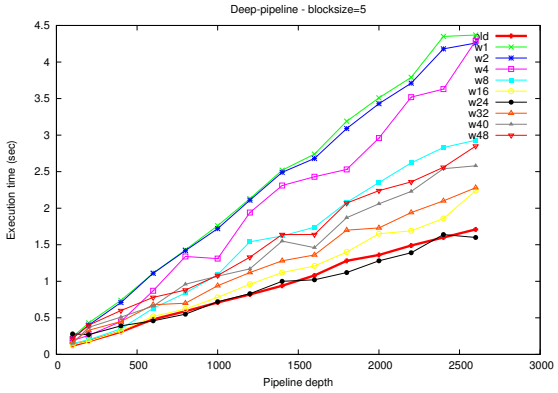
This benchmark is aimed at testing performance of task creation and destruction. A synchrocell is placed in a serial replication combinator:

```
net synchro connect [|{A},{B}|]*{A,B};
```

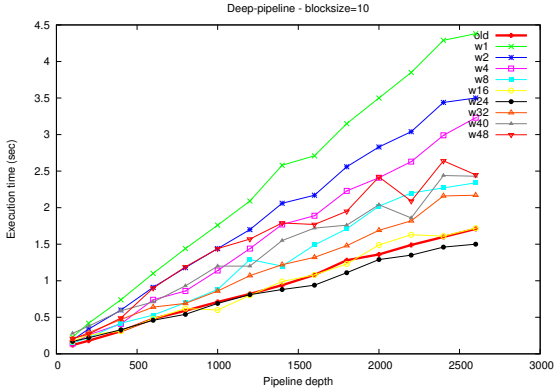
The synchrocell merges a record containing a field A and a record containing a field B, and emits a record containing both A and B. The emitted record matches the termination pattern and leaves the pipeline immediately. As a synchrocell is destroyed after a single merging of records, for continuous synchronisation subsequent stages of the serial replication pipeline are created. Also, if a record with field A is sent to a synchrocell that has already received a record with field A, the record is forwarded to the next pipeline stage. For the benchmark, sequences of five records containing A and five records containing B are sent into the network, i.e., the *synchronic distance* between A and B is Five. The process is repeated N times. These sequences will lead to an unfolding of the serial replication pipeline (task creation) of five times, and then, as



(a) Short-range, no blocks



(b) Short-range, blocksize=5



(c) Short-range, blocksize=10

Figure 6.3: Results of the deep replication pipeline benchmark, on the 48-core machine.

the "merge partner" records arrive, to destruction of the synchrocells. Subsequent records cause again unfolding of the replication pipeline, etc. Hence, the system is faced with permanent task creation and destruction. In the results, N is referred to as pipeline depth. Again, the wall clock time is measured.

Results.

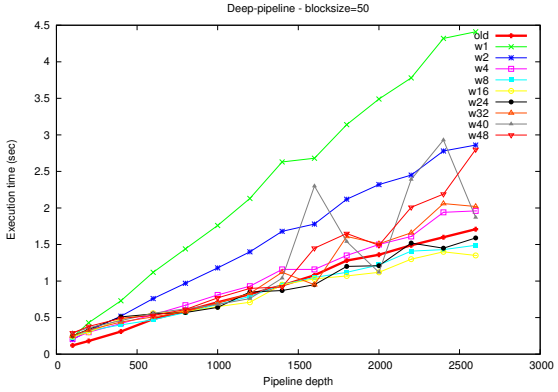
Figure 6.7 shows the results of the synchronisation pipeline benchmark, for the dual-core machine. The LPEL-based implementation has a speedup of about two. The number of workers, one or two, does not make any difference.

Figure 6.8 shows the results for the 48-core machine. An interesting fact is that for depths greater than 300, the execution with the old implementation failed, as no kernel-level thread could be created anymore. The LPEL-based implementation yielded the best result for the execution with 16 workers.

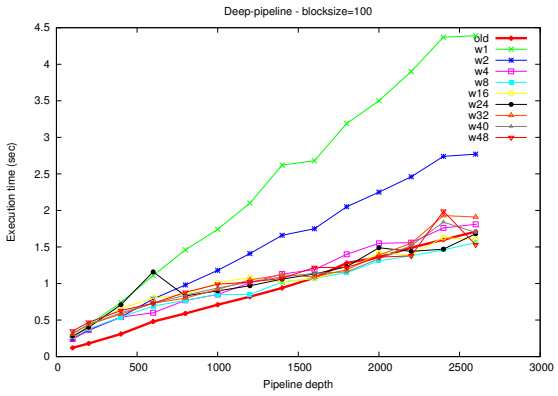
6.1.3 Discussion

The performance benchmarks have shown that the LPEL-based implementation is able to efficiently cope with a very high number of tasks. Although its scalability on the 48-core machine was observed to be be more limited than on the dual-core machine, the fact that not all cores were busy with computation for yielding the best result shows that with the LPEL-layer it is possible to make deliberate use of processing resources.

What might also be considered a sequential bottleneck, effectively limiting scalability, is the single collector-task at the merge-point where all records have to pass through. This bottleneck surely affects both implementations, as it is inherent to the S-Net network.

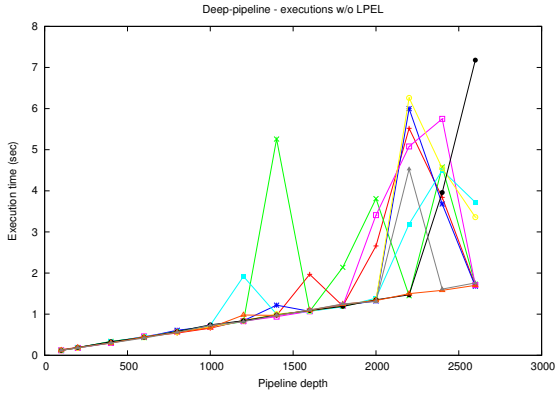


(a) Short-range, blocksize=50

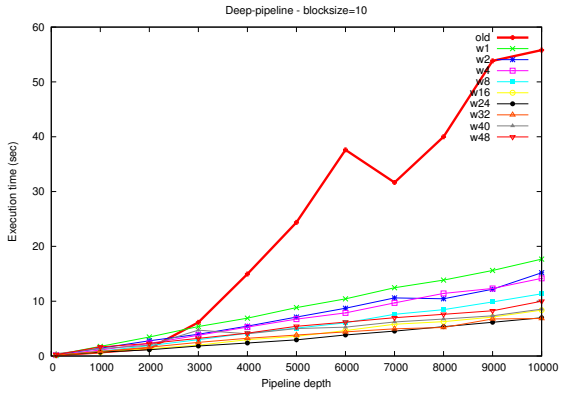


(b) Short-range, blocksize=100

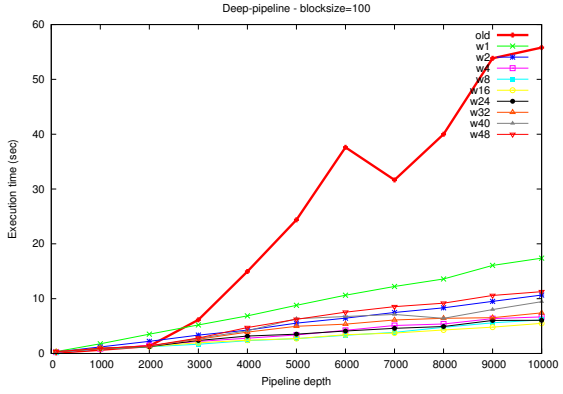
Figure 6.4: Results of the deep replication pipeline benchmark, on the 48-core machine. (ctd.)



(a) Short-range, several executions of the old implementation

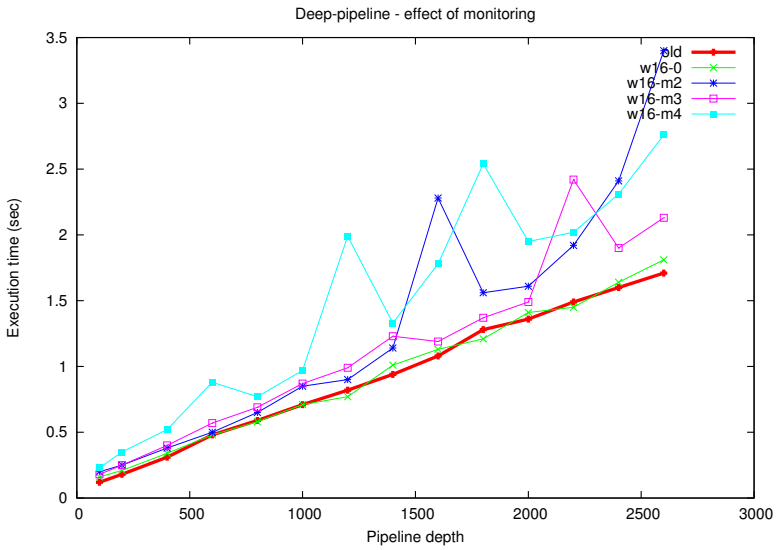


(b) Long-range, blocksize=10

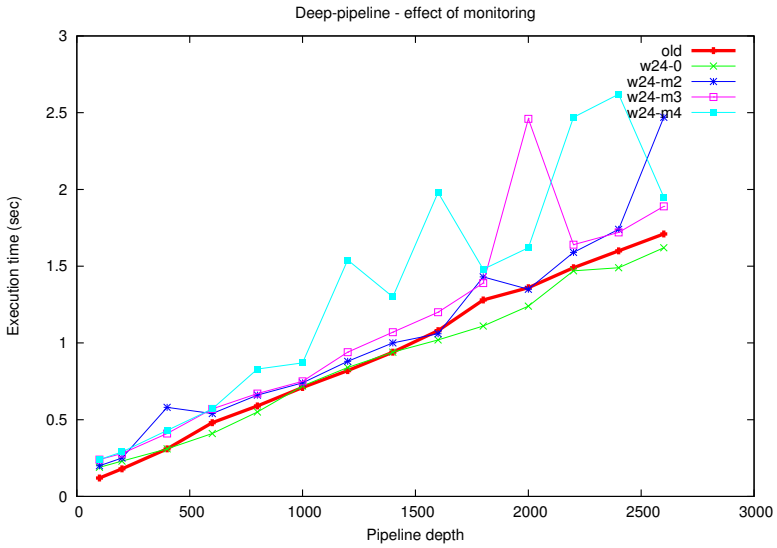


(c) Long-range, blocksize=100

Figure 6.5: Results of the deep replication pipeline benchmark, on the 48-core machine. (ctd.2)



(a) Different monitoring levels, 16 workers



(b) Different monitoring levels, 24 workers

Figure 6.6: Impact of monitoring on the execution time, on the 48-core machine.

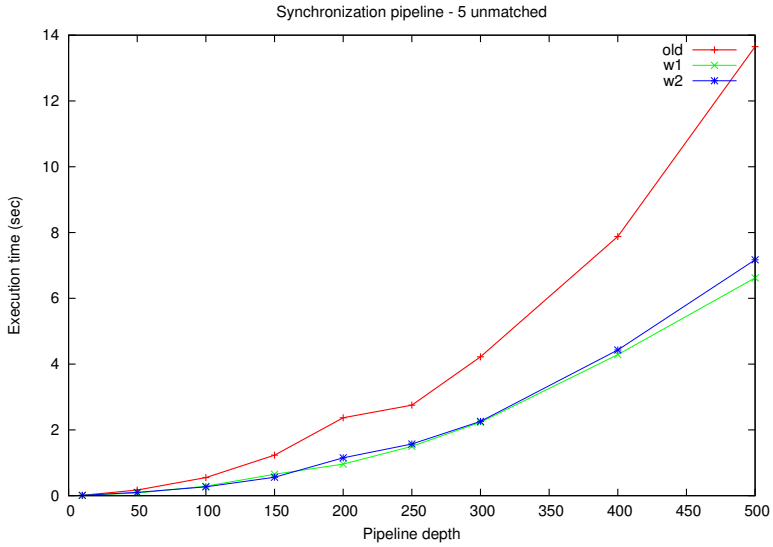
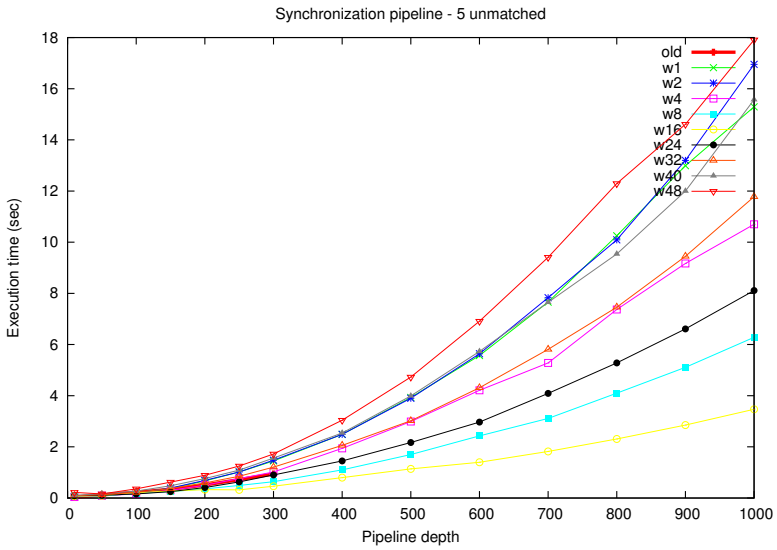
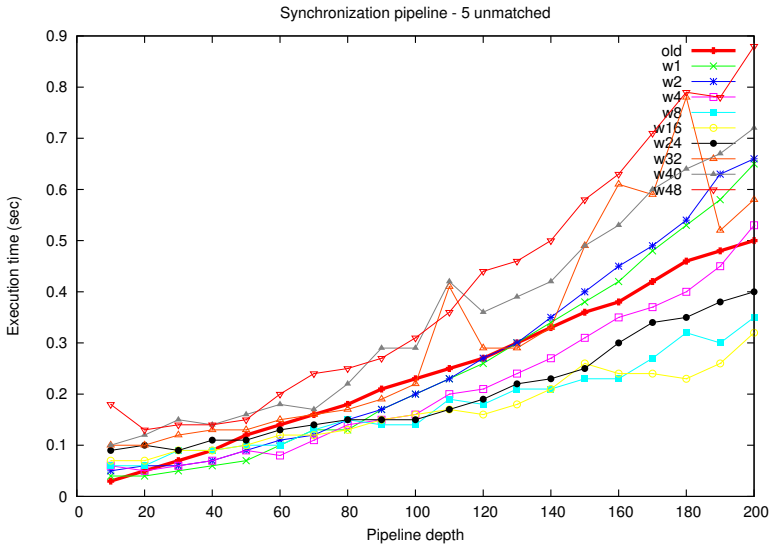


Figure 6.7: Results of the synchronisation pipeline benchmark for the dual-core machine.



(a) Depth 1000



(b) Depth 200

Figure 6.8: Results of the synchronisation pipeline benchmark for the 48-core machine.

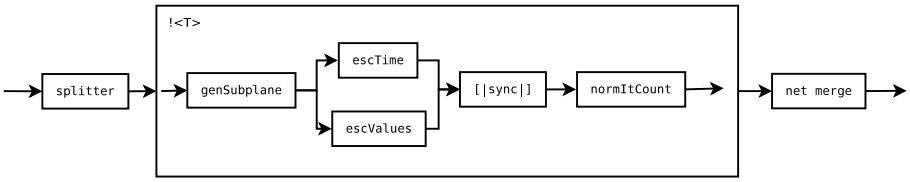


Figure 6.9: The Mandelbrot example network.

6.2 Experimenting with Assignment and Scheduling

As a case study, a real-world example has been selected for analysing monitoring information, and to explore how this information can be used to devise strategies for assignment and scheduling. As no intelligent task-to-worker assignment and no dynamic load-balancing is used in the LPEL-based S-Net runtime system, the pure KLT-based S-Net runtime system is expected to outperform it initially. The goal is to achieve a comparable performance by deriving a more diligent task-to-worker assignment and (worker-local) scheduling policy.

The example network is an application to compute a fractal image, based on Mandelbrot sets. The S-Net network is depicted in Figure 6.9. It reads a single record from the global input stream, defining the view port on the complex plane. After computing the initial values for that plane, it is split into strips that can be processed independently by the sub-net depicted within the parallel replication combinator. After the parallel replication, the strips are merged together again by a "merge" sub-net.

The experiment was performed on the dual-core machine which was also used for the benchmarks in the previous section. The plane was cut into eight strips to be computed independently. The execution time of using both the old and the LPEL-based implementation was measured, using a simple round-robin assignment for tasks to workers and a FIFO scheduling policy on each worker. For benchmarking execution time, an average out of ten executions was considered.

The execution time of the old implementation was 2.970 seconds, whereas the execution time of the LPEL-based version was 3.170 seconds. Analysing the monitoring logs produced by the LPEL-based implementation revealed, by calculating the average box execution times, that the three boxes `escValues`, `escTime` and `normItCount` are computationally intensive, whereas the other boxes can be neglected.

6.2.1 Balanced Assignment

More precisely, analysing the monitoring logs of the workers separately, we obtained the total and average execution times of the tasks of worker 0:

```
...
*** genSubPlane: total 0.000000, avg 0.000000
*** splitter: total 0.000178, avg 0.000178
*** escTime: total 1.631908, avg 0.203989
*** normItCount: total 0.360298, avg 0.045037
*** <sync>: total 0.000000, avg 0.000000
```

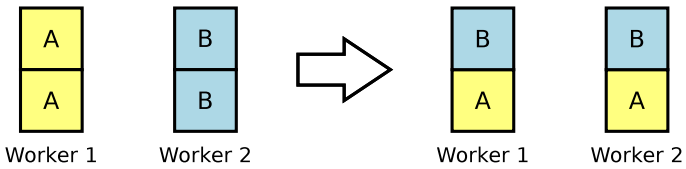


Figure 6.10: Distributing equal boxes on the workers.

```
*** merge: total 0.004329, avg 0.000541
*** init: total 0.000000, avg 0.000000
*** <parallel>: total 0.000176, avg 0.000007
*** escVals: total 0.000000, avg 0.000000
*** <collector>: total 0.000560, avg 0.000016
*** <split>: total 0.000000, avg 0.000000
...
```

Similarly, we obtained for the tasks of worker 1:

```
...
*** genSubPlane: total 0.306727, avg 0.038341
*** splitter: total 0.000000, avg 0.000000
*** escTime: total 0.000000, avg 0.000000
*** normItCount: total 0.000000, avg 0.000000
*** <sync>: total 0.000531, avg 0.000033
*** merge: total 0.000000, avg 0.000000
*** init: total 0.005793, avg 0.000724
*** <parallel>: total 0.000764, avg 0.000032
*** escVals: total 1.861941, avg 0.232743
*** <collector>: total 0.000824, avg 0.000024
*** <split>: total 0.000601, avg 0.000601
...
```

Effectively, all instances of `normItCount` and `escTime` were assigned to worker 0, whereas all instances of `escValues` were assigned to worker 1. This kind of load imbalance, resulting from a simple round-robin assignment of tasks to workers can be solved by keeping track of how many instances of each box are assigned to each worker. With this information, it is tried to distribute equal boxes on the workers, as depicted in Figure 6.10.

Measuring the execution time with the adapted assignment module lead to an average execution time of 3.013 seconds, which was still higher than the execution time of the pure KLT based S-Net runtime-system implementation.

6.2.2 Priorities for Scheduling

The monitoring logs for one of the new executions revealed that the workers had imbalanced wait-times, blocking on their mailbox. (The last entries of 0.740 are the wait times until the workers received the message to signal termination.)

```
...
248954687985216 tid 191 disp 3 st Z et 7473
      creat 248954684456346 [246,w,C,2,-!*;245,r,C,2,--*;]
248954687997369 *** worker 0 waited (18) for 0.000056292
```

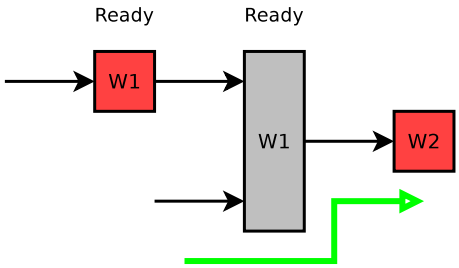


Figure 6.11: Prioritising non-box tasks over computationally intensive box-tasks which could prevent another worker from progress.

```
248954688005261 tid 4 disp 19 st Z et 4470
      creat 248952382164573 [6,w,C,2,--*;246,r,C,2,--*;]
248955428427159 *** worker 0 waited (19) for 0.740413447
248955428433514 *** Worker 0 exited. wait_cnt 19, wait_time 1.034697843
248955428578225 *** worker 1 waited (16) for 0.740584349
248955428585140 *** Worker 1 exited. wait_cnt 16, wait_time 0.923745482
```

One way to reduce the wait times was assumed to be prioritising "non-box tasks", which have a shorter execution time. This would prevent workers to execute computationally intensive box tasks, while other (shorter) tasks could be dispatched that could lead to earlier unblocking of tasks on other workers. Figure 6.11 illustrates this situation. Assume on worker W1 there are two ready tasks, one a computationally intensive box and one a collector that could forward data from one of its inputs to its output. At the same time on W2, there are no ready tasks, but one task that blocks on the collector's output. If worker W1 schedules the collector, this will cause worker W2 to dispatch the task blocked on the collector's output, whereas by dispatching the box, W2 will be further delayed.

Taking this into consideration, the scheduling module was extended by another task-queue, for employing a priority-based scheduling policy with two priorities: "non-box tasks" with a high priority, and box tasks with a low priority. The scheduler first fetches all (ready) tasks from the high priority queue before fetching tasks from the low priority queue. Tasks becoming unblocked are inserted into the appropriate queue according to their priority. This was, due to the modular design of the LPEL, easily achieved. Execution with both balanced assignment and priority based scheduling lead to an average execution time of 2.871 seconds, displaying the best performance.

The monitoring logs confirmed the initial assumption, by showing (almost) balanced wait times for the workers.

```
...
249246590900603 tid 4 disp 17 st Z et 10895
      creat 249244526826468 [6,w,C,2,-!*;246,r,C,2,?-*;]
249247337186439 *** worker 1 waited (16) for 0.746295753
249247337194540 *** Worker 1 exited. wait_cnt 16, wait_time 0.762573869
249247337324864 *** worker 0 waited (16) for 0.746413225
249247337426483 *** Worker 0 exited. wait_cnt 16, wait_time 0.750208474
```

Table 6.1 summarises the execution times.

Runtime system and configuration	Execution Time
Pure Kernel-level threaded runtime system	2.970 sec
LPEL-based runtime system, naive assignment	3.170 sec
LPEL, with balanced assignment	3.013 sec
LPEL, with bal. ass. +priority scheduling	2.871 sec

Table 6.1: Summary of achieved average execution times for the Mandelbrot example network.

6.2.3 Discussion

Although the results from the experiments with the Mandelbrot example network cannot simply be generalised, they provide insight of how the information collected by monitoring can be used to create assignment and scheduling policies for real-world and large-scale applications.

It also has to be pointed out that the Mandelbrot example only one input record from the top-level network input stream was processed. In this case, the efficient stream synchronisation mechanisms of the LPEL are not leveraged to their full extent. Performance gains are expected to be higher with applications that have a high and constant data throughput, such that the efficient stream synchronisation and light-weight task handling mechanisms are exploited, e.g., the application domain of signal-processing.

Chapter 7

Conclusion

7.1 Summary

This thesis presented the design and implementation of a light-weight user-task management layer (LPEL) for stream processing. It presented related concepts, starting with computational models for stream-processing, following user-level task management and concurrent data structures, which allow efficient and scalable application designs. Runtime systems for stream processing were presented, and the multi-threaded runtime system implementation of S-Net was explained in more detail. Finally, the prototype implementation of the LPEL, and of S-Net on top of the LPEL were benchmarked.

The architecture of the LPEL allows deliberate allocation of available processing resources. Lock-free synchronisation techniques and user-level threading make it possible to handle a large number of tasks simultaneously. Due to its modular and flexible approach, different assignment and scheduling policies can easily be incorporated. Also, detailed profiling information during an execution of a streaming network can be gathered. Adopting the LPEL for the S-Net runtime system makes the latter open for extensions towards profiling-based dynamic reconfiguration and for running (soft) real-time applications, by employing appropriate scheduling policies.

7.2 Future Work

The next step in the development of LPEL is to provide dynamic load balancing strategies to be able to cope with inevitable load imbalance during runtime. In conjunction with S-Net, appropriate scheduling and assignment heuristics and policies have to be investigated, preferably considering the measured execution time of tasks and the network topology. Scheduling policies for applications that have to meet (soft) real-time constraints have to be developed. Ways have to be deduced how to use the monitored profiling information for dynamic reconfiguration decisions. Concepts for how S-Net applications can provide fault-tolerance and quality of service need to be examined.

Bibliography

[ADM+09] Marco Aldinucci, Marco Danelutto, Massimiliano Meneghin, Peter Kilpatrick, and Massimo Torquati. Efficient streaming applications on multi-core with FastFlow: the biosequence alignment test-bed. In Barbara Chapman, Frédéric Desprez, Gerhard R. Joubert, Alain Lichnewsky, and Frans Peters anf Thierry Priol, editors, *Parallel Computing: From Multicores and GPU's to Petascale (Proc. of PARCO 2009, Lyon, France)*, volume 19 of *Advances in Parallel Computing*, pages 273–280, Lyon, France, September 2009. IOS press.

[Amd67] G.M. Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference, Atlantic City, New Jersey, USA*, pages 483–485. AFIPS Press, Reston, Virginia, USA, 1967.

[AR06] Shameem Akhter and Jason Roberts. *Multi-Core Programming: Increasing Performance through Software Multithreading (Programming)*. Intel Press, 1st edition, 2006.

[BH01] T. Basten and J. Hoogerbrugge. Efficient Execution of Process Networks. In Alan G. Chalmers, Majid Mirmehdi, and Henk Muller, editors, *Communicating Process Architectures 2001*, pages 1–14, sep 2001.

[Buc93] Joseph T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. PhD thesis, EECS Department, University of California, Berkeley, 1993.

[dES+00] E.A. de Kock, G. Essink, W.J.M. Smits, P. van der Wolf, J.-Y. Brunel, W.M. Kruijtzer, P. Lieverse, and K.A. Vissers. YAPI: Application modelling for signal processing systems. In *Proc. of the 37th Design Automation Conference,*, pages 402–405, 2000.

[For94] Message P Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[GFSF11] GCC GNU Free Software Foundation. *GCC Reference Manual*, 2011, (accessed Jan 2011). URL: `http://gcc.gnu.org/onlinedocs/gcc/Atomic-Builtins.html`.

[GJP09] Clemens Grelck, Jukka Julku, and Frank Penczek. Distributed S-Net. In M. Morazan, editor, *Implementation and Application of Functional Languages, 21st International Symposium, IFL'09, South Orange, NJ, USA*. Seton Hall University, 2009.

[GMV08] John Giacomoni, Tipp Moseley, and Manish Vachharajani. FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 43–52, New York, NY, USA, 2008. ACM.

[GP10] 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*. Springer-Verlag, 2010.

[GSS10] Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. Asynchronous stream processing with S-Net. *International Journal of Parallel Programming*, 38:38–67, 2010. 10.1007/s10766-009-0121-x.

[Ins95] Institute of Electrical and Electronic Engineers, Inc. Information Technology — Portable Operating Systems Interface (POSIX) — Part: System Application Program Interface (API) — Amendment 2: Threads Extension [C Language]. IEEE Standard 1003.1c–1995, IEEE, New York City, New York, USA, 1995. also ISO/IEC 9945-1:1990b.

[Kah74] G Kahn. The semantics of a simple language for parallel programming. In L Rosenfeld, editor, *Information Processing 74, Proc. IFIP Congress 74. August 5-10, Stockholm, Sweden*, pages 471–475. North-Holland, 1974.

[KM77] Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In *Information Processing 77*, pages 993–998. North Holland Publishing Company, 1977.

[Lam83] Leslie Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, 5(2):190–222, 1983.

[Lib11] Davide Libenzi. *GNU Portable Coroutine Library*, 2011 (accessed Jan 2011). URL: `http://www.xmailserver.org/libpcl.html`.

[LM87] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. In *Proceedings of the IEEE*, volume 75, pages 1235–1245, September 1987.

[MS96] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, pages 267–275, 1996.

[MS04] M. Moir and N. Shavit. Concurrent data structures. In *Handbook of Data Structures and Applications, D. Metha and S. Sahni Editors*, pages 47–14 — 47–30, 2004. Chapman and Hall/CRC Press.

[Par95] Thomas M Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California, Berkeley., 1995.

[Pen07] Frank Penczek. Design and Implementation of a Multithreaded Runtime System for the Stream Processing Language S-Net. Master's thesis, Institute of Software Technology and Programming Languages, University of Lübeck, Germany, 2007.

[PSG08] Frank Penczek, Sven-Bodo Scholz, and Clemens Grelck. Towards reconfiguration and self-adaptivity in S-Net. In Sven-Bodo Scholz, editor, *Implementation and Application of Functional Languages, 20th international symposium, IFL'08, Hatfield, Hertfordshire, UK*, Technical Report 474, pages 330–339. University of Hertfordshire, England, UK, 9 2008.

[Sta09] William Stallings. *Operating Systems. Internals and Design Principles*. Prentice Hall, 6th edition, 2009.

[Tre86] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ5118, IBM Almaden Research Center, April 1986.

[VHG+09a] Željko Vrba, Pal Halvorsen, Carsten Griwodz, Paul Beskow, and Dag Johansen. The Nornir run-time system for parallel programs using Kahn process networks. In *NPC '09: Proceedings of the 2009 Sixth IFIP International Conference on Network and Parallel Computing*, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[VHG09b] Zeljko Vrba, Pål Halvorsen, and Carsten Griwodz. Evaluating the run-time performance of kahn process network implementation techniques on shared-memory multiprocessors. In NA, editor, *Complex, Intelligent and Software Intensive Systems, International Conference*, pages 639–644. IEEE Computer Society, 2009.

[Vrb09] Zeljko Vrba. *Implementation and performance aspects of Kahn process networks - an investigation of problem modeling, implementation techniques, and scheduling strategies*. Phd thesis, Simula Research Laboratory / University of Oslo, Unipub, Kristian Ottosens hus, Pb. 33 Blindern, 0313 Oslo, 2009.