
Software Plattform Embedded Systems 2020

GEFÖRDERT VOM


# Deliverable D5.2.B (Werkzeugprototyp): Umsetzung von Strategien zum modellbasierten Deployment

## Version: 1.0

| | | |
|---|---|---|
| Projektbezeichnung | SPES 2020 | |
| Verantwortlich | Tobias Schüle | |
| QS-Verantwortlich | Andreas Schramm, Fraunhofer FIRST | |
| Erstellt am | 06.12.2010 | |
| Zuletzt geändert | 16.12.2010 15:55 | |
| Freigabestatus | | Vertraulich für Partner: <Partner1>; <Partner2>; … |
| | | Projektöffentlich |
| | X | Öffentlich |
| Bearbeitungszustand | | in Bearbeitung |
| | | vorgelegt |
| | X | fertig gestellt |

## Weitere Produktinformationen

| | |
|---|---|
| Erzeugung | Tobias Schüle |
| Mitwirkend | Wolfgang Schwitzer |

## Änderungsverzeichnis

| Änderung | | | Geänderte Kapitel | Beschreibung der Änderung | Autor | Zustand |
|---|---|---|---|---|---|---|
| Nr. | Datum | Version | | | | |
| 1 | 06.12.10 | 1.0 | Alle | Initiale Produkterstellung | T. Schüle | |
| 2 | 16.12.10 | 1.1 | 1.2 | Erläuterung nichtlinearer Strukturen | T. Schüle | |

# Kurzfassung

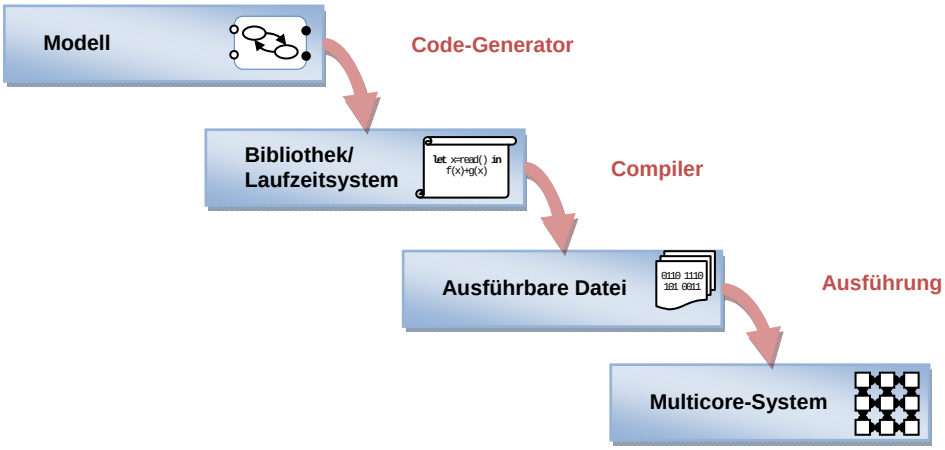
Dieses Dokument beschreibt einen im Arbeitspaket 5 des SPES-Zentralprojekts entwickelte Ansatz für die parallele Ausführung von Software für eingebettete Systeme. Darauf aufbauend wurde eine Bibliothek implementiert, die als Grundlage für die automatische Code-Generierung in der modellbasierten Software-Entwicklung verwendet werden kann. Die Bibliothek bildet somit eine technische Schnittstelle zwischen den in ZP-AP1 und ZP-AP5 untersuchten Methoden und Werkzeugen.

# Inhalt

# 1 Einordnung und Kurzbeschreibung

## 1.1 Motivation und Einordnung

In eingebetteten Systemen kommen zunehmend Multicore-Prozessoren zum Einsatz, bei denen mehrere unabhängige Prozessorkerne auf einem Chip untergebracht sind. Um von den Vorteilen solcher Prozessoren profitieren zu können, müssen darauf auszuführende Anwendungen parallelisiert werden. Dies erfordert jedoch insbesondere bei der Entwicklung sicherheitsrelevanter eingebetteter Systeme neue Programmiermethoden und -werkzeuge. Hierfür bieten sich modellbasierte Verfahren an, mit deren Hilfe sich aus formalen Beschreibungen unter Abstraktion von implementierungsspezifischen Details automatisch Software erzeugen lässt. Die Generierung der Software basiert dabei üblicherweise auf in der Zielsprache implementierten Bibliotheken bzw. Laufzeitsystemen, die wiederkehrende Grundfunktionen zur Verfügung stellen. Die folgende Abbildung zeigt beispielhaft den Entwurfsablauf vom Modell bis zur Ausführung der generierten Software auf einem Multicore-System:




Für die Beschreibung der Modelle eignen sich insbesondere die im Arbeitspaket ZP-AP1 untersuchten Methoden und Werkzeuge für die modellbasierte Entwicklung eingebetteter Systeme (vgl. [1]) sowie die in ZP-Task 5.1 vorgeschlagene Koordinierungssprache [2] (siehe Deliverable D5.1.A).

## 1.2 Management Summary

Im Arbeitspaket 5 des Zentralprojekts wurde eine C++-Bibliothek inkl. Laufzeitsystem (Middleware) entwickelt, die als Grundlage für die Code-Generierung beim modellbasierten Entwurf dienen kann. Die Bibliothek ermöglicht eine plattformübergreifende, parallele Ausführung datenstromverarbeitender Systeme auf Multicore-Rechnern durch Abstraktion von der zugrunde liegenden Hardware.

Ein wesentliches Merkmal der Bibliothek ist die Unterstützung nichtlinearer Strukturen. Ein System hat eine nichtlineare Struktur, wenn Datenströme vervielfältigt oder aufgespalten bzw. vereinigt werden. Eine solche Struktur liegt z.B. bereits dann vor, wenn das Ergebnis einer Berechnung von mehreren Funktionsblöcken als Eingabe verwendet wird. Solche Systeme lassen sich mit herkömmlichen Pipelines, die aus (linearen) Ketten von Pipelinestufen bestehen, nur auf Umwegen beschreiben. Nichtlineare Strukturen haben unter anderem den Vorteil einer Reduzierung der Latenz-

zeit. Die Latenzzeit ist vor allem bei eingebetteten Systemen von Bedeutung, da diese in der Regel Echtzeitanforderungen unterliegen.

Die Bibliothek basiert auf einem deterministischen Ausführungsmodell, wodurch typische Probleme bei der Entwicklung paralleler Software wie erschwerte Testbarkeit vermieden werden. Die folgende Auflistung gibt eine Zusammenfassung der wichtigsten Eigenschaften:

- Abstraktion von der zugrunde liegenden Hardware
- Deterministisches Ausführungsmodell
- Unterstützung nichtlinearer Strukturen zur Reduzierung der Latenzzeit
- Automatische Lastbalancierung
- Skalierbarkeit durch verteilte Synchronisation und Vermeidung globaler Datenstrukturen
- Integration von bestehendem Programmteilen (legacy code)
- Wahlweise Ausführung von Operationen außerhalb der Reihenfolge (out-of-order execution)
- Bedingte Steuerung der Datenströme in Abhängigkeit der Eingabedaten
- Automatische Speicherverwaltung
- Erhöhte Typsicherheit durch statischen Polymorphismus
- Flexibilität durch Austausch von zentralen Komponenten (z.B. Scheduler)
- Hohe Effizienz durch lock-freie Algorithmen und Datenstrukturen
- Generische Programmierung unter Verwendung von Templates, Lambda-Funktionen etc.

## 1.3 Überblick

Ein detaillierte Beschreibung der Konzepte, Methoden und Algorithmen, die der Bibliothek zugrunde liegen, ist in [3] zu finden (die Veröffentlichung wurde diesem Dokument als Anhang beigefügt). Deshalb wird an dieser Stelle auf eine weiterführende Betrachtung verzichtet.

# 2 Literaturverzeichnis

[1] M. Broy, K. Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer, 2001

[2] T. Schüle. *A Coordination Language for Programming Embedded Multi-Core Systems*. International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT). IEEE, 2009

[3] T. Schüle. Efficient Parallel Execution of Streaming Applications on Multi-Core Processors. International Conference on Parallel, Distributed and Network-Based Computing (PDP). IEEE, 2011

# Efficient Parallel Execution of Streaming Applications on Multi-Core Processors

Tobias Schuele
*System Architecture and Platforms*
*Corporate Technology*
*Siemens AG*
*Munich, Germany*
tobias.schuele@siemens.com

***Abstract*—We propose a method for the parallel execution of applications that process continuous streams of data. Unlike pipeline-based approaches, which are frequently employed to parallelize software for multi-core processors, our method supports nonlinear structures that may contain conditionals. Nonlinear structures reduce the latency for processing an element from a stream, which is particularly important for embedded systems that are subject to real-time constraints.**

## I. Introduction

To utilize the power of multi-core and future many-core processors, computations have to be split into tasks that can be executed in parallel. In many applications that process continuous streams of data, e.g., in digital signal processing, computations are split into series of tasks such that each task processes an element from a stream. Computations on streams are thus performed in a pipelined fashion, where the tasks (pipeline stages) operate in parallel [1], [2]. In its basic form, however, the throughput of a pipeline is limited by the slowest stage. To solve this problem, multiple invocations of a stage may be executed in parallel to achieve a maximum of performance [3]. Hence, a new item, subsequently also called token, may enter a parallel stage before the previous one has left it, provided that the invocations do not interfere. For stages with side effects it is usually required that at most one invocation is active at a time. Such stages are said to be serial [4].

Another technique to improve the performance of pipelines is to process the items of a stream out-of-order. This means that the order in which the tokens enter the pipeline is not necessarily preserved. Out-of-order execution helps to reduce idle times of the processor cores that arise due to varying runtimes of parallel stages. However, certain stages may require that the tokens arrive in the original order. A typical application is video processing, where some computations depend on the order of consecutive video frames. Moreover, stages that perform input/output operations such as writing to a file on a hard disk or controlling an actuator must be processed in-order. Otherwise, the result may be wrong. In most libraries and language extensions that support pipelining like Intel's Threading Building Blocks (TBB) [4], parallel stages are always executed out-of-order, which simplifies the implementation.

Pipelining exploits parallelism by increasing the throughput, but does not decrease the latency, the time required to process an element from a stream. The latency is a crucial concern in embedded systems that have tight constraints on the timing. A reduction of the latency can be achieved if more sophisticated structures than pipelines are employed such as dataflow process networks (DPNs) [5]. In contrast to pipelines, which have a string-like (linear) structure, the actors of a DPN may be connected in an arbitrary nonlinear way. Hence, DPNs can be viewed as a generalization of pipelines. As another advantage of DPNs, the flow of tokens may be controlled dynamically depending on the input data. For example, a stream may be split into substreams using a conditional, and the substreams may be merged at the end of the conditional. However, differences in the runtimes of the branches of a conditional can lead to idle times that may significantly hurt performance. Moreover, DPNs do not support multiple parallel invocations of an actor, since all operations are performed in-order.

In this paper, we present an approach that solves these problems. It supports parallel as well as serial actors and can deal with out-of-order execution in the presence of conditionals. For that purpose, each token carries besides its value two indices that specify the token's position in the stream and the position of its predecessor. Additionally, serial actors as well as switch operations, which are used to implement conditionals, keep track of the most recently processed token. In this way, the original order of the tokens can be reconstructed even if some tokens are missing, because they have been sent to the other branch of a conditional. As a major advantage, our method reduces idle times that may occur during conventional in-order execution. Moreover, it is based on a deterministic model of computation, which simplifies testing and debugging [6]. Determinism is particularly important in embedded system design, where high demands are put on correctness and reliability (a comprehensive treatment of programming models for embedded multiprocessor systems can be found [7]).

This work has been partially funded by the German Federal Ministry of Education and Research (BMBF) as part of the alliance project SPES2020, grant 01IS08045.

Stream processing has a long history in computer science [8] and has gained resurgent interest with the emergence of multi-core processors. In recent years, various languages and libraries that facilitate parallel programming with streams have been proposed (see, e.g., [9]–[12]). A method that permits out-of-order execution of tasks in pipelines was presented in [3]. To ensure that all tokens are processed in the correct order, each token carries an associated sequence number that specifies its position in the stream. If a task must be executed in-order, the tokens are reordered according to their sequence numbers. A similar approach is implemented in TBB [4]. As in [3], however, only linear pipelines are supported. Hence, nonlinear structures must be linearized, which increases the latency and complicates the implementation. The approach presented in this paper is closely related to the tagged-token scheme proposed for executing programs on dataflow computers [13], [14]. The basic idea of this scheme is to attach to each token a tag that specifies the context. The tags ensure that only tokens of the same context are processed by an operation. However, the tagged-token scheme does not provide any means to execute certain operations in-order, since they are assumed to be side-effect free. Our approach is also inspired by Lamport's logical clocks that are used to order events in distributed systems [15]. Out-of-order execution has many other application areas. In microprocessors, for example, out-of-order execution is used to increase instruction throughput by resolving unnecessary dependencies that stem from the sequential nature of the von Neumann architecture [16].

The rest of this paper is organized as follows: In the next section, we explain the foundations of our approach and define the semantics of the basic actors. In Section III, we describe a lock-free implementation that can be efficiently executed on shared-memory systems. After that, we present experimental results in Section IV. Finally, we conclude with a summary and directions for future work (Section V).

## II. Foundations and Semantics

Dataflow process networks [5] can be viewed as a special case of Kahn process networks (KPNs) [17]. A KPN consists of a set of processes that communicate with each other through unidirectional FIFO channels. Processes read and write tokens from and to the channels, where each channel is written to by exactly one process. KPNs are deterministic under the following conditions [5], [17]: writing to a channel is non-blocking, while reading from a channel is blocking, i.e., a process that reads from an empty channel will stall and can only continue when the channel contains at least one token. Moreover, processes are not allowed to test an input channel for emptiness, and each process must be deterministic. KPNs lend themselves well for modeling parallel systems, since their behavior does not depend on computation or communication delays.

However, the blocking read condition may cause considerable overhead for context switching in case a process wants to read from an empty input channel. DPNs avoid this overhead by replacing processes with actors that are executed according to a predefined set of rules [5]. Thus, instead of immediately scheduling a process, which may block due to missing data, an actor is not scheduled until all data is available.[1] To be able to perform certain operations out-of-order, we assume that the actors communicate with each other through unordered buffers instead of FIFO channels. Hence, actors that must be executed in-order have to reconstruct the original order using the tokens' indices. Before we go into detail, however, we need a precise notion of tokens and streams.

*Definition 1:* A token of type $T$ is a triple $\langle x, i, j\rangle \in T \times \mathbb{N} \times \mathbb{N}_0$ with $j < i$, where $x$ is the value of the token, $i$ is the token's index, and $j$ is the index of the predecessor token. The set of all tokens of type $T$ is denoted by $\mathbb{T}_T$. Given a token $t = \langle x, i, j\rangle$, we define $\mathsf{idx}(t) = i$ and $\mathsf{idx}^{-1}(t) = j$. Two tokens $t_1, t_2 \in \mathbb{T}_T$ are *connected*, written $t_1 \rightsquigarrow t_2$, iff $\mathsf{idx}(t_1) = \mathsf{idx}^{-1}(t_2)$ holds. In this case, $t_2$ is a *successor* of $t_1$, and $t_1$ is a *predecessor* of $t_2$.

It should be emphasized that a token value need not necessarily be a scalar value; it may also be a compound data structure or a collection such as a vector or a matrix. Hence, a single token may represent a complex object like a network packet or a video frame.

*Definition 2:* A stream $S\colon \mathbb{N} \to \mathbb{T}_T$ is an infinite[2] sequence of tokens of type $T$. A stream $S$ is *well-formed* iff it contains exactly one token without predecessor and all other tokens have exactly one predecessor. Moreover, every token must have exactly one successor:

- $\exists n \in \mathbb{N}.\mathsf{idx}^{-1}(S(n)) = 0 \wedge \neg\exists n' \in \mathbb{N}.n' \neq n \wedge \mathsf{idx}^{-1}(S(n')) = 0$
- $\forall m \in \mathbb{N}.\mathsf{idx}^{-1}(S(m)) \neq 0 \to (\exists n \in \mathbb{N}.S(n) \rightsquigarrow S(m) \wedge \neg\exists n' \in \mathbb{N}.n' \neq n \wedge S(n') \rightsquigarrow S(m))$
- $\forall m \in \mathbb{N}.\exists n \in \mathbb{N}.S(m) \rightsquigarrow S(n) \wedge \neg\exists n' \in \mathbb{N}.n' \neq n \wedge S(m) \rightsquigarrow S(n')$

A stream $S$ is *ordered* iff the tokens occur in their original order:

- $\mathsf{idx}^{-1}(S(1)) = 0 \wedge \forall n \in \mathbb{N}. \to S(n) \rightsquigarrow S(n+1)$

Note that every ordered stream is well-formed, but not vice versa. Finally, two streams $S_1$ and $S_2$ are *compatible* iff the token indices are consistent:

- $\forall n_1, n_2 \in \mathbb{N}.\mathsf{idx}(S_1(n_1)) = \mathsf{idx}(S_2(n_2)) \to \mathsf{idx}^{-1}(S_1(n_1)) = \mathsf{idx}^{-1}(S_2(n_2))$

[1] For similar reasons, many libraries and language extensions for parallel programming such as TBB [4], OpenMP [18], [19], and Cilk [20] avoid blocking operations and perform synchronization at the level of lightweight tasks, which are scheduled among a set of worker threads.

[2] Since we are mainly concerned with non-terminating (reactive) systems, we focus on infinite streams. Nevertheless, our approach can also be applied to finite streams, e.g., file streams.

For example, a stream $S$ with $S(1) = \langle x_1, 1, 0\rangle$, $S(2) = \langle x_2, 3, 1\rangle$, and $S(3) = \langle x_3, 2, 1\rangle$ is not well-formed since $S(2)$ and $S(3)$ have the same predecessor. A stream $S$ with $S(1) = \langle x_1, 1, 0\rangle$, $S(2) = \langle x_2, 3, 2\rangle$, and $S(3) = \langle x_3, 2, 1\rangle$ is well-formed, provided that the remaining part is well-formed. However, it is not ordered since $S(2) \not\rightsquigarrow S(3)$. If $S(2)$ and $S(3)$ are swapped, the resulting stream is ordered, provided that the remaining part is ordered. Two streams $S_1$ and $S_2$ with $S_1(1) = \langle x_1, 2, 1\rangle$ and $S_2(1) = \langle y_1, 2, 0\rangle$ are not compatible, since $\mathsf{idx}^{-1}(S_1(1)) \neq \mathsf{idx}^{-1}(S_2(1))$.

The semantics of our actors is given by the transition rules depicted in Figure 1. Besides parallel and serial functions, we also consider switches and selectors, which are used to control the dataflow in a DPN (see [13], [21], [22] for a description of different kinds of operations in dataflow computing). The left-hand side of a rule shows the state before an actor is executed and the right-hand side the state after execution. As can be seen from Figure 1, a parallel function is ready for execution if all incoming tokens have the same indices $i$ and $j$. Since several instances of a parallel function may be executed simultaneously, execution of a new instance may start before any previous invocations are completed. If the runtime of the function depends on the input data, the outgoing stream may be unordered. However, since the resulting tokens have the same indices as the incoming tokens, the outgoing stream is well-formed, provided that the incoming streams are well-formed.

Serial functions store the index of the previously processed tokens to ensure that subsequent tokens are processed in the correct order (as indicated in Figure 1, the stored index is initialized to zero). A serial function is thus only executed if the incoming tokens have the same indices $i$ and $j$ and if $j$ equals the stored index. After execution, the index of the previously processed token is set to $i$. It is easy to see that the outgoing stream is ordered, provided that the incoming streams are well-formed (if they are not well-formed, a serial function might block, since there might never be a token $\langle x, i, j\rangle$ such that $j$ is equal to the stored index).

The switch actor consumes two tokens: a control token carrying a Boolean value and a data token. Depending on the value of the control token, the data token is sent either to the true (left) or the false (right) output. Similar to serial functions, switches store the indices of the previously processed tokens. In Figure 1, these indices are denoted by $k$ and $l$ (both are initialized to zero). A switch actor is enabled if the indices of the incoming tokens match and the index $j$ of the previous token equals the maximum of the stored indices $k$ and $l$. After execution, the index of the token last sent to the true or false output is set to $i$. Again, the outgoing stream is ordered, provided that the incoming streams are well-formed. Note that switches must be executed in-order, since processing tokens out-of-order would require to know in advance which tokens will be sent to either output in order to determine the index of the previous token.

Finally, the select actor merges two streams which—in case of a conditional—may be generated by a switch actor. Depending on the Boolean value of the token at the control input, the select actor transfers a token from either the true or the false input to the output (no token is consumed from the other input). Note that the indices of the previous tokens need not match, since the control input and the data inputs refer to different streams. Hence, a selector is enabled if the incoming tokens have the same index $i$. Unlike switches, selectors may be executed out-of-order (the outgoing stream is well-formed if the control stream is well-formed).

So far, we only considered actors that both consume and produce tokens. In addition, we need sources that only produce tokens and sinks that only consume tokens. Sources can be implemented by means of serial functions without inputs, where the stored index serves as a counter that is incremented after each execution. Hence, a source produces a stream $S$ of tokens such that $S(1) \rightsquigarrow S(2) \rightsquigarrow S(3) \rightsquigarrow \ldots$ Sinks are simply serial or parallel functions without output. Thus, sinks may be executed out-of-order, whereas sources are always executed in-order.

As an example, Figure 2 shows the execution of a DPN using our approach. We assume that the switch as well as the selector each requires one time unit to execute, and that the functions $f$, $g$, and $h$ require 6, 1, and 4 time units to execute, respectively. Initially, there are the tokens $\langle x, 1, 0\rangle$ and $\langle y, 2, 1\rangle$ at the data input of the switch, and the tokens $\langle \mathsf{T}, 1, 0\rangle$ and $\langle \mathsf{F}, 2, 1\rangle$ at the control inputs of the switch and the selector. First, the tokens $\langle x, 1, 0\rangle$ and $\langle y, 2, 1\rangle$ are transferred to the true and the false branch of the conditional, respectively. At time $t = 2$, the functions $f$ and $g$ start to execute. Since $g$ only requires one unit of time to complete, the result $\langle g(y), 2, 0\rangle$ is available at time $t = 3$. It can be immediately processed out-of-order by the selector and need not wait until function $f$ is completed. Hence, at time $t = 4$ function $h$ starts to process the token $\langle g(y), 2, 1\rangle$, and at time $t = 8$ it starts to process the token $\langle f(x), 1, 0\rangle$. Execution ends at time $t = 12$ when both executions of $h$ are completed. Figure 3 shows a possible schedule on a dual-core processor, where hatched areas represent idle times. Conventional in-order execution of the DPN depicted in Figure 2 would require a total of 16 time units to complete (the amount of idle time would increase from 5 to 13 time units on a dual-core processor). For this example, we thus achieve a performance improvement of 25%.

## III. Implementation

In this section, we present an implementation of our approach that does not rely on locks to ensure mutual exclusion between concurrent threads. Lock-free algorithms have several advantages over conventional lock-based approaches [23]: Firstly, there is no need to perform a context switch by the operating system, which may significantly hurt performance, if a shared resource is used by another thread.

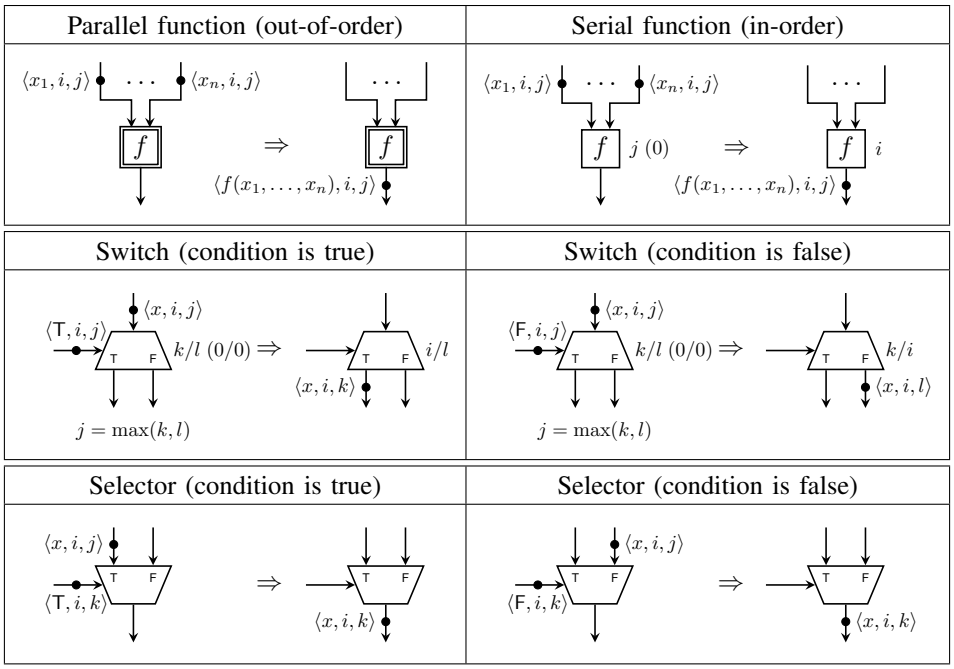



Figure 1. Transition rules for basic actors in dataflow process networks

Secondly, they have the potential to prevent synchronization pathologies such as deadlocks, convoying, and priority inversion (system-wide progress is guaranteed). Thirdly, lock-free algorithms can be efficiently executed using task schedulers for non-preemptive, lightweight tasks.

The algorithms given in the following can be implemented on shared-memory systems that support the compare-and-swap operation (CAS) [23], [24] and atomic incrementation of integers. They are given in a simple object-oriented language, where objects are always accessed through references. For the sake of brevity, we restrict ourselves to functions[3] and assume that the incoming streams are compatible (an error message should be emitted if tokens from incompatible streams are to be processed).

Figure 4 shows some basic classes that are used by the algorithms in Figures 5 and 6. The definition of the class Token follows directly from Def. 1, where $T$ is the type of the token value. The class Actor, which serves as base class for the actors of a DPN, contains a list of pairs that specify the successors of an actor (the first element is a reference to the actor, and the second element specifies the channel). If an actor wants to send a token to another actor, it calls the target actor's receive method, which stores the received token in a buffer according to the specified channel. As we will see later, token matching is performed via hash tables. For that purpose, the values of tokens having the same index are grouped into tuples and stored in objects of the class ParallelEntry (from which SerialEntry is derived). Additionally, the class ParallelEntry contains a counter that keeps track of the number of tokens received for a given index (the counter is initialized to zero). For efficiency reasons, we store in each object of the class SerialEntry the index of the successor tokens, if already available. Initially, it contains the value zero, which means that no successor token has yet arrived. If it is not equal to zero, the successor tokens can be processed right after the current invocation has finished execution (see below).

Let us now consider Figure 5, which shows the implementation of parallel functions. The hash map declared in line 2 stores for each index an associated object of the class ParallelEntry (lock-free implementations of hash maps are described, e.g., in [23], [25]). The method run, which must be defined in a subclass of ParallelFunction, executes the actual user-defined code. The main part is contained in the method receive: As the first step, the algorithm searches[4] the hash map for an entry with index idx (line 6) and stores the value of the received token in the associated tuple (line 7). Additionally, the counter is incremented by one (line 8). If the tuple is complete, i.e., all tokens with index idx have arrived, the user-defined code is executed (line 9) and the result is sent to the successors (lines 10–11). In order to run the successors in parallel, a new task is created for each successor using the **spawn** statement (execution of

[3]The implementation of switches, which are executed in-order, is similar to the implementation of serial functions, and the implementation of selectors, which may be executed out-of-order, is similar to the implementation of parallel functions.

[4]The call map.insert(idx, ParallelEntry()) inserts an empty entry into the hash map if no entry with index idx was found, and returns a reference to the new entry. Otherwise, a reference to the existing entry is returned.

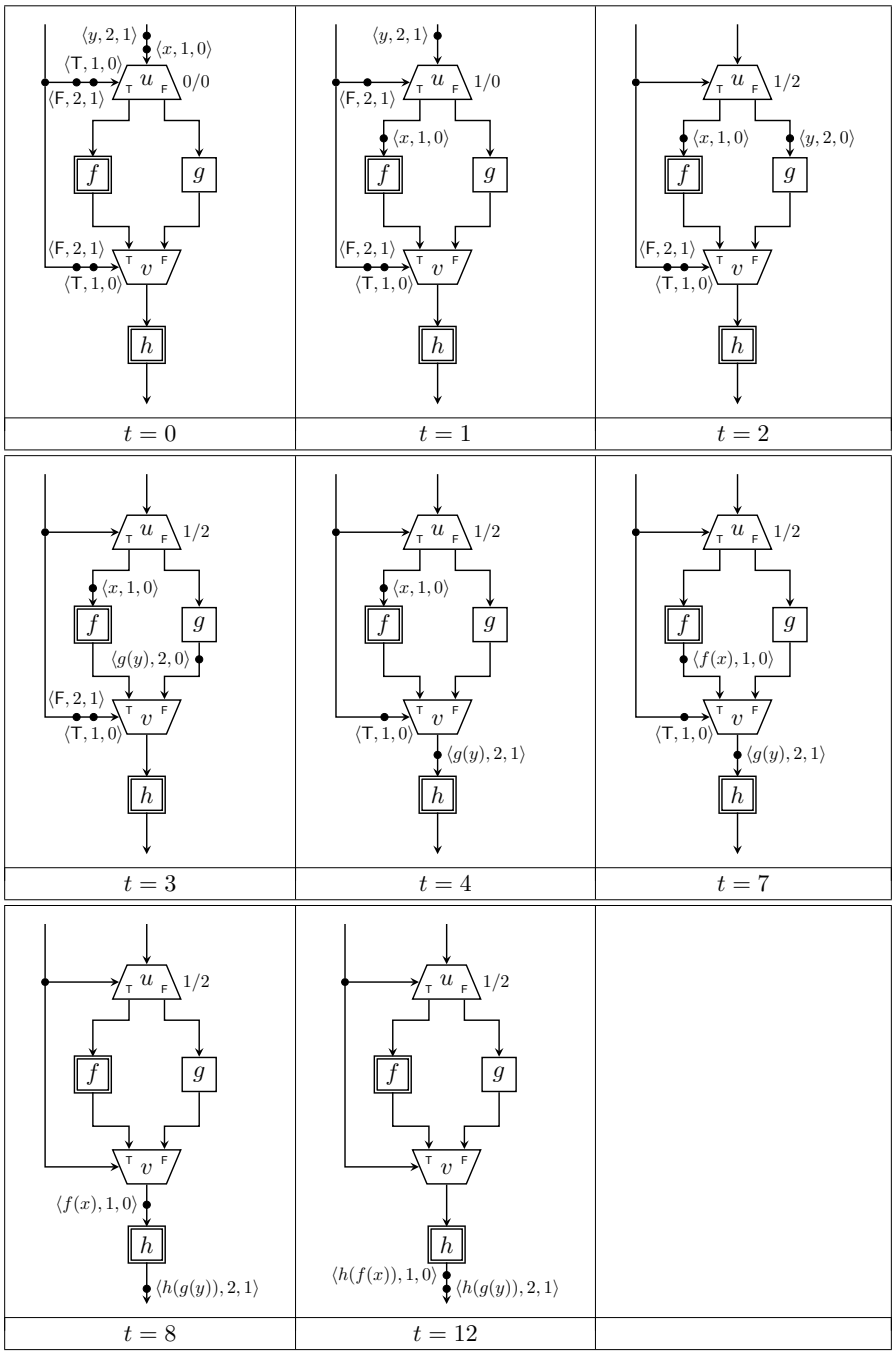



Figure 2. Out-of-order execution of a dataflow process network containing a conditional

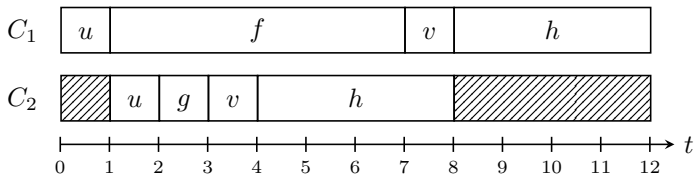



Figure 3. Schedule on a dual-core processor for the trace shown in Figure 2

```
class Token {
  T val;
  int idx, pidx;
}

class Actor {
  List⟨Actor, int⟩ successors;
  pure virtual void receive(Token token, int channel);
}

class ParallelEntry {
  Tuple⟨T_1, ..., T_n⟩ tuple;
  int counter(0);
}

class SerialEntry : ParallelEntry {
  int succ(0);
}
```

Figure 4. Basic classes for the algorithms in Figures 5 and 6

the current task continues immediately after a new task has been spawned). Finally, the entry with index idx is removed from the hash map (line 12). Note that multiple tasks may simultaneously call the method receive. For this reason, incrementation of the counter must be done atomically.

```
1  class ParallelFunction : Actor {
2    LockFreeHashMap⟨int, ParallelEntry⟩ map;
3    pure virtual R run(Tuple⟨T_1, ..., T_n⟩ tuple);
4    void receive(Token token, int channel) {
5      (idx, pidx) ← (token.idx, token.pidx);
6      entry ← map.insert(idx, ParallelEntry());
7      entry.tuple.set(channel, token.val);
8      if (entry.counter.atomic_inc() = n) {
9        res ← Token(run(entry.tuple), idx, pidx);
10       for each (actor, channel) in successors
11         spawn actor.receive(channel, res);
12       map.remove(idx);
13     }
14   }
15 }
```

Figure 5. Lock-free implementation of parallel functions

The implementation of serial functions is shown in Figure 6. As described in Section II, we have to store the index of the previously processed token (line 2) to ensure that all tokens are processed in the correct order. The method receive first updates the hash map (lines 7–8) and checks whether the tuple of token values is complete (line 9). If so, there are two cases to distinguish depending on whether the tokens arrived in-order or out-of-order:

If index equals the previous token's index pidx (line 11), the algorithm starts to process the tokens with index idx. As the first step, however, the entry of the previously processed tokens is removed from the hash map (line 13). In contrast to parallel functions, an entry cannot be removed immediately after a tuple has been processed, since another task may concurrently update the variable succ. After that, the user-defined code is executed (line 14) and the result is sent to the successor actors (lines 15–16). If the successor tuple is already complete (curr.succ $\neq 0$, line 17), the task updates the variables identifying the current context (lines 19-20) and starts over (line 12). Otherwise, curr.succ is set to $-1$ by the CAS operation indicating that the successor tuple will not be processed by the current task, and the variable index is updated to the current index.

In case index is not equal to the previous token's index pidx, the algorithm inserts a new entry for the previous token into the hash map, provided that it does not already exist (line 25). Then, it tries to update the variable succ using the CAS operation (line 26): If succ is zero, it is set to the current index and the task terminates (the tuple will be processed by another task once the predecessor tuple has been processed). Otherwise, the task starts over and retries to process the current tuple. It is easy to see that the outer loop (lines 10–29) is always aborted after a finite number of iterations: If succ is not zero in line 26, it must be $-1$. In this case, the inner loop (lines 12–21) will be aborted and index will be set to idx. Consequently, the condition of the **if** statement in line 11 will be satisfied in one of the following iterations of the outer loop, and the control flow will eventually reach the **break** statement in line 23. The number of iterations of the outer loop is thus bounded by the time that is required by the task processing the current token to get from line 18 to line 23.

## IV. Experimental Results

We implemented the presented algorithms in a C++ library that allows software developers to easily parallelize streaming applications. For efficiency reasons, most of the classes and functions are implemented using templates. This also facilitates type safety: connecting the output of an actor to an input of a different type results in a compile-time error. The library supports both in-place and out-of-place operations.[5] The library's memory management automatically allocates and deallocates space depending on the type of operation (in-place vs. out-of-place), the size of the token value, and the structure of the DPN.[6]

[5] An in-place operation modifies the incoming value, whereas an out-of-place operation writes the result to a new memory location. In image processing, for example, in-place operations are more efficient than out-of-place operations if only a subarea of an image is processed.

[6] As embedded systems often have tight restrictions on memory consumption, the maximum number of tokens in flight and the size of the hash tables can be set to a fixed value. Additionally, the token indices are mapped to integers with a finite bit width (overflows are handled explicitly).

```
1  class SerialFunction: Actor {
2    int index(0);
3    LockFreeHashMap⟨int, SerialEntry⟩ map;
4    pure virtual R run(Tuple⟨T_1, ..., T_n⟩ tuple);
5    void receive(Token token, int channel) {
6      (idx, pidx) ← (token.idx, token.pidx);
7      curr ← map.insert(idx, SerialEntry());
8      curr.tuple.set(channel, token.val);
9      if (curr.counter.atomic_inc() = n) {
10       while (true) {
11         if (index.load() = pidx) {
12           while (true) {
13             map.remove(pidx);
14             res ← Token(run(curr.tuple), idx, pidx);
15             for each (actor, channel) in successors
16               spawn actor.receive(channel, res);
17             if (curr.succ.compare_and_swap(0, –1))
18               break;
19             (idx, pidx) ← (curr.succ.load(), idx);
20             curr ← map.find(idx);
21           }
22           index.store(idx);
23           break;
24         } else {
25           pred ← map.insert(pidx, SerialEntry());
26           if (pred.succ.compare_and_swap(0, idx))
27             break;
28         }
29       }
30     }
31   }
32 }
```

Figure 6. Lock-free implementation of serial functions

To evaluate our approach, we have parallelized an image recognition application developed at Siemens for industrial automation. The application takes an image from a camera and performs a number of image processing steps until the considered object is recognized. The image processing steps range from simple local filters to complex operations such as the computation of connected components or image compression. Additionally, we parallelized the application using Intel's TBB. In order to obtain comparable results, we employed the task scheduler provided by TBB, which dynamically schedules tasks among a fixed number of worker threads created during initialization [4]. Moreover, we used the same kinds of operations (in-place vs. out-of-place). The experiments were performed on a system with two quad-core Xeon processors (2.83 GHz) and 6 GB RAM running Linux.

Table I shows the results. For each implementation, we measured the average throughput (in frames per second) and the maximum latency (in milliseconds) for processing 500 frames on one to eight cores. Moreover, we calculated the relative speedup with respect to the original sequential implementation.

Table I
EXPERIMENTAL RESULTS

| Cores | Throughput [f/s] | | Speedup | | Latency [ms] | |
|---|---|---|---|---|---|---|
| | TBB | DPN | TBB | DPN | TBB | DPN |
| 1 | 13.0 | 13.1 | 1.0 | 1.0 | 81 | 79 |
| 2 | 25.0 | 24.7 | 1.9 | 1.9 | 116 | 117 |
| 3 | 36.2 | 36.2 | 2.8 | 2.8 | 130 | 114 |
| 4 | 47.0 | 45.8 | 3.6 | 3.5 | 141 | 124 |
| 5 | 54.8 | 52.2 | 4.2 | 4.0 | 153 | 134 |
| 6 | 59.4 | 58.2 | 4.6 | 4.5 | 225 | 139 |
| 7 | 63.3 | 62.5 | 4.9 | 4.8 | 261 | 156 |
| 8 | 68.8 | 66.4 | 5.3 | 5.2 | 263 | 158 |

As can be seen from Table I, both implementations perform comparably well in terms of throughput. On four cores the speedup is 3.6 (TBB) and 3.5 (DPN), which is close to the theoretical maximum. On five to eight cores the speedup levels off slightly, which indicates that the sequential parts increasingly become a bottleneck. Interestingly, the overhead of the parallel implementations is negligible compared to the sequential one (speedup of 1.0 on a single core). Regarding the latency, our approach performs significantly better than TBB. On eight cores, the latency is reduced from 263ms to 158ms, which corresponds to a reduction of 40%. This is mainly because our approach supports nonlinear structures, whereas TBB is limited to linear pipelines.

## V. SUMMARY AND CONCLUSION

We presented a method and a lock-free implementation for combined in-order and out-of-order execution of dataflow process networks. In contrast to conventional pipeline-based approaches, which are frequently employed for the parallel execution of streaming applications, DPNs do not require linearization of nonlinear structures and support conditional execution. As another advantage, DPNs provide a deterministic programming model, which is often essential in the development of safety-critical embedded systems. Moreover, the developer is relieved from the burden of thread synchronization, since access to shared memory areas (communication channels) is hidden in the actors.

Using our approach, the externally visible behavior of systems that continuously interact with their environment is preserved, even though some tasks might be executed out-of-order. Out-of-order execution is important in the presence of conditionals, because the points of time at which the tokens arrive at the end of a conditional may vary significantly. In this way, idle times of the processor cores are cut down, which increases performance and reduces energy consumption. This is due to the fact that most frameworks for parallel

programming try to avoid time consuming context switches by the operating system in case a core is idle. As a result, idle cores are busy waiting and cannot be used by other threads (or put into power saving mode).

Our method does not maintain any global data structures that may result in a bottleneck and limit scalability. The transformation of token indices is accomplished in a distributed manner and causes little overhead, since only two integers are required to represent the indices. Moreover, conditionals can be nested without the need for additional data structures such as stacks that keep track of the nesting depth. Our experimental results indicate that the proposed implementation can compete with state-of-the-art libraries such as TBB in terms of throughput, while significantly reducing the latency.

To achieve a further reduction of the latency, we plan to develop a scheduler that supports task priorities. One of the problems with current task schedulers is that there is little control over the order in which ready tasks are executed. This may lead to undesirable behavior in stream-based applications. If, for example, a source actor is much faster than its successors, it may happen that the system is flooded with tokens entering the DPN, while other actors starve and hinder older tokens to leave the DPN. Using task priorities it can be guaranteed that the tokens in flight are processed before new ones enter the DPN.

## REFERENCES


[1] T. Mattson, B. Sanders, and B. Massingill, *Patterns for Parallel Programming*. Addison Wesley, 2005.

[2] J. Ortega-Arjona, *Patterns for Parallel Software Design*. Wiley, 2010.

[3] S. MacDonald, D. Szafron, and J. Schaeffer, "Rethinking the pipeline as object-oriented states with transformations," in *High-Level Parallel Programming Models and Supportive Environments (HIPS)*. Santa Fe, NM, USA: IEEE Computer Society, 2004, pp. 12–21.

[4] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly, 2007.

[5] E. Lee and T. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, May 1995.

[6] E. Lee, "The problem with threads," *IEEE Computer*, vol. 39, no. 5, pp. 33–42, May 2006.

[7] S. Sriram and S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*, 2nd ed. CRC Press, 2009.

[8] R. Stephens, "A survey of stream processing," *Acta Informatica*, vol. 34, no. 7, pp. 491–541, July 1997.

[9] F. Otto, V. Pankratius, and W. Tichy, "Xjava: Exploiting parallelism with object-oriented stream programming," in *European Conference on Parallel and Distributed Computing (Euro-Par)*, ser. LNCS, vol. 5704. Delft, The Netherlands: Springer, 2009, pp. 875–886.

[10] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *International Conference on Compiler Construction (CC)*, ser. LNCS, R. Horspool, Ed., vol. 2304. Grenoble, France: Springer, 2002, pp. 179–196.

[11] T. Schuele, "A coordination language for programming embedded multi-core systems," in *Parallel and Distributed Computing, Applications and Technologies (PDCAT)*. Hiroshima, Japan: IEEE Computer Society, 2009.

[12] M. Aldinucci, M. Torquati, and M. Meneghin, "FastFlow: Efficient parallel streaming applications on multi-core," Università di Pisa, Dipartimento di Informatica, Italy, Tech. Rep. TR-09-12, Sep. 2009.

[13] Arvind and K. Gostelow, "The U-interpreter," *IEEE Computer*, vol. 15, no. 2, pp. 42–49, 1982.

[14] Arvind and R. Nikhil, "Executing a program on the MIT tagged-token dataflow architecture," *IEEE Transactions on Computers*, vol. 39, no. 3, pp. 300–318, March 1990.

[15] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, July 1978.

[16] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 4th ed. Morgan Kaufmann, 2006.

[17] G. Kahn, "The semantics of a simple language for parallel programming," in *Information Processing*, J. Rosenfeld, Ed. Stockholm, Sweden: North Holland, 1974, pp. 471–475.

[18] B. Chapman, G. Jost, and R. van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, 2008.

[19] *OpenMP Application Program Interface (Ver. 3.0)*, OpenMP Architecture Review Board, 2008.

[20] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Symposium on Principles and Practice of Parallel Programming (PPoPP)*. Santa Barbara, CA, USA: ACM, 1995, pp. 207–216.

[21] A. Davis and R. Keller, "Data flow program graphs," *IEEE Computer*, vol. 15, no. 2, pp. 26–41, February 1982.

[22] J. Dennis, "First version of a data-flow procedure language," in *Programming Symposium*, ser. LNCS, B. Robinet, Ed., vol. 19. Paris, France: Springer, 1974, pp. 362–376.

[23] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Elsevier, 2008.

[24] M. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 1, pp. 124–149, January 1991.

[25] M. M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *Symposium on Parallel Algorithms and Architectures (SPAA)*. Winnipeg, Canada: ACM, 2002, pp. 73–82.