Expandable Process Networks to Efficiently Specify and Explore Task, Data, and Pipeline Parallelism

Lars Schor, Hoeseok Yang, Iuliana Bacivarov, and Lothar Thiele
Computer Engineering and Networks Laboratory
ETH Zurich, 8092 Zurich, Switzerland
firstname.lastname@tik.ee.ethz.ch

ABSTRACT

Running each application of a many-core system on an isolated (virtual) guest machine is a widely considered solution for performance and reliability issues. When a new application is started, the guest machine is assigned with an amount of computing resources that depends on the overall workload of the system and is not known to the designer at specification time. For instance, the computing resources might consist of many slow or a few fast processing elements. If the application is statically specified, as, for example, with Kahn process networks, the number of processing elements usable by an application is upper bounded by its number of processes. Similarly, the inter-process communication overhead might limit the maximum performance if the number of processing elements is significantly smaller than the number of processes. In this paper, we propose a formal extension for streaming programming models called expandable process networks (EPNs) that tackles this challenge by abstracting several possible granularities in a single specification. This enables the automatic exploration of task, data, and pipeline parallelism by two basic design transformation techniques, namely replication and unfolding. Then, the EPN semantics facilitates the synthesis of multiple design implementations that are all derived from one high-level specification. At runtime, the best fitting implementation for the given computing resources is selected to maximize the performance. Finally, we demonstrate the effectiveness of the proposed model on Intel's 48-core SCC processor.

1. INTRODUCTION

Current trends show that the next generation of multi/many-core systems will incorporate tens of cores on a single die [11, 26]. Such systems will provide a tremendous amount of processing power enabling multiple applications to share the same system at the same time. However, multiple applications sharing the same system can cause reliability or performance issues due to interference between applications. For example, the performance of one application might be affected by another application so that quality of service requirements cannot be met anymore. A widely considered solution is to separate these applications by running each application on an isolated (virtual) guest machine [7, 22].

Typically, a hypervisor is in charge to distribute the available computing resources at runtime. When a new application is started, the hypervisor assigns to the corresponding guest machine an amount of computing resources that depends on the overall system workload. For instance, the set of computing resources might contain many slow or a few fast processing elements. A drawback of this solution is that the system designer does not know anymore the available computing resources at specification time.

This challenge can be tackled by considering the parallelism provided by the application. However, if the application is statically specified, the maximum number of cores that can be utilized is limited to the number of parallel processes. On the other hand, having too many parallel processes might result in inefficient implementations of the application due to overheads in scheduling and inter-process communication. Thus, the optimal degree of parallelism for maximum performance often depends on the available computing resources. Another approach is to have multiple (different) specifications for the same application that are all synthesized. At runtime, the implementation leading to the best performance is selected depending on the available computing resources.

In this paper, we argue that for a certain class of applications, namely applications that are specified as process networks, the application can be specified in a manner that enables automatic exploration of task, data, and pipeline parallelism [9]. Task parallelism is achieved by executing different processes on different processing elements. In contrast, data parallelism refers to creating multiple instances of a process that perform the same task on different pieces of distributed data. Finally, pipeline parallelism is achieved by splitting a process into stages and assigning each stage to a different processing element.

We call the proposed model of computation (MoC) *expandable process networks (EPNs)*. EPNs extend conventional streaming programming models by abstracting several possible granularities in a single specification. This enables the automatic exploration of task, data, and pipeline parallelism by two design transformations, namely *replication* and *unfolding*. Replicating processes increases data parallelism and structural unfolding of a process increases the task and pipeline parallelism by hierarchically instantiating more processes in the process network. Furthermore, as recursive algorithms are commonly used in mathematical [1] and multimedia [2] applications, we study the recursive description of processes as a structural unfolding method.

For illustration purpose, we apply the proposed semantics of EPNs to Kahn process networks (KPNs) [12]. We will show that an EPN specification can be used to automatically synthesize multiple instances of the same application, each optimized for a different set of available computing resources. Experimental results on Intel's 48-core SCC processor show that having an abstract application specification outperforms a static specification when the available computing resources are not known at design time.

The contributions of this paper are summarized as follows:

- The proposed semantics of EPNs is formally described.
- We detail the concepts of *replication* and *unfolding*, and show that these two methods enable the automatic exploration of the application's task, data, and pipeline parallelism.
- The Y-chart design approach [13] is extended by the proposed semantics.
- An analytical performance model to analyze applications specified as EPNs is proposed and employed to explore different degrees of parallelism.

The remainder of this paper is organized as follows. We review related work in the next section. In Section 3, the proposed concepts are integrated into system design. In Section 4, the proposed EPN semantics is formally described. In Section 5, the concepts of replication and unfolding are detailed. Section 6 formulates the optimization problem for parallelizing and mapping EPNs. Finally, the results of the performed experiments are presented in Section 7.

2. RELATED WORK

Process networks and dataflow models are popular for specifying signal processing and streaming multimedia applications. Thus, in the last few years, various MoCs have been proposed that have different characteristics. Kahn process network (KPN) [12] programs, for instance, are determinate, provide asynchronous execution, and are capable to describe data-dependent behavior. The synchronous dataflow (SDF) model [16] restricts each process to produce and consume a fixed number of tokens in every iteration so that the application is amenable to compile-time scheduling techniques. Process networks might be automatically generated from sequential programs, e.g., the Compaan tool [14] transforms a nested loop program into a KPN.

As the SDF model has limited expressiveness, several extensions have been proposed that offer flexible and dynamic behavior and still preserve the capability to statically analyze the program. For instance, the synchronous piggybacked dataflow (SPDF) [19] enables $if-else$ and for clauses in SDFs. Hierarchical representations of process networks are allowed in some design frameworks, e.g., [3], but they are semantically equivalent to basic process networks, as they can be flattened at design time. However, none of these extensions brings any benefit in terms of parallelism, as they keep the process network topology unchanged. Dynamic representations such as reactive process networks [8] or scenario-based MoCs [24] offer the possibility to capture runtime topology changes by (de-)activating independent process networks. Even if the topology of active process networks is no longer static, the degree of parallelism is still statically determined at specification level.

Formal design and program transformation is considered to be an efficient approach to optimize an application towards the final architecture. A survey of existing transformation methods for functional programs is given, e.g., in [20]. The above discussed MoCs have in common that they specify the application in a high abstraction level, suitable for design transformation methods. However, due to the simplicity, most transformation methods focus on SDF graphs. In [17], successive iterations of an SDF graph are considered as a block enabling the concurrent execution of multiple instances of a single process. Our approach, in contrast, uses replication to concurrently execute multiple instances of a single process. In [21], a clustering technique for SDF graphs is proposed that first completely unfolds the graph, and then uses clustering techniques to reduce the number of actors per processing element to optimize the scheduling. On the other hand, our technique only unfolds a graph if a performance gain is achieved by the additional parallelism. Moreover, various transformation rules to balance a network are presented in the context of the ForSyDe methodology [23], a synchronous computational model. In contrast, our technique is based on KPNs and by applying the proposed semantics to individual processes, a finer granularity is achieved.

Another SDF transformation technique is proposed in [9] in the context of the StreamIt language [28]. It uses so-called fusion and fission operators at the mapping stage to incorporate architecture-dependent profiling information. For efficient data parallel execution, StreamIt coarsens the SDF granularity to increase the computation to communication ratio by *fusion*, while *fission* is used to distribute data parallel tasks to multiple cores. In contrast to our approach, StreamIt transformations are restricted to data parallelism and structural expansions are not considered. Therefore, the maximum degree of task and pipeline parallelism is still upper bounded by the number of processes.

Recursion is the procedure that repeatedly calls itself, and is typically used in programming to divide a problem into multiple subproblems with the same (repeated) behavior. In case that a huge amount of independent data needs to be processed in a similar manner, recursive implementations are of practical use due to their simplicity. There is a wide range of mathematical algorithms that can be implemented recursively [6] and even multimedia applications, such as ray-tracing [2], can be specified as a recursive algorithm. In summary, conventional process networks or dataflow models are too static and monolithic to explore different application structures at design time. Moreover, they cannot represent recursive dependencies necessary to effectively describe certain classes of applications. The proposed EPN semantics extends conventional MoCs and enables the exploration of an efficient application structure by exploiting task, data, and pipeline parallelism.

3. SYSTEM DESIGN

To include the proposed concepts in the system design, we extend the current Y-chart design practice [13] with an additional design step called *design transformation*, see Fig. 1.

The inputs to the system design flow are an application specified as EPN and a high-level specification of the target architecture. In a first step, possible sub-architectures are identified. A sub-architecture contains only a subset of all processing elements of the target architecture. It is used to represent the fact that a hypervisor might not assign all available processing elements to an application, but only a subset of them. The set of sub-architectures generated for a given architecture should cover the variety of processing element subsets that a hypervisor can assign to an application. For a homogeneous platform as Intel's SCC proces-

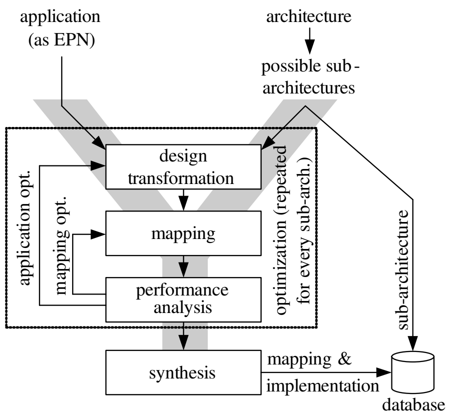


Figure 1: Modified Y-chart design flow for applications specified as EPNs.

sor [11], which has 48 identical cores, each sub-architecture might just differ in the number of processing cores. In case that the target architecture is heterogeneous, the number of possible sub-architectures might be larger. However, the number can be reduced by considering the symmetry of the architecture, or only selecting a subset of all possible sub-architectures and ignoring sub-architectures with similar computing power. Even more, the set of valid sub-architectures might be reduced if the application has specific performance requirements that can only be met if the sub-architecture provides a minimum amount of computing power.

The application and the mapping of the application is optimized separately for each sub-architecture, aiming to maximize the throughput of the application. The optimization stage consists of the design transformation, the mapping, and the performance analysis. In the design transformation step, *replication* and *unfolding* are used to explore the parallelism. We will detail these concepts later in Section 5. Afterwards, a mapping is calculated and the throughput of the application is evaluated. The design transformation and mapping optimization steps are repeated until a degree of parallelism is found that fulfills the performance requirements of the system. Based on the information obtained in the optimization steps, a concrete (replicated and unfolded) implementation of the application is generated during synthesis. Finally, the description of the sub-architecture is stored in a database together with the implementation and the respective mapping information. Later, at runtime, the hypervisor uses the database to select an optimized implementation based on the available computing resources.

4. EXPANDABLE PROCESS NETWORKS

In this section, the semantics of expandable process networks (EPNs) is formally described. EPNs extend conventional streaming programming models by abstracting several possible granularities in a single specification.

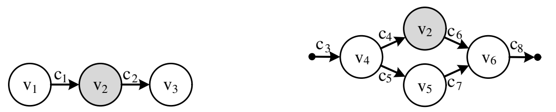
The proposed semantics is applied to the KPN [12] MoC. A KPN consists of autonomous processes that can communicate through unbounded point-to-point FIFO channels. In the KPN model, a process is a monotonic and determinate mapping F of one or more input streams to one or more output streams. Furthermore, a process might have an internal state that affects its behavior.

Formally, a process network $p = \langle V, C \rangle$ consists of a set of processes V and a set of channels $C \subseteq V \times V$. Each process $v \in V$ is characterized by a tuple $\langle name, type, replicable, in, out \rangle$, where $name$ is a unique string identifier, $type \in \{behav, struct\}$ describes the type of the instantiated process, *replicable* $\in \{false, true\}$ indicates if a process can be replicated, and $in/out \subseteq C$ denotes the set of incoming/outgoing channels of process v. A channel $c = \langle v_1, v_2 \rangle \in C$ represents a directed communication from process v_1 to v_2. Except the elements *type* and *replicable*, the proposed process network description is identical to an ordinary specification of a KPN. The *type* identifier reveals that some processes do not only have a behavioral, but also a structural description. The type *struct* specifies that the process has a behavioral as well as a structural description, while the type *behav* specifies that the process has only a behavioral description.

A behavioral description specifies the functionality of a process in a high-level programming language. A structural description defines the functionality as another process network, i.e., as a set of processes and channels. Both the behavioral and structural description have to be functionally equivalent in the sense that, for a given sequence of input tokens, they produce the same sequence of output tokens. In case the functionality of a process is only given by either a behavioral or a structural description, one might obtain the other description by a transformation. For instance, a behavioral description can be obtained from a structural one by implementing the channels as shared variables. Conversely, one might use the techniques described in [14, 29] to obtain a structural description out of a behavioral one.

An EPN is defined as a tuple $e = \langle P, u, l, p_{org} \rangle$, where P is a set of process networks, u and l are transformation functions, and p_{org} is the top-level process network from which processes may be further replicated or structurally unfolded. The top-level process network is the most coarsened process network abstraction of the application; it might even consist of only one process. In order to specify functions u and l, we define the set of all processes of the EPN e as $V^e = \bigcup_{p \in P} V$ and the set of all channels of the EPN e as $C^e = \bigcup_{p \in P} C$. Function $u : V^e \to P$ maps a process v to a corresponding process network $p = u(v)$. Thus, $u(v)$ represents the structural specification of process v. Function $l : C^e \to C^e$ maps a channel c to a corresponding channel $l(c)$ representing the match between the input and output channels of a process v and the input and output channels of the structural specification $u(v)$ of v.

Example 1. Consider the example specification shown in Fig. 2. The EPN $e = \langle \{p_{org}, p_{v_2}\}, l, u, p_{org} \rangle$ has the top-level process network $p_{org} = \langle \{v_1, v_2, v_3\}, \{c_1, c_2\} \rangle$. $v_1 = \langle `v_1\text{'}, behav, false, \emptyset, \{c_1\} \rangle$ and $v_3 = \langle `v_3\text{'}, behav, false, \{c_2\}, \emptyset \rangle$ are ordinary processes which have no further unfolding capabilities. $v_2 = \langle `v_2\text{'}, struct, true, \{c_1\}, \{c_2\} \rangle$ is a process of type *struct*, which has both a behavioral and a structural description. The structural description of v_2 is another process network $p_{v_2} = \langle \{v_2, v_4, v_5, v_6\}, \{c_3, c_4, c_5, c_6, c_7, c_8\} \rangle$. Note that process v_2 appears in both p_{org} and p_{v_2} enabling recursive unfolding. As v_2 is the only process of type *struct*, u returns null for all inputs except v_2, i.e., $u(v_2) = p_{v_2}$. Similarly, function l returns null for all inputs except the input and output channels of v_2. However, as v_2 occurs in both p_{org} and p_{v_2}, function l contains four assignments, namely $l(c_1) = c_3$, $l(c_2) = c_8$, $l(c_4) = c_3$, and $l(c_6) = c_8$.



(a) Top-level process network p_{org}.

(b) Structural description of process v_2.

Figure 2: EPN specification example.

Algorithm 1 Replicate $v \in V$ in $p = \langle V, C\rangle$ χ times.

```
$V \leftarrow V - \{v\}$                                   ▷ remove process $v$
for $i = 1 \rightarrow \chi$ do                            ▷ generate a replicated process
    $v_i \leftarrow \langle name^{\{i\}}, type, false, \emptyset, \emptyset\rangle$
    $V \leftarrow V \cup \{v_i\}$
end for
                                                           ▷ for each incoming channel
for all $c = \langle v_{src}, v_{dst}\rangle$ s.t. $v_{dst} = v$ and $v_{src} <> v$ do
    $C \leftarrow C - \{c\}$                               ▷ remove channel $c$
    for $i = 1 \rightarrow \chi$ do
        $c_i \leftarrow \langle v_{src}, v_i\rangle$       ▷ add a replicated channel
        $C \leftarrow C \cup \{c_i\}$
        $in$ of $v_i \leftarrow in$ of $v_i \cup \{c_i\}$
    end for
end for
                                                           ▷ for each outgoing channel
for all $c = \langle v_{src}, v_{dst}\rangle$ s.t. $v_{src} = v$ and $v_{dst} <> v$ do
    replace $c$ with the set of replicated channels
end for
                                                           ▷ for each self-loop channel
for all $c = \langle v_{src}, v_{dst}\rangle$ s.t. $v_{src} = v$ and $v_{dst} = v$ do
    $C \leftarrow C - \{c\}$                               ▷ remove channel $c$
    for $i = 1 \rightarrow \chi$ do
        $c_i \leftarrow \langle v_i, v_{\{(i+1) \bmod \chi\}}\rangle$   ▷ circular connection
        $C \leftarrow C \cup \{c_i\}$
        $out$ of $v_i \leftarrow out$ of $v_i \cup \{c_i\}$
        $in$ of $v_{\{(i+1) \bmod \chi\}} \leftarrow in$ of $v_{\{(i+1) \bmod \chi\}} \cup \{c_i\}$
    end for
end for
```

5. EXPLOITING TASK, DATA, AND PIPELINE PARALLELISM

EPNs enable an efficient architecture independent specification of process networks. However, as shown in Section 3, an additional design step is required in the Y-chart design practice to include the concepts into the system design. In the following, we will detail the design transformation step that, by applying the *replication* and *unfolding* concepts, exploits the different kinds of parallelism and transforms the EPN $e = \langle P, u, l, p_{org}\rangle$ into the KPN $p_a = \langle V_a^e, C_a^e\rangle$. The section continues with the concepts of *replication* and *unfolding*. Afterwards, we show that the correctness of the top-level process network is not affected by the proposed transformation methods. Finally, we propose a high-level application programming interface (API) for EPNs.

5.1 Replication

Handling parallelism inside a process is typically difficult as a process is mapped as a whole onto a single processing unit. Artificially parallelizing the process later using conventional parallel processing APIs [4, 18] is undesirable as the implicit parallelism makes design-time analysis impossible. Exposing this information at the process network level is more beneficial as it results in higher predictability and better mapping decisions.

In the EPN semantics, the step of handling parallelism inside a process is called *replication*. Replication is particularly applicable to algorithms that have a high data level parallelism, as it is often the case with algorithms optimized for single instruction, multiple data (SIMD) processors. Typical examples are deinterlacing algorithms used to convert interlaced videos, image noise reduction algorithms, or video (de)compression algorithms. Consider, for example, a video decoder that uses intraframe-compression. As there is no relation between the frames, multiple frames might be processed in parallel on different processing elements. Furthermore, various video compression algorithms support the segmentation of a frame into macroblocks and these macroblocks can individually be processed.

The concept of replication has already been widely used to improve the performance of process networks [9, 27]. Typically, the bottleneck process has been replicated to improve the overall performance. However, in contrast to all these concepts, we do not statically define the number of replications at specification level, but argue that it is the task of the optimizer to find a good degree of parallelism that maximizes the performance.

Algorithm 1 illustrates the steps to modify the topology of process network $p = \langle V, C\rangle$ so that process $v = \langle name, type, replicable, in, out\rangle$ is χ times replicated. First, it removes process v and adds replicated clones, where $v_i = \langle name^{\{i\}}, type, false, \emptyset, \emptyset\rangle$. Then, each incoming channel $c = \langle v_{src}, v\rangle$ of v is replaced by a set of replicated channels with $c_i = \langle v_{src}, v_i\rangle$. Similarly, all outgoing channels of v are also replaced by a set of replicated channels. Self-loop channels, i.e., channels that connect v with itself, are handled last. For each self-loop channel, a new chain of channels is introduced with one channel connecting v_i with v_{i+1}. As such a chain of channels introduces a circular dependency between the processes, it typically limits the maximum speed-up that can be obtained by replication. For instance, no speed-up can be achieved if v is reading from the self-loop channel at the beginning of the iteration and writing to it at the end of the iteration. In all other situations, the replicated copies of the process can still partly overlap their execution so that the system will achieve a speed-up higher than one.

Replicating processes with an internal state is supported by adding a self-loop channel representing the state of the process. Replicating two consecutive processes in a row is not allowed to prohibit complex communication behavior. If consecutive replication is needed for optimized performance, two consecutive processes should be specified as a structural process, and then replicated together.

5.2 Unfolding

The EPN specification abstracts several possible granularities in a single specification. The step of exploring different task and pipeline parallelisms by hierarchically instantiating more processes is called *unfolding* and is explained next.

Given an application specified as EPN, a process of type *struct* can be *unfolded* by exposing internal parallelism at process network level. In other words, unfolding merely replaces the behavioral description of process v with its structural description $u(v)$. In addition, unfolding enables recursion as the structural representation $u(v)$ of process v may have v as an internal process. In contrast to all previously proposed models, the maximum number of tasks is

Algorithm 2 Unfold $v \in V$ in $p = \langle V, C \rangle$ with $p_v = \langle V_v, C_v \rangle$.

```
1:  for all $v_i \in V_v$ do                        ▷prefix $v$ to all names of $v_i \in V_v$
2:      $v_i.name \leftarrow v.name + v_i.name$
3:  end for
                                                  ▷prefix $v$ to all names of $c_i \in C_v$
4:  for all $c_i = \langle src_i, dst_i \rangle \in C_v$ do
5:      $src_i \leftarrow v.name + src_i$
6:      $dst_i \leftarrow v.name + dst_i$
7:  end for
8:  $V \leftarrow (V - \{v\}) \cup V_v$              ▷remove $v$, add unfolded processes
9:  $C \leftarrow C \cup C_v$                       ▷add unfolded channels
                                                  ▷for each incoming channel
10: for all $c = \langle src, dst \rangle$ s.t. $dst = v$ do
11:     for all $c_i = \langle src_i, dst_i \rangle \in C_v$ do
12:         if $l(c) = c_i$ then                    ▷find a match
13:             $src_i \leftarrow c.src$            ▷adjust src of $c_i$
14:             $C = C - \{c\}$                     ▷remove the unused channel
15:             break
16:         end if
17:     end for
18: end for
                                                  ▷for each outgoing channel
19: for all $c = \langle src, dst \rangle$ s.t. $src = v$ do
20:     remove all outgoing channels of $v$
21: end for
```

not statically determined. General instructions to unfold a process $v \in V$ in process network $p = \langle V, C \rangle$ with network p_v are given in Algorithm 2. The algorithm first prefixes the name of process v to all unfolded processes and channels to keep them unique after design transformation. Afterwards, it removes process v and adds the cloned copy of the unfolded network p_v. Finally, it replaces each incoming and outgoing channel of v by the corresponding match in C_v.

Example 2. Consider the EPN outlined in Fig. 2. v_2 is of type *struct*, which means that it has a behavioral and a structural description. By recursively unfolding v_2, it will repetitively be replaced by p_{v_2}. Figure 3 illustrates the process network if process v_2 has been unfolded twice.

Typically, the input of a process network limits the amount of times that a process can recursively be instantiated. For instance, if recursively unfolding means that an array is split into two smaller arrays, the maximum recursion depth is defined by the length of the input array. In order to avoid deadlocks, the designer has either to specify a termination condition for recursively unfolding (e.g., by knowing the minimum length of the array in the previous example) or to guarantee that the application is not blocked if the input prohibits further recursion. The later might be achieved by forwarding either the result or a predefined 'empty' string. If the maximum recursion depth is known, the unfolding method can use this information to avoid blocking.

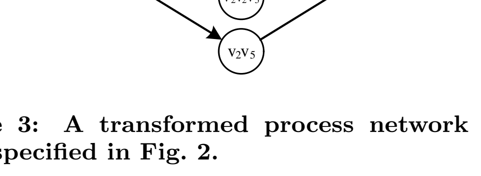


Figure 3: A transformed process network of the EPN specified in Fig. 2.

In summary, starting with the top-level process network, which is the most coarsened process network abstraction, all possible abstractions can be explored by applying the concepts of *replication* and *unfolding*.

5.3 Correctness

We will show that the proposed transformation methods preserve the correctness of the underlying programming model if certain conditions, which we will define later, are fulfilled. Formally, the design transformation step transforms the EPN $e = \langle P, u, l, p_{org} \rangle$ into the KPN $p_a = \langle V_a^e, C_a^e \rangle$ with V_a^e the set of processes of p_a and C_a^e the corresponding set of channels. A transformation method preserves correctness if, for a given input sequence, the transformed process network p_a produces the same output sequence as the top-level process network p_{org}. Clearly, a first requirement for the correctness is that each process network $p \in P$ is a valid KPN in the sense that it does not cause deadlocks.

We suppose to have the top-level network $p_{org} = \langle \{v_1, v_2, v_3\}, \{c_1, c_2\} \rangle$ shown in Fig. 2a. To show that *replication* preserves correctness, process $v_2 = \langle v_2, behav, true, \{c_1\}, \{c_2\} \rangle$ is replicated during design transformation. Thus, channel c_1 is replaced by a set of channels $c_1^{\{x\}}$, and channel c_2 is replaced by a set of channels $c_2^{\{x\}}$. If v_3 reads the incoming channels $c_2^{\{x\}}$ in the same order as v_1 is writing to the channels $c_1^{\{x\}}$, the concatenation of all incoming tokens of v_3 is the same for the transformed process network p_a and for the top-level process network p_{org}. Thus, *replication* preserves the correctness under the described condition.

The correctness of *unfolding* is shown by considering EPN $e = \langle P, u, l, p_{org} \rangle$. We first suppose that function l contains all possible channel matches and that all structural extensions defined by function u preserve the input/output relation. In other words, suppose that function u defines the structural extension $u(v) = p$ of process v with $p \in P$. Then, process v has the same amount of input and output channels as process network p and applying the same input sequence to v and p produces the same output sequence. As unfolding merely replaces certain processes v with their structural description $u(v)$, the transformed process network has still the same input/output behavior.

5.4 Application Specification

After describing the basic concepts of EPNs and defining the conditions when *replication* and *unfolding* preserve the correctness, we are able to come up with a high-level API to specify EPNs. A high-level API for process networks consists of two elements, namely the specification of the functionality of the individual processes and the topology of the network. In the following, we will discuss how an API for KPNs can be extended to support the semantics of EPNs.

The topology specification of an EPN is composed of the topology of multiple process networks, each specified as a KPN. The process element is extended with the attributes *type* and *replicable* as defined in Section 4. In addition to the specification of the networks, the transformation functions u and l have to be defined within the topology specification.

We suppose that the functionality of the individual processes is described in separate description files written in a high-level language like C/C++. As the structural extension of a process with a process network does not change the external interface, i.e., the incoming and outgoing channels,

Listing 1: Pseudo code of a process sending data to a replicated channel to illustrate the proposed high-level API. The fire method specifies the functionality of a process and is executed once per iteration.

```
int FIRE(ProcessState state)
    ...
    String channel_basename = "c1";
    channel = CREATECHANNEL(channel_basename,
          state->channelCounter);
    WRITE(channel, buffer, size);
    state->channelCounter = (state->
          channelCounter + 1) % REPLICATIONS_C1;
    ...
}
```

the structural extension is completely transparent towards the functionality of the other processes. However, in case of replication, the actual processes and channels are not known at specification time. Thus, the source and sink processes of a replicated process have to write/read data to/from a channel that is not known at specification time. In this case, we propose the API outlined in Listing 1 to iterate over all possible channels. The basic idea is that, per iteration, the source/sink process still writes/reads to/from one instance of the replicated process. We propose that the channels are addressed by their basename, i.e., the name of the channel before replicating, and a counter, which is stored in the state of the process. The number of replications per channel can be obtained from a global variable that is automatically set during synthesis. In Listing 1, the synthesizer sets the variable `REPLICATIONS_C1` to the number of replications of channel "c1". Practically, the code shown in Listing 1 might be created by an automated source-to-source code transformation during software synthesis.

Example 3. Consider again the process network outlined in Fig. 2a, but now, process v_2 is replicated resulting in the process network shown in Fig. 4. At specification time, process v_1 writes to channel c_1, which does not anymore exist in the transformed process network. Instead, v_1 has three output channels $c_1^{\{1\}}$, $c_1^{\{2\}}$, and $c_1^{\{3\}}$. The API outlined in Listing 1 hides the transformation details from the programmer that can still use channel c_1.

6. APPLICATION AND MAPPING OPTIMIZATION

In this section, we introduce a novel performance analysis model for applications specified as EPNs and show how the performance model can be used to optimize the parallelism and the mapping. The aim of the optimization is to identify which abstraction of the EPN leads to the highest throughput on a given (sub-)architecture. To this end, we minimize

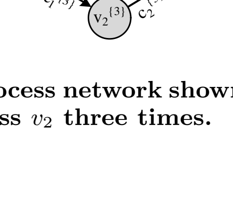


Figure 4: The process network shown in Fig. 2a after replicating process v_2 three times.

Table 1: List of parameters that are used in the performance model (with process network p, process v, and core d).

Param.	Description
$\Gamma(v,d)$	assignment function (1 if and only if v is mapped onto d, otherwise 0)
t_d^0	cycle time on d
n_v^{co}	number of computation cycles of v
$n_{i,\lambda}^{re}$ / $n_{o,\lambda}^{wr}$	number of cycles to read/write from/to channel i/o for communication mode λ
n_v	total number of cycles per iteration of v
$f_{v,p}^{rel}$	relative execution rate of v in p
f_v^{abs}	absolute execution rate of v
η_i^{re} / η_o^{wr}	number of readings/writings from/to channel i/o per iteration
γ_d	context overhead per time instance on d
T_d^{cont}	context switch time on d
ν_v^{co}	relative number of computation cycles of v when it is recursively unfolded
ν_i^{re} / ν_o^{wr}	relative number of cycles that v has to read/write from/to channel i/o when v is recursively unfolded

the maximum core utilization for a given invocation interval of the source process. The invocation interval is then adjusted so that the maximum core utilization becomes 100 %. Finally, the new invocation interval is used to calculate the maximum throughput of the EPN.

Note that the performance analysis model is not restricted to the maximization of the throughput, but can also be applied to optimize other performance metrics as, for instance, the energy consumption. In this case, the invocation interval might be fixed and the utilization of the individual cores is used to calculate the average expected energy consumption of the system.

6.1 Preliminaries and Notation

A sub-architecture consists of a set of cores D that are connected by a communication network, e.g., a bus or a network-on-chip. A core $d \in D$ is characterized by the cycle time t_d^0.

Suppose that the EPN $e = \langle P, u, l, p_{org} \rangle$ is transformed into the KPN $p_a = \langle V_a^e, C_a^e \rangle$. The mapping of p_a is defined by the assignment function $\Gamma(v,d) \in \{0,1\}$ that is 1 if and only if process $v \in V_a^e$ is mapped onto core d. In order to guarantee that each task is assigned to exactly one core, the following equation has to be fulfilled for all processes $v \in V_a^e$:

$$\sum_{\ell=1}^{|D|} \Gamma(v,d) = 1 \quad \forall v \in V_a^e. \tag{1}$$

Table 1 summaries the most important parameters that are used in the following.

6.2 Performance Model

Next, we describe a novel performance model that is used in the design space exploration to analyze a candidate network $p_a = \langle V_a^e, C_a^e \rangle$. The performance model aims to provide a metric for the average utilization of a core so that the maximum average core utilization can be minimized.

The iterative execution of a behavioral process v is characterized (per iteration) by a number of computation cycles n_v^{co}, a number of read cycles $n_{i,\lambda}^{re}$ per input channel $i \in in$, and a number of write cycles $n_{o,\lambda}^{wr}$ per output channel $o \in out$. The number of read and write cycles depends on the data-volume, thus the (average) size of the packets and the number of packets that are read or written per iteration. The factor λ indicates the dependencies of the read and write instructions on the location of channel i or o. For simplicity, we just differ between inter-core and intra-core communication, thus $\lambda \in \{\textit{inter-core}, \textit{intra-core}\}$. Note that the concept can be extended to more complex communication topologies, e.g., by differentiating between the number of hops that a packet experiences. The average number of cycles that a process v performs in one iteration is therefore:

$$n_v = n_v^{co} + \sum_{i \in in} n_{i,\lambda}^{re} + \sum_{o \in out} n_{o,\lambda}^{wr}. \tag{2}$$

The average utilization of a core also depends on the average execution rates of all processes $v \in V_a^e$. First, we note that the absolute execution rate of a process v cannot be specified in advance as it depends on the execution rate of its parent processes. However, the later might be known only after the design transformation. Second, we note that a process can occur in multiple process networks and that its execution rate might be different for every process network p. Thus, at specification time, we characterize a process v by a set of relative execution rates $f_{v,p}^{rel}$ with one execution rate per process network p that v can occur in. In addition, a single process $\tilde{v}$ of the top-level process network p_{org} is characterized by an absolute execution rate $f_{\tilde{v}}^{abs}$. This enables us to calculate the absolute execution rates f_v^{abs} of all processes $v \in V_a^e$ after the design transformation step. Later, we use these absolute execution rates to calculate the average utilization of a core. The absolute execution rates can be calculated in a top-down approach following the performed design transformations. The algorithm starts with process $\tilde{v}$ and executes the following rules for any process v:

1. If v belongs to p_{org}, then $f_v^{abs} = f_{v,p}^{rel} \cdot f_{\tilde{v}}^{abs}$.
2. If v belongs to $p \in \{P \setminus p_{org}\}$ and replaces process $\hat{v}$, then $f_v^{abs} = f_{v,p}^{rel} \cdot f_{\hat{v}}^{abs}$.
3. If v is instantiated by replicating $\hat{v}$ χ times, then $f_v^{abs} = \frac{1}{\chi} \cdot f_{\hat{v}}^{abs}$.

If multiple processes share the same resource, the synchronization and scheduling overhead might affect the overall performance. In this paper, we differ between event-triggered and time-triggered scheduling policies. Time-triggered scheduling policies typical cause a constant overhead, while the overhead caused by an event-triggered scheduling policy depends on the workload. Suppose that multiple processes $v = \langle \textit{name}, \textit{type}, \textit{replicable}, \textit{in}, \textit{out} \rangle$ share the same core. A process can become blocked when reading from an empty input FIFO $i \in in$ or writing to a full output FIFO $o \in out$. A pessimistic assumption for a non-preemptive scheduling policy is that the process is always blocked when reading/writing from/to a FIFO channel. Thus, the total average context overhead per time instance on core d is given by:

$$\gamma_d = \left(\sum_{v \in V_a^e} \Gamma(v,d) \cdot f_v^{abs} \cdot \left(\sum_{i \in in} \eta_i^{re} + \sum_{o \in out} \eta_o^{wr} \right) \right) \cdot T_d^{cont}, \tag{3}$$

where T_d^{cont} is the context switch time on core d, η_i^{re} is the average number of readings per iteration from channel i, and η_o^{wr} is the average number of writings per iteration to channel o. Clearly, if only one process is mapped onto core d, there is no context overhead and $\gamma_d = 0$.

If a process v is recursively instantiated, the execution time might be reduced with every recursion step. In order to model this reduction in the performance model, a recursive process v is also annotated by a relative number of computation cycles ν_v^{co}, a relative number of read cycles ν_i^{re} per input channel $i \in in$, and a relative number of write cycles ν_o^{wr} per output channel $o \in out$. Suppose, for instance, that v has to perform n_v^{co} computation cycles (when belonging to p_{org}). If v is recursively unfolded, its number of computation cycles is reduced to $\nu^v \cdot n_v^{co}$ in every recursion step. Similarly, if the execution time does not change, $\nu^v = 1$.

6.3 Optimization Problem

The goal of the optimization step is to find a set of design transformations and a mapping that minimize the maximum core utilization. The objective function can formally be stated as:

$$\min \left\{ \max_{d \in D} \left\{ \gamma_d + \sum_{v \in V_a^e} \Gamma(v,d) \cdot f_a^{v,p_a} \cdot n_v \cdot t_d^0 \right\} \right\}, \tag{4}$$

where γ_d is defined as in (3), $\Gamma(v,d)$ has to fulfill the constraint specified by (1), and $p_a = \langle V_a^e, C_a^e \rangle$ is a valid design transformation of EPN e.

Thus, the overall optimization problem involving application and mapping exploration can be stated as:

> *Given an EPN e that is mapped onto a sub-architecture with a set of cores D. Then, the goal is to find a KPN $p_a = \langle V_a^e, C_a^e \rangle$ that is a valid design transformation of e and an assignment function $\Gamma(v,d)$ with $v \in V_a^e$ and $d \in D$ that minimize the maximum average core utilization as stated in (4) and fulfills the constraint specified by (1).*

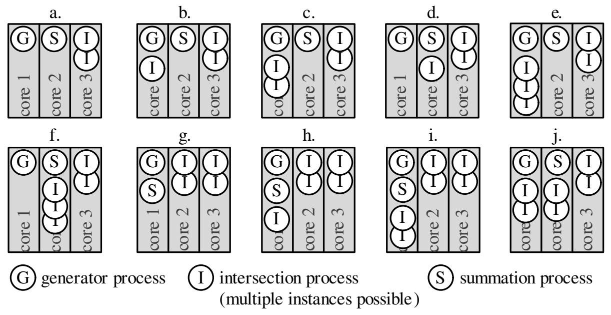
The EPN semantics can be applied to a wide variety of optimization techniques. Optimizing an application specified as an EPN for a given (sub-)architecture involves two steps. First, the transformation methods proposed in the last section are applied to obtain a KPN $p_a = \langle V_a^e, C_a^e \rangle$. Afterwards, the mapping of a candidate p_a is optimized. In order to reduce the time for design space exploration, the two steps can be combined into one optimization problem and the design transformations can be considered as an additional degree of freedom when optimizing the mapping.

7. EXPERIMENTAL RESULTS

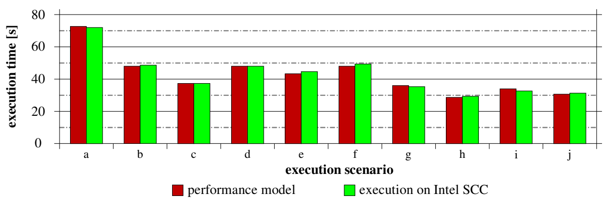
In this section, we present case studies demonstrating the effectiveness of the EPN semantics targeting Intel's SCC processor. We consider three benchmarks, namely a ray-tracing application, a video decoder, and an array sorting application. As applications are running in isolated guest machines, the effectiveness can be studied individually for each application.

7.1 Experimental Setup

In order to test the effectiveness of the EPN semantics, we implemented a concrete runtime-system and software



(a) Mapping and replication scenarios. "G" refers to the "generator" process, "S" to the "summation" process, and "I" to an instance of the "intersection" process.



(b) Execution time calculated with the performance analysis model described in Section 6 vs. the measured execution time on Intel's SCC processor.

Figure 5: Execution time of the ray-tracing application outlined in Fig. 6 for different mapping and replication scenarios. The target platform are three cores of Intel's SCC processor.

synthesis tool chain for applications specified either as an EPN or a KPN targeting Intel's SCC processor [11]. The SCC processor is an experimental prototype of an on-chip many-core system with 24 tiles each containing a pair of P54C cores. Each core is clocked at 533 MHz and is hosting a Linux operating system with kernel 2.6.32. In all experiments, the compiler is ICC 8.1 with optimization level -O2. The runtime-system runs on top of Linux and uses the POSIX library to execute multiple processes in a quasi-parallel fashion, see [25] for more details about the runtime-system. Ring buffers in private memory are used for intra-core communication and inter-core communication is realized by the RCKMPI library [5]. When an application is started, a runtime-manager assigns to the application an amount of cores that depends on the overall workload and the application's performance requirements.

The software synthesis tool chain follows the Y-chart design approach outlined in Fig. 1. First, the performance parameters listed in Table 1 are obtained by running some benchmark configurations on Intel's SCC processor. Afterwards, simulated annealing [15] is used to solve the optimization problem stated in Section 6.3. When selecting a new neighboring state, the algorithm randomly modifies either the set of design transformations or the mapping of the current implementation onto the architecture.

When reporting measurements from Intel's SCC processor, the values are the average of ten repetitions.

7.2 The Optimal Degree of Parallelism

The first case study considers a ray-tracing application to study the effect of parallelism on the execution time. We compare the execution time of different application transformations and mappings both when it is estimated with the performance model introduced in Section 6 and when the time is measured on Intel's SCC processor.

Figure 6 illustrates the process network of the ray-tracing application. It can have multiple "intersection" processes to concurrently analyze multiple rays. In addition, the "generator" process generates the rays and the "summation" process merges the rays into a single image.

The ray-tracing application is running on three cores of the SCC. Figure 5a outlines 10 different mapping and replication scenarios of the application. For example, a "G" bubble on core 1 indicates that the "generator" process is

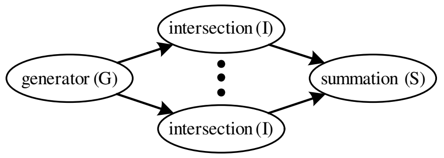


Figure 6: EPN specification of the ray-tracing application that supports replication.

mapped onto core 1. When measuring the execution time on Intel's SCC processor, each core has been selected from a different tile to reduce timing variations due to the network.

To study how replication affects the execution time, the time to generate an image of 100 × 100 pixels with 10 samples per pixel has been measured for the scenarios outlined in Fig. 5a. Figure 5b reports the time that was measured when the scenarios have been executed on the SCC and the time predicted by the performance model described in Section 6. Out of all 10 scenarios, the execution time is reduced the most in scenario "h", which balances the execution time best between the cores maximizing the speedup. Thus, in this example, the optimal degree of parallelism is achieved with five "intersection" processes. The longest execution time is observed in scenario "a", which has only two instances of the intersection process. The execution time of the ray-tracing application in scenario "h" is 29.5 s on the real platform, which corresponds to a speed-up of 2.43 compared to scenario "a", which has an execution time of 72.0 s on the real platform.

The average (absolute) difference between the prediction based on the performance model and the measurement on the real platform is 0.6 s. This indicates that the performance model is able to accurately predict the (average) execution time if replication is used to improve the parallelism of an application.

7.3 Replication and Unfolding

Next, a motion JPEG (MJPEG) decoder is used to measure the speed-up achieved with replication and unfolding. Figure 7 depicts the EPN specification of the MJPEG decoder that supports both replication and unfolding. "split stream" splits the stream into individual frames that are forwarded to the "decode frame" process decoding a com-

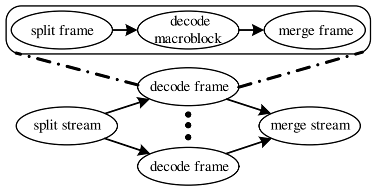


Figure 7: EPN specification of the MJPEG decoder that supports both unfolding and replication.

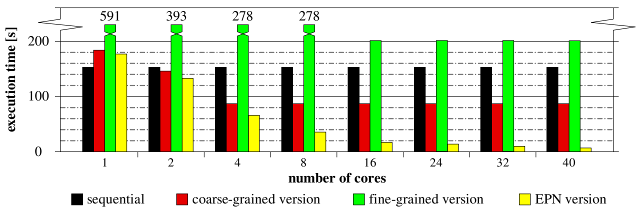


Figure 8: Execution time of the MJPEG decoder application for a varying number of available cores.

plete frame. Afterwards, "merge stream" merges the decoded frames back into a frame. The "decode frame" process can further be unfolded. Then, "split frame" splits a frame into segments of 40 macroblocks that are individually decoded. As the MJPEG decoder only applies intra-frame compression, multiple frames can be decoded in parallel without having dependencies between the replicated processes.

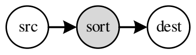
In Fig. 8, the time to decode 5000 frames is compared for a varying number of available cores. For brevity, we only report the results for an exclusive set of sub-architectures; however, the results for the other sub-architectures exhibit similar trends. Motivated by the previous results for the ray-tracing application, we use the optimization framework to find a good level of parallelism and a good mapping for the MJPEG application. Afterwards, we measure the execution time of this implementation on Intel's SCC processor, thus the reported time is the measured execution time. First, as a reference, we measure the execution time of a sequential implementation that consists of just one process. Then, we look at two KPN implementations of the MJPEG decoder, both having a static replication degree of three. In the coarse-grained implementation, a complete frame is decoded in the process "decode frame". This process is further unfolded in the fine-grained implementation. The mapping of both KPN implementations has been optimized using the optimization framework described in Section 7.1. Finally, the execution time of the transformed and optimized EPN implementation is measured. The EPN implementation is based on the specification illustrated in Fig. 7. To improve the performance of the EPN implementation, the degree of parallelism and the mapping are optimized for every considered number of available cores.

The execution time of the EPN implementation is never larger than the execution time of the fine-grained or coarse-grained MJPEG implementation. As the coarse-grained implementation has only five processes, the execution time cannot anymore be improved if more than five cores are available. The execution time of the fine-grained implementation is always larger than the coarse-grained implementation, which might be due to the additional inter-process communication overhead. The fine-grained implementation passes many small packets between the processes so that a large amount of time is just spent to send and receive tokens. If the number of processes is larger than the number of cores, the inter-process communication overhead could further be reduced by using specialized FIFO implementations as, for instance, Windowed FIFOs [10]; however, the trend remains the same. The selected abstraction of the EPN depends on the number of available cores. Up to eight cores, the optimization framework allocates one "decode frame" process and the "split stream" process together on one core, one "decode frame" process together with the "merge stream" process on another core, and one "decode frame" process on every other core. Afterwards, for 16 cores, it allocates on 15 cores one "decode frame" process and maps the "split stream" and "merge stream" process together on one core. For even more cores, it assigns the "split stream" and "merge stream" processes to a dedicated core.

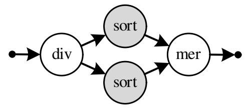
7.4 Speed-up due to Recursion

In the last case study, we evaluate the performance of quicksort, a recursive array-sorting algorithm. The EPN specification is illustrated in Fig. 9. Process "src" ("dest") generates (displays) the input (output) array, and process "sort" sorts the elements in ascending order. As the quicksort algorithm recursively sorts the array, process "sort" can be replaced by a structural description. "div" first partitions the array into two groups: the first group contains the elements that are smaller than the median value and the second group contains the remaining elements. The divided arrays are individually sent to a different instance of the "sort" process. Finally, the individually sorted sub-arrays are merged into a single array. Thus, by recursively unfolding the "sort" process, the original topology can be transformed to have more tasks. As the length of the array that each "sort" process has to sort is halved in each recursion step, the execution time is reduced with each recursion step, as well.

In Fig. 10, the execution time to sort 5000 random arrays with each 5000 elements is compared for a varying number of available cores and different recursion depths. Similar to the MJPEG decoder application, the optimization framework has been used to optimize the final implementation, but the reported numbers are obtained from running the final implementation on the real platform. *No unfolding* refers to the basic quicksort algorithm illustrated in Fig. 9a. *x-times unfolded* refers to an implementation where the "sort" process is x times recursively unfolded. Finally, the EPN implementation leaves the task of unfolding to the optimization framework. The evaluation shows that the optimization framework selects a different unfolding degree depending on the available number of cores. On the one hand, if the num-



(a) Top-level process network of the quicksort algorithm.



(b) Structural description of process "sort".

Figure 9: EPN specification of the quicksort algorithm.

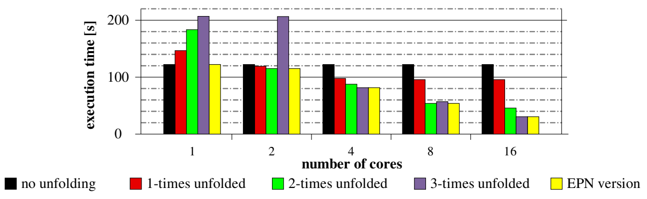


Figure 10: Execution time of quicksort for a varying number of available cores.

ber of cores is small, a low recursion depth achieves best performance as the switching and communication overhead is low. On the other hand, if the number of cores is large, a high recursion depth accomplishes a lower execution time as the array is sorted in parallel.

Even though the EPN specification tries to optimize both the degree of parallelism and the mapping to minimize the execution time, the speed-up is much lower than the optimal speed-up. The speed-up is 2.3 when using eight cores instead of one core and 4.1 when using sixteen cores instead of one core. This might be because additional time is spent to first partition the array into two groups and then to transmit the intermediate results between the different cores.

8. CONCLUSION

In this paper, the semantics of expandable process networks (EPNs) is proposed. EPNs are an extension for streaming programming models that enable the automatic exploration of task, data, and pipeline parallelism by replication and unfolding. The later enables the explicit specification of recursion, commonly used in mathematical and multimedia applications. To include the proposed concepts in the system design, we extend the current Y-chart design practice by an additional design step, which does not only optimize the mapping, but also the application structure to match the available target architecture. This is particularly useful when the available computing resources are not known at design time, as it is the case when a hypervisor distributes the computing resources of a many-core system at runtime. In this case, the EPN semantics enables the synthesis of multiple design implementations that are all derived from one high-level specification. At runtime, the hypervisor can simply select the best-suited implementation depending on the available computing resources. Finally, extensive experiments have been carried out on a 48-core platform that show the effectiveness of the EPN semantics compared to conventional process networks.

In the future, we plan to extend the proposed formalism to adapt the degree of parallelism at runtime when the system's state changes. We further intend to investigate how the number of sub-architectures can be efficiently reduced, e.g., by studying the correlation between the number of available processing elements and the performed design transformation steps.

Acknowledgments

This work was supported by EU FP7 project EURETILE, under grant numbers 247846. Lars Schor was also partially supported by an Intel PhD Fellowship.

References

[1] B. Abali et al. Balanced Parallel Sort on Hypercube Multiprocessors. *IEEE T. Parall. Distr.*, 4(5):572–581, 1993.
[2] J. Bigler et al. Design for Parallel Interactive Ray Tracing Systems. In *Proc. Symp. on Interactive Ray Tracing*, pages 187–196, 2006.
[3] C. Brooks et al. Ptolemy II - Heterogeneous Concurrent Modeling and Design in Java. Technical report, University of California at Berkeley, 2005.
[4] B. Chapman et al. *Using OpenMP: Portable Shared Memory Parallel Programming.* The MIT Press, 2007.
[5] I. Comprés Ureña et al. RCKMPI – Lightweight MPI Implementation for Intel's Single-chip Cloud Computer (SCC). In *Recent Advances in the Message Passing Interface*, volume 6960 of *LNCS*, pages 208–217. Springer, 2011.
[6] T. H. Cormen et al. *Introduction to Algorithms.* McGraw-Hill Higher Education, 2nd edition, 2001.
[7] A. Fedorova et al. Cypress: A Scheduling Infrastructure for a Many-Core Hypervisor. In *Proc. MMCS*, 2008.
[8] M. Geilen and T. Basten. Reactive Process Networks. In *Proc. EMSOFT*, pages 137–146, 2004.
[9] M. I. Gordon et al. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs. *SIGPLAN Not.*, 41(11):151–162, 2006.
[10] W. Haid et al. Efficient Execution of Kahn Process Networks on Multi-Processor Systems Using Protothreads and Windowed FIFOs. In *Proc. ESTIMedia*, pages 35–44, 2009.
[11] J. Howard et al. A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. In *Proc. ISSCC*, pages 108–109, 2010.
[12] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In *Information Processing*, pages 471–475, 1974.
[13] B. Kienhuis et al. An Approach for Quantitative Analysis of Application-Specific Dataflow Architectures. In *Proc. ASAP*, pages 338–349, 1997.
[14] B. Kienhuis et al. Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures. In *Proc. CODES*, pages 13–17, 2000.
[15] S. Kirkpatrick et al. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.
[16] E. Lee and D. Messerschmitt. Synchronous Data Flow. *Proc. IEEE*, 75(9):1235–1245, 1987.
[17] P. Murthy and E. Lee. On the Optimal Blocking Factor for Blocked, Non-Overlapped Multiprocessor Schedules. In *Proc. of Asilomar Conf. on SSC*, pages 1052–1057, 1994.
[18] P. S. Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.
[19] C. Park et al. Extended Synchronous Dataflow for Efficient DSP System Prototyping. *Design Automation for Embedded Systems*, 6:295–322, 2002.
[20] A. Pettorossi and M. Proietti. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Comput. Surv. (CSUR)*, 28(2):360–414, June 1996.
[21] J. Pino and E. Lee. Hierarchical Static Scheduling of Dataflow Graphs onto Multiple Processors. In *Proc. ICASSP*, volume 4, pages 2643–2646, 1995.
[22] A. Polze and P. Tröger. Trends and Challenges in Operating Systems – from Parallel Computing to Cloud Computing. *Concurrency and Computation: Practice and Experience*, 24(7):676–686, 2012.
[23] I. Sander and A. Jantsch. System Modeling and Transformational Design Refinement in ForSyDe. *IEEE TCAD*, 23(1):17–32, 2004.
[24] L. Schor et al. Scenario-Based Design Flow for Mapping Streaming Applications onto On-Chip Many-Core Systems. In *Proc. CASES*, pages 71–80, 2012.
[25] L. Schor et al. Reliable and Efficient Execution of Multiple Streaming Applications on Intels SCC Processor. In *Proc. ROME*, 2013.
[26] L. Seiler et al. Larrabee: A Many-Core x86 Architecture for Visual Computing. *ACM Trans. Graph.*, 27(3):18:1–18:15, 2008.
[27] L. Thiele et al. Mapping Applications to Tiled Multiprocessor Embedded Systems. In *Proc. ACSD*, pages 29–40, 2007.
[28] W. Thies et al. StreamIt: A Language for Streaming Applications. In *Proc. Int'l Conf. on Compiler Construction*, pages 179–196, 2001.
[29] S. Verdoolaege et al. pn: A Tool for Improved Derivation of Process Networks. *EURASIP J. Embedded Syst.*, 2007(1):19:1–19:13, 2007.