Scenario-Based Design Flow for Mapping Streaming Applications onto On-Chip Many-Core Systems

Lars Schori, Iluliana Bacivarov, Devendra Rai, Hoeseok Yang, Shin-Haeng Kang, and Lothar Thiele

1Computer Engineering and Networks Laboratory, ETH Zurich, CH-8092 Zurich, Switzerland
2School of Electrical Engineering and Computer Science, Seoul National University, Seoul, South Korea
firstname.lastname@tik.ee.ethz.ch

ABSTRACT

The next generation of embedded software has high performance requirements and is increasingly dynamic. Multiple applications are typically sharing the system, running in parallel in different combinations, starting and stopping their individual execution at different moments in time. The different combinations of applications are forming system execution scenarios. In this paper, we present the distributed application layer, a scenario-based design flow for mapping a set of applications onto heterogeneous on-chip many-core systems. Applications are specified as Kahn process networks and the execution scenarios are combined into a finite state machine. Transitions between scenarios are triggered by behavioral events generated by either running applications or the run-time system. A set of optimal mappings are precalculated during design-time analysis. Later, at run-time, hierarchically organized controllers monitor behavioral events, and apply the precalculated mappings when starting new applications. To handle architectural failures, spare cores are allocated at design-time. At run-time, the controllers have the ability to move all processes assigned to a faulty physical core to a spare core. Finally, we apply the proposed design flow to design and optimize a picture-in-picture software.

Categories and Subject Descriptors

C.3 [Special-purpose and application-based systems]: Real-time and embedded systems; C.1.4 [Parallel architectures]: Distributed architectures

General Terms

Algorithm, Design, Performance

Keywords

On-chip many-core systems, design flow, scenario-based model of computation, MPSoC, mapping optimization

1. INTRODUCTION

Real-time physics, artificial intelligence, or 3D rendering effects will soon be state-of-the-art in embedded devices and have in common that they have high performance requirements, are highly parallelizable, and increasingly dynamic. The demand for a high degree of visual realism in multimedia applications has driven system architects to use on-chip many-core systems [16]. Intel’s SCC processor [10] is a prominent example of such an on-chip many-core architecture. By incorporating 48 cores into a single processor, the SCC processor is a prototype of future embedded platforms. Even more, the next generation of on-chip many-core systems is supposed to have hundreds of heterogeneous cores [3]. Thus, traditional methods to design multiprocessor systems are not anymore appropriate to many-core systems that are architecturally more complex. The software of future embedded systems is composed of a set of applications and only a fraction of all applications is running in parallel. We call a scenario a certain system state with a predefined set of running or paused applications. Typically for embedded systems, the number of possible scenarios is restricted and the scenarios are known at design time. Consequently, a design flow for mapping dynamic streaming applications onto on-chip many-core systems has to provide three key features to the system architect. First, a high-level specification model that hides unnecessary implementation details but provides enough flexibility to specify dynamic interactions between applications. Second, an optimal mapping of the application onto the architecture in a transparent manner. Third, run-time support to dynamically change the workload of the system.

This paper proposes the distributed application layer (DAL), a scenario-based design flow that supports design, optimization, and simultaneous execution of multiple applications targeting heterogeneous many-core systems. Applications are specified as Kahn process networks (KPNs) [11]. KPNs are suitable for a general description of a high-level design flow as they are determinate, provide asynchronous execution, and are capable to describe data-dependent behavior. In case a higher predictability is required, the application model can be restricted, e.g., to synchronous data flow (SDF) graphs [15]. To coordinate the execution of different applications, we use a finite state machine (FSM), where each scenario is represented by a state. Transitions between scenarios are triggered by behavioral events generated by either running applications or the run-time system.

The design flow that we propose in this paper is illustrated in Fig. 1. During design-time analysis, a set of opti-
Figure 1: Overall structure of the scenario-based design flow.

1. RELATED WORK

Programming paradigms for many-core systems have to tackle various new challenges. Techniques that worked well for systems with just a few cores will become the bottleneck in the next few years [25]. The KPN model of computation [11] is the basis of several frameworks for designing multi-processor systems, such as Daedalus [19], DOL [24], Koski [12], or SHIM [7]. As they provide a single mapping, they are only able to handle dynamism by over-provisioning the system.

To capture the increasing dynamism in future embedded applications, mapping strategies are proposed that generate a set of mappings at design-time [9, 17, 23]. Then, a run-time mechanism selects the best fitting mapping depending on the actual resource requirements of all active applications. The concept of system scenarios is introduced in [9] by automatically analyzing a system for similarities from a cost perspective. It has been applied in [23] to comprehend the dynamic behavior of an application as a set of scenarios. Each scenario is specified as an SDF [15] graph. In contrast, our work just specifies the running and paused applications per scenario, and each application is separately specified as a KPN. We think that the KPN model of computation is better suited to describe a high-level design flow and the individual specification of each application enables a better resource usage. Finally, multiple mappings that differ from each other in terms of power consumption and performance are generated in [17], but the approach is not scalable due to the centralized run-time manager.

The concept of hybrid mapping strategies has already been investigated in various other works. In [1], it is proposed to compute various system configurations and to calculate an optimal task allocation and scheduling for each of them. At run-time, the decision whether a transition between allocations is feasible, is based on precalculated migration costs. In our work, we assume that processes are resident. This makes design-time analysis more complex, but eliminates undesired disruption due to process migration. Similarly, process migration is prohibited in [21]. They use statistical methods to compute mappings for different interconnected usage-scenarios. As the approach evaluates a large number of mappings, it might not scale with the size of the platform. A hybrid mapping strategy is proposed in [22] that calculates several resource-throughput trade-off points at design-time. At run-time, it selects the best point with respect to available resources. However, the approach is restricted to homogeneous platforms and the schedulability of the system is only known at run-time.

In order to tolerate run-time processor failures, a multi-step mapping strategy is proposed in [14]. After calculating a static mapping for all possible failure scenarios, a processor-to-processor mapping is performed at run-time. As analyzing and storing a mapping scenario for each failure scenario is not scalable, we allocate spare cores at design-time.

Various options to design a run-time manager have been discussed in literature. On the one hand, a fully centralized approach can be seen as a broker running on its own core. While centralized approaches are widely used in multi-core systems [17, 20], they impose a performance bottleneck on many-core systems. On the other hand, a fully distributed approach [4, 13] leads to a high complexity. Therefore, we
propose a hierarchical centralized approach, that takes system scalability into account at a low complexity.

The KPN model of computation has been extended in [8] with the ability to support sporadic control events. However, the work includes neither concrete execution semantics nor mapping strategies. By separately specifying the execution scenarios as an FSM, we are able to formally define an execution semantic and to propose a hybrid design-time/run-time mapping strategy to efficiently execute multiple dynamic KPNs on a many-core platform. Finally, we define the semantics of a scenario change and propose a high-level interface for behavioral and fault events.

3. ARCHITECTURE MODEL

In this section, the architecture model is introduced. In order to describe the considered architecture in an abstract manner, we use a hierarchical representation. The on-chip many-core architecture \( A = \{C, D, N^{(1)}, \ldots, N^{(9)}\} \) consists of a set of cores \( C \), a set of core types \( D \), \( \eta \) sets of networks \( N^{(1)} \) to \( N^{(9)} \), and a function \( z \). The function \( z : C \to D \) assigns each core \( c \in C \) its type \( z(c) \in D \). The set of core types might be used to differ between DSP and RISC components or to distinguish between different operating frequencies. Each set of networks corresponds to a communication layer so that the architecture consists of \( \eta \) communication layers. A network \( n^{(k)} \in N^{(k)} \) is defined as a subset of \( C \). In particular, a network \( n^{(1)} \in N^{(1)} \) represents the intra-core communication, i.e., \( |N^{(1)}| = |C| \) and for each \( c \in C \), there is a network \( n^{(1)} \in N^{(1)} \) with \( n^{(1)} = \{c\} \). The second set of networks \( N^{(2)} \) partitions the cores into tiles so that each core is assigned to exactly one tile, i.e., we have \( \bigcup_{n^{(2)} \in N^{(2)}} = C \) and \( n^{(2)} \cap n^{(2)} = \emptyset \) for all \( n^{(2)} \in N^{(2)}, n^{(2)} \in N^{(2)} \) and \( i \neq j \). Similarly, every other set of networks \( N^{(k)} \) partitions the cores so that each network \( n^{(k)} \in N^{(k)} \) contains multiple subordinate networks, i.e., there exists a \( n^{(k)} \in N^{(k)} \) with \( n^{(k-1)} \in N^{(k)} \) for all \( n^{(k-1)} \in N^{(k-1)}, \bigcup_{n^{(k-1)} \in N^{(k-1)}} = C \), and \( n^{(k)} \cap n^{(k)} = \emptyset \) for all \( n^{(k)} \in N^{(k)}, n^{(k)} \in N^{(k)} \) and \( i \neq j \). Finally, \( N^{(\eta)} = \{n^{(\eta)}\} \) contains a single network hierarchically connecting all processors, i.e., \( n^{(\eta)} = C \). Furthermore, the type of a network is defined as concatenation of all core types of the network.

The hierarchical representation of the architecture is a generalization of the well-known tile-based multiprocessor model [6]. First prototypes of future on-chip many-core systems typically consist of three sets of networks, i.e., \( \eta = 3 \), which correspond to the three communication layers intra-core, intra-tile, and inter-tile communication [10, 18, 26]. A shared bus is often used for intra-tile communication and a NoC for inter-tile communication. Figure 2 sketches a typical on-chip many-core system with \( \eta = 3 \) and its abstract representation is illustrated in Fig. 3.

Due to high power densities, on-chip many-core systems are prone to failures. In this work, we restrict ourselves to a failure of a core or a router. In case that a router fails, the tile is not anymore available. In any case, we suppose that either the failed component or any other component detects the failure and sends a fault event to the run-time manager. In order to handle architectural failures at run-time, spare cores and tiles are allocated at design-time. We call the abstract representation of the architecture without spare cores and tiles virtual architecture \( \mathcal{A} \). \( \mathcal{A} = \{C, D, N^{(1)}, \ldots, N^{(9)}, z\} \) consists of a set of virtual cores \( \mathcal{C} \), the set of core types \( D \) of architecture \( A \), \( \eta \) sets of virtual networks \( \mathcal{V}^{(1)}, \ldots, \mathcal{V}^{(9)} \), and a function \( \mathcal{z} \). The function \( \mathcal{z} : \mathcal{C} \to D \) assigns each core \( c \in \mathcal{C} \) its type \( \mathcal{z}(c) \in D \). It is the system architect’s task to specify the spare components at design-time. One possibility to generate \( \mathcal{A} \) is to remove from each network \( n^{(i)} \in N^{(i)} \), one subordinate network \( n^{(i-1)} \in N^{(i-1)} \) per network type so that each network is able to correct one failure. Finally, each virtual network \( \mathcal{V}^{(i-1)} \) can be mapped onto any physical network \( n^{(i-1)} \) that belongs to the same superior network \( n^{(i)} \) and has the same type as \( n^{(i)} \). Consider for example the system illustrated in Fig. 2. Suppose that all cores are of the same type and the system architect selects \( n^{(3)} \) as spare network of \( n^{(2)} \). Then the virtual networks \( n^{(1)}_1 \) and \( n^{(1)}_2 \) can be mapped onto the physical networks \( n^{(2)}_1 \), \( n^{(2)}_2 \), and \( n^{(2)}_3 \), but not on \( n^{(1)}_4 \) as it belongs to a different tile.

4. MODEL OF COMPUTATION

In this section, we formally define the scenario-based model of computation for streaming applications. We first discuss the specification of individual applications as KPNs. Afterwards, the dynamic behavior of the system is captured by a set of scenarios.

4.1 Application Specification

The KPN [11] model of computation is considered in this paper to specify the application behavior. In particular, an application \( p = (V, Q) \) consists of autonomous processes \( v \in V \) that can only communicate through unbounded point-to-point FIFO channels \( \eta \in Q \). A process \( v \in V \) is a mono-tonic and determinate mapping \( F \) from (one or more) input streams to one (or more) output streams. As every process \( v \in V \) is monotonic and determinate, there is no notion of time and the output just depends on the sequence of tokens in the individual input streams [8]. Conceptually, a KPN is non-terminating, i.e., once the process network has started it does not stop running. As this is not in accordance with the specification of a dynamic system, we extend the definition of a KPN with the ability to terminate and pause. To this end, we first propose the high-
level API illustrated in Listing 1 to specify KPN processes. Roughly speaking, the INIT procedure is responsible for the initialization and is executed once at the startup of the application. Afterwards, the execution of a process is split into individual executions of the FIRE procedure, which is repeatedly invoked by the system scheduler. Once an application is stopped, the FINISH procedure is called for cleanup. Communication is enabled by calling high-level read and write procedures and each process has the ability to request a scenario change by calling the send_event procedure.

Listing 1: Implementation of a KPN process using the proposed API.

```
01 procedure INIT(ProcessData *p) // process initialization
02  initialize();
03 end procedure
04
05 procedure FIRE(ProcessData *p) // process execution
06  fifo->read(buf, size); // read from fifo
07  if (buf[0] == eventkey)
08    send_event(e); // send event e
09  end if
10  manipulate();
11  fifo->write(buf, size); // write to fifo
12 end procedure
13
14 procedure FINISH(ProcessData *p) // process cleanup
15  cleanup();
16 end procedure
```

Now, we are able to introduce and specify the four generic actions START, STOP, PAUSE, and RESUME of a KPN \( p = (V, Q) \). The semantic of those four actions is summarized in Table 1. Stopping an application \( p \) might be problematic. Therefore, the FIRE method of all processes \( v \in V \) is aborted only at predefined points such as when process \( v \) is calling a read or write procedure, or the execution of the FIRE procedure is finished. In the case that a process is blocked, i.e., the process attempts to read from an empty channel, the blocking is resolved before the FIRE method can be aborted. Finally, the FINISH procedure is executed to perform cleanup operations.

4.2 Scenario Specification

The dynamic behavior of a system can be captured by a set of scenarios. Each scenario represents a set of concurrently running or paused applications. Scenario transitions are triggered by behavioral events generated by either running applications or the run-time system. Consider, for example, the (simplified) car entertainment system shown in Fig. 4. The software has five scenarios, with one to three applications. After startup, the system enters the map scenario where the MAP application is running and displaying the current position of the car on a map. Depending on the situation, the scenario might change. For example, the driver starts to drive backwards so that the parking assistant is started (scenario parking), the voice navigation notifies the driver to take the next exit (scenario nav), or the driver starts listening to some music (scenario map and music). In addition, the voice navigation might notify the driver to change the driving direction while listening to music. To this end, the system switches to the scenario nav and music, and pauses the MP3 application.

Formally, we define the above described dynamic behavior of a system by an FSM \( \mathcal{F} = (S, E, T, P, s_0, a, r, h) \) that consists of the set of scenarios \( S \), the set of events \( E \), the set of directed transitions \( T \in S \times S \), the set of applications \( P \), an initial scenario \( s_0 \in S \), and three functions \( a, r, \) and \( h \). The function \( a : T \to E \) maps a transition \( t \in T \) to a set of triggering events \( a(t) \subseteq E \) for all \( t \in T \). The function \( r : S \to P \) assigns each scenario \( s \in S \) a set of running applications \( \{s\} \subseteq P \) and the function \( h : S \to P \) assigns each scenario \( s \in S \) a set of paused applications \( \{s\} \subseteq P \). As we suppose that there is only one instance of an application, \( \{s\} \cap \{h\} = \emptyset \) for all \( s \in S \). Figure 5 presents an example of an FSM \( \mathcal{F} = (S, T, E, P, s_0, a, r, h) \) with four scenarios \( s_0, s_1, s_2, \) and \( s_3 \) among which \( s_0 \) is initially active. The scenarios are linked by the set of transitions \( T = \{t_1, t_2, t_3, t_4, t_5\} \) such that \( t_1 = (s_0, s_1), t_2 = (s_1, s_2), t_3 = (s_2, s_3), t_4 = (s_3, s_1) \), and \( t_5 = (s_3, s_0) \). The function \( a \) assigns each transition its triggering events. For example, the transition \( t_3 \) from scenario \( s_1 \) to scenario \( s_2 \) happens when the events \( e_2 \) or \( e_3 \) are detected in scenario \( s_1 \). Finally, the functions \( r \) and \( h \) assign each scenario a list of running and paused applications.

4.3 Execution Semantics

The above introduced model of \( \mathcal{F} \) is a Moore machine, i.e., each scenario has a list of running and paused applications, and each transition between scenarios has a set of events that trigger the transition. However, in terms of execution, each transition is associated with a set of actions. For example, transition \( t_1 \) of the FSM \( \mathcal{F} \) illustrated in Fig. 5 is associated with the action \( \{\text{pause application } p_1\} \), and transition \( t_4 \) is associated with the actions \( \{\text{stop application } p_2, \text{start application } p_3\} \).

Therefore, in terms of execution, we map the system evolution to a Mealy machine and transform \( \mathcal{F} \) into a new
then paused, and resumed. Suppose that transition \( \tilde{S} = \text{transition set of scenarios} \) \( S \), \( T, E, P, s \), \( T, P, t \), and \( a : T \rightarrow E \) are defined as in Section 4.2 and \( T = T \cup \Omega_0 \). The functions \( u^x, u^y, u^p \), and \( u' \) assign each transition \( t \in T \) the set of applications to be started, stopped, paused, and resumed. Suppose that transition \( t = (s_x, s_y) \), then \( u^x(t), u^y(t), u^p(t), \) and \( u'(t) \) are formally defined as:

START: \( u^x(t) = r(s_y) \setminus (r(s_x) \cup h(s_x)) \subseteq P \)

STOP: \( u^y(t) = (r(s_x) \cup h(s_y)) \setminus (r(s_y) \cup h(s_y)) \subseteq P \)

PAUSE: \( u^p(t) = h(s_y) \cap r(s_x) \subseteq P \)

RESUME: \( u'(t) = r(s_y) \cap h(s_x) \subseteq P \)

In other words, whenever transition \( t \in \tilde{T} \) is triggered, all applications \( p \in u^x(t) \) are started, all applications \( p \in u^y(t) \) are stopped, all applications \( p \in u^p(t) \) are paused, and all applications \( p \in u'(t) \) are resumed.

In terms of execution, the initial transition \( t_0 \) takes place after startup so that the FSM enters scenario \( s_0 \). Whenever an event \( e \in E \) is received that corresponds to one of the outgoing transitions of the current scenario, the transition takes place. In other words, an event \( e \in E \) triggers a transition \( t \in T \) if and only if \( e \in a(t), t = (s_x, s_y), \) and \( s_x \) is the current scenario of the FSM.

Conceptually, the reaction of the system to an event is immediate, i.e., the actions listed in Table 1 are performed in zero time. However, as the production and execution of these actions take a certain amount of time, we have to come up with additional rules, which preserve the described semantics. In particular, we assume that a transition is only triggered if the system is in a stable scenario. A stable scenario is reached if the execution of all actions triggered by the previous transition is completed. This rule is required as events might arrive faster than they can be processed. If the system is not yet in a stable scenario, the execution of new actions might cause the system to move to an unknown or wrong scenario. Practically, this requirement can be realized by storing all incoming events in a FIFO queue so that the events are processed in a First-Come-First-Served (FCFS) manner. If the current scenario has an outgoing transition that is sensitive to the head event of the FIFO queue, the transition takes place and the event is removed from the FIFO queue. Otherwise, the event is removed without changing the active scenario.

Figure 5: Example of an FSM \( \mathcal{F} = (S, T, E, P, s_0, a, r, h) \).

FSM \( \tilde{\mathcal{F}} = (S, E, \tilde{T}, P, t_0, a, u^x, u^y, u^p, u') \) that consists of the set of scenarios \( S \), the set of events \( E \), the set of directed transitions \( \tilde{T} \subseteq S \times S \), the set of applications \( P \), an initial transition \( t_0 \in T \), and six functions \( a, u^x, u^y, u^p, \) and \( u' \). \( S, E, P, \) and \( a : \tilde{T} \rightarrow E \) are defined as in Section 4.2 and \( T = T \cup \Omega_0 \). The functions \( u^x, u^y, u^p, \) and \( u' \) assign each transition \( t \in \tilde{T} \) the set of applications to be started, stopped, paused, and resumed. Suppose that transition \( t = (s_x, s_y) \), then \( u^x(t), u^y(t), u^p(t), \) and \( u'(t) \) are formally defined as:

START: \( u^x(t) = r(s_y) \setminus (r(s_x) \cup h(s_x)) \subseteq P \)

STOP: \( u^y(t) = (r(s_x) \cup h(s_y)) \setminus (r(s_y) \cup h(s_y)) \subseteq P \)

PAUSE: \( u^p(t) = h(s_y) \cap r(s_x) \subseteq P \)

RESUME: \( u'(t) = r(s_y) \cap h(s_x) \subseteq P \)

In other words, whenever transition \( t \in \tilde{T} \) is triggered, all applications \( p \in u^x(t) \) are started, all applications \( p \in u^y(t) \) are stopped, all applications \( p \in u^p(t) \) are paused, and all applications \( p \in u'(t) \) are resumed.

In terms of execution, the initial transition \( t_0 \) takes place after startup so that the FSM enters scenario \( s_0 \). Whenever an event \( e \in E \) is received that corresponds to one of the outgoing transitions of the current scenario, the transition takes place. In other words, an event \( e \in E \) triggers a transition \( t \in \tilde{T} \) if and only if \( e \in a(t), t = (s_x, s_y), \) and \( s_x \) is the current scenario of the FSM.

Conceptually, the reaction of the system to an event is immediate, i.e., the actions listed in Table 1 are performed in zero time. However, as the production and execution of these actions take a certain amount of time, we have to come up with additional rules, which preserve the described semantics. In particular, we assume that a transition is only triggered if the system is in a stable scenario. A stable scenario is reached if the execution of all actions triggered by the previous transition is completed. This rule is required as events might arrive faster than they can be processed. If the system is not yet in a stable scenario, the execution of new actions might cause the system to move to an unknown or wrong scenario. Practically, this requirement can be realized by storing all incoming events in a FIFO queue so that the events are processed in a First-Come-First-Served (FCFS) manner. If the current scenario has an outgoing transition that is sensitive to the head event of the FIFO queue, the transition takes place and the event is removed from the FIFO queue. Otherwise, the event is removed without changing the active scenario.

Figure 6: Overall mapping optimization approach with design-time and run-time component.

5. HYBRID MAPPING OPTIMIZATION

In this section, we present a hybrid design-time / run-time strategy for mapping streaming applications onto on-chip many-core platforms. The design-time component calculates an optimal mapping for each application and scenario where the application is either running or paused. At run-time, the dynamic mapping of the applications onto the architecture is controlled by a run-time manager, which monitors events, chooses an appropriate mapping, and finally executes the required actions, see Fig. 6.

5.1 Design-Time Analysis and Optimization

In this subsection, we introduce the proposed approach for design-time optimization. We minimize the maximum core utilization subject to utilization and communication constraints so that we obtain a system with a balanced workload.

5.1.1 Motivational Example

We start with an example. Consider the car entertainment system outlined in Fig. 4. As the workload of scenario \( \text{nav} \) is different from the workload of scenario \( \text{map and music} \), different mappings should be used in both scenarios to optimize the performance. However, changing the process to core assignment with each scenario transition might lead to bad performance due to reconfiguration overhead. Therefore, the approach proposed in this paper assumes that processes are resident, i.e., once a process is mapped onto a core, it will not be remapped to another core. In other words, if an application is active in two connected scenarios, it has the same mapping in both scenarios. We think that this restriction is well suited for embedded systems where process migration leads to non-negligible costs in terms of time and system overhead. For example, consider again the car entertainment system. The \( \text{NAV} \) application will have the same mapping in all active scenarios. However, the mapping of the \( \text{NAV} \) application might be different in the scenarios \( \text{nav} \) and \( \text{music} \) as they are not connected by a direct transition.

5.1.2 Mapping Specification

The design-time component calculates an optimal mapping for each application and scenario where the application is either running or paused. Thus, the output of the design-time analysis is a collection \( M \) of optimal mappings and exactly one mapping \( \hat{M} \) of \( M \) is valid for a pair of an application and a scenario.
Formally, a mapping $m \in M$ is a triple $(p, S^m, B^m)$ where $p$ is an application, $S^m \subseteq S$ is a subset of scenarios, and $B^m \subseteq \mathcal{V} \times \mathcal{C}$ the set of binding relations. $S^m$ denotes the set of scenarios for which mapping $m$ is valid. As processes are resident, the same mapping might be valid for more than one scenario. Finally, a binding relation $(v, c) \in B^m$ denotes that process $v$ is bound to the virtual core $c$.

In the following, we propose a two-step procedure to calculate the mappings $m \in M$. First, we calculate pairs $<p, S^m>$ of an application $p$ and a subset of scenarios $S^m$ so that the size of each subset is minimized and no process migration is required if application $p$ is using the same mapping in all scenarios $s \in S^m$. At the end of the first step, we allocate for each pair $<p, S^m>$ a mapping $m \in M$. Afterwards, in a second step, we calculate for each mapping $m \in M$ a set of optimized binding relations $B^m$ so that an objective function is minimized and additional architectural constraints are fulfilled.

5.1.3 Mapping Generation

In a first step, we calculate pairs $<p, S^m>$ of an application $p$ and a subset of scenarios $S^m$ so that the size of each subset is minimized and additional constraints are fulfilled. The additional constraints ensure that no process migration is required if application $p$ is using the same mapping in all scenarios $s \in S^m$. In particular, we can identify the following three constraints:

**Constraint 1**: Each application is mapped:

$p_t \in (r(s) \cup h(s))$

$\Rightarrow \exists m = (p, S^m, B^m) \in M : p_t = p$ and $s \in S^m$.

**Constraint 2**: Two mappings do not overlap:

$m_1 = (p, S^{m_1}, B^{m_1})$ and $m_2 = (p, S^{m_2}, B^{m_2})$

$S^{m_1} \cap S^{m_2} = \emptyset$.

**Constraint 3**: Process migration is not allowed:

$p_t \in ((r(s_1) \cup h(s_1)) \cap (r(s_2) \cup h(s_2)))$ and $t = (s_1, s_2) \in T$

$\Rightarrow \exists m = (p, S^m, B^m) \in M : p_t = p$ and $s_1, s_2 \in S^m$.

The mapping generation problem can be solved by calculating for each application $p$ the maximally connected components of a subgraph that just contains all scenarios where $p$ is either running or paused. Then, we can generate a new pair $<p, S^m>$ for each component of this subgraph. Algorithm 1 presents the pseudocode to calculate all pairs $<p, S^m>$. The algorithm generates the pairs $<p, S^m>$ by sequentially analyzing all applications. First, a subgraph $G = (S^{\text{sub}}, T^{\text{sub}})$ is determined by removing all scenarios $s \in S$ in which application $p$ is neither running nor paused. Then, we determine the maximally connected components $G_{\text{conn}}^{s} = (S^{\text{conn}}(s), T^{\text{conn}}(s)) \in G^{\text{conn}}$ of subgraph $G$. In other words, the scenarios are partitioned into non-overlapping sets such that there is no transition between nodes in different subsets $G_{\text{conn}}^{s}$ and the subsets are as large as possible. Finally, a new pair $<p, S^{\text{conn}}(s)>$ is generated for each maximally connected component. By relying on a breadth-first search algorithm to calculate the set of all maximally connected components, the calculation of all pairs has a computational complexity of $O(|P|(|T| + |S|))$.

Finally, as application $p$ uses the same mapping in all scenarios $s \in S^m$, we can allocate for each pair $<p, S^m>$ a mapping $m = (p, S^m, \cdot) \in M$.

**Algorithm 1**: Pseudocode to generate all pairs $<p, S^m>$ of an application $p$ and a subset of scenarios $S^m$ so that the number of elements per subset is minimized and the constraints specified in Section 5.1.3 are fulfilled.

**Input**: FSM $\mathcal{F} = (S, T, E, P, s_0, a, r, b)$

**Output**: set of pairs $<p, S^m>$

1. for all applications $p \in P$ do
2. $S^{\text{sub}} \leftarrow \{ s \in S | p \in (r(s) \cup h(s)) \}$
3. $T^{\text{sub}} \leftarrow \{ t = (s_1, s_2) \in T | p \in (r(s_1) \cup h(s_1)) \lor p \in (r(s_2) \cup h(s_2)) \}$
4. $G \leftarrow (S^{\text{sub}}, T^{\text{sub}})$
5. $G^{\text{conn}} \leftarrow$ set of all maximally connected components of $G$
6. for all $G_{\text{conn}}^{s} \in G^{\text{conn}}$ do
7. $\text{gen. pairs for each component}$
8. add $<p, S^{\text{conn}}(s)>$
9. end for
10. end for

5.1.4 Mapping Optimization

In the second step, we calculate for each mapping $m \in M$ the set of binding relations $B^m$ so that the objective function, i.e., the maximum core utilization, is minimized and a set of predefined architectural constraints are fulfilled. The number of firings of process $v$ per time unit is $f(v)$ and the maximum execution time of process $v$ on a core of type $d$ is $w(v, d)$. Furthermore, $M^* \subseteq M$ denotes the subset of all mappings with $s \in S$ and $p \in r(s)$, i.e., $M^* = \{ (p, S^m, B^m) \in M | p \in r(s) \land s \in S^m \}$. The binding relations $B^m$ are calculated so that the maximum core utilization is minimized, and the utilization and communication constraints are met in each scenario:

**Objective function**: The optimization goal of this problem is to minimize the maximum core utilization. In order to incorporate the different scenarios into a single objective function, we assign each scenario $s \in S$ an execution probability $\chi_s$ [21] so that the object function can formally be stated as:

$$\min \left( \max_{v \in \mathcal{V}} \sum_{s \in S} \sum_{m \in M^*} \sum_{v \in V} \chi_s \cdot f(v) \cdot w(v, c_{s}(v)) \right).$$

**Constraint 4**: In order to make sure that the cores are able to handle the processing load, the following relation has to be satisfied for all cores $c \in \mathcal{C}$ and all states $s \in S$ of the FSM $\mathcal{F}$:

$$\sum_{m \in M^*} \sum_{v \in V} f(v) \cdot w(v, c_{s}(v)) \leq 1.$$

**Constraint 5**: Similarly, we can formulate the bandwidth requirement for each network by adding the data volume per time unit of each channel. Then, the aggregated data volume for each network $n$ must be smaller than its supported rate. As the applications are mapped onto a virtual architecture, one has to consider all possible separations between the processes. However, due to the hierarchical structure of the architecture, a virtual network is only mapped onto a physical network within the same superior network so that the maximum separation is bounded.

5.2 Run-Time Manager

In this subsection, we discuss the required run-time support to execute a set of applications $P$ on an on-chip many-
core architecture $\mathcal{A}$. The required run-time support is provided by a run-time manager that has the task to generate commands towards the operating system to ensure the execution semantics described in Section 4.3. Traditionally, run-time managers are either centralized or distributed. However, as a centralized approach comes with a performance bottleneck and a distributed approach leads to a high complexity, both approaches do not fulfill the requirements of embedded many-core systems. In this paper, we propose to split the workload among hierarchically organized controllers. In the following, we first discuss the general ideas of a hierarchical control mechanism and then describe the functionality of each individual controller.

5.2.1 Hierarchical Control Mechanism

The general idea of the hierarchical control mechanism is to assign each network $n \in \{N^{(2)}, \ldots, N^{(9)}\}$ its own controller $v^c \in V^c$ that handles all inner-network dynamism. In particular, the controller assigned to a network $n \in N^{(2)}$ monitors for behavioral and fault events. Whenever such a controller receives an event, it handles the event if it just affects the controller’s network, and otherwise it sends the event to the controller of its superior network.

As the controllers communicate via FIFO channels $q^c \in Q^c$, the hierarchical control mechanism can be represented as a process network $p^c = (V^c, Q^c)$. Algorithm 2 shows the pseudocode to generate the process network $p^c$ for architecture $\mathcal{A}$. To provide bidirectional communication, two FIFO channels connect each controller with its superior controller.

Figure 7 illustrates the process network $p^c$ for the architecture shown in Fig. 3. The controllers can be categorized into three different types:

- A **slave** controller is responsible for a tile, i.e., for a network $n^{(2)} \in N^{(2)}$. All architectural units in network $n^{(2)}$ and all processes $v$ assigned to a core $c \in n^{(2)}$ are able to send events to the **slave** controller. In order to control the execution of a process, a **slave** controller is also able to send commands to the underlying operating system.

- A **interlayer** controller is responsible for a network $n^{(i)} \in N^{(i)}$ with $i = [3, \eta - 1]$ and $\eta$ the number of communication layers. It receives all events that cannot be handled by its subordinates. The **interlayer** controller processes an event if it only affects its own network. Otherwise, it sends the event to its superior controller.

- The **master** controller is responsible for network $n^{(0)} \in N^{(0)}$. It processes all events that cannot be handled by any other controllers.

Nowadays, all cores of a tile often share the same operating system so that one **slave** controller can dynamically allocate processes to all cores of the tile. In case that each core has its dedicated operating system, we assign each core its own **slave** controller to ensure the interaction between the control mechanism and the operating system.

5.2.2 Hierarchical Event Processing

A controller of the hierarchical control mechanism only handles events that just affect the controller’s network. Otherwise, the controller sends the event to the controller of its superior network. In the following, we detail this procedure for an **interlayer** controller.

So far, we have seen that events can be categorized into two groups that cause different behavior. The first group contains behavioral events that trigger a scenario change. The second group contains fault events that are of the form $(tag, n)$, where $tag$ denotes the fault type, and $n$ the affected network. Fault events only change the mapping of the virtual architecture $\mathcal{V}^A$ onto the physical architecture $\mathcal{A}$, but not the mapping of the applications onto the virtual architecture $\mathcal{V}^A$. Consequently, each controller consists of two components, see Fig. 6. The first component is responsible to handle behavioral events and ensures the execution semantics. It is just aware of the virtual architecture $\mathcal{V}^A$, i.e., it generates commands towards $\mathcal{V}^A$. The second component processes the fault events and redirects the commands to the corresponding physical network.

Next, we detail the procedure of an **interlayer** controller when it receives a fault event. To this end, we suppose that controller $v^c$ belongs to network $n^{(k)}$. Once it receives a fault event of the form $(fault, n^{(l)})$, it executes the procedure outlined in Algorithm 3. If $n^{(l)}$ is not a subordinate network of $n^{(k)}$, i.e., $l \neq k - 1$, $v^c$ has only to reinstall the affected channels (Lines 9–11). Otherwise, if $n^{(l)}$ is a subordinate network of $n^{(k)}$, i.e., $l = k - 1$, $v^c$ has handle the fault by migrating all processes mapped onto the faulty network $n^{(l)}$ to a spare physical network (Lines 2–8).

As a fault can be handled without additional mapping optimization, the system has a high responsiveness to faults. In case that a network $n^{(l)}$ is not anymore faulty, it sends a reintegration event of the form $(available, n^{(l)})$ to the controller, which marks $n^{(l)}$ as a spare network.
Algorithm 3: Pseudocode to handle a fault event \((\text{fault}, n^{(l)})\) under the assumption that controller \(v^c\) belongs to network \(n^{(k)}\).

\[
\begin{align*}
\text{Input:} \ & \text{fault event } (\text{fault}, n^{(l)}) \\
01 \ & n^c \leftarrow \text{virtual network mapped onto } n^{(l)} \\
02 \ & \text{if } l = k - 1 \text{ then } \ n_{\text{map}}^{(l)} \text{ is subordinate of } n^{(k)} \\
03 \ & \text{if } n^{(k)} \text{ has a spare subordinate network } n_{\text{s}}^{(k)} \text{ of the same type as } n^{(l)} \text{ then } \\
04 \ & \text{migrate } n^c \text{ to } n_{\text{s}}^{(k)} \\
05 \ & \text{else } v^c \text{ is unable to handle the fault, reports } n^{(k)} \text{ as faulty} \\
06 \ & \text{send } (\text{fault}, n^{(k)}) \text{ to superior network and return} \\
07 \ & \text{end if} \\
08 \ & \text{end if} \\
09 \ & \text{for all } q \in Q \text{ that connect a } v \text{ mapped onto } n^v \text{ with a } v \text{ mapped onto a physical core } c \in n^{(k)} \text{ and } c \notin n^{(l)} \text{ do} \\
10 \ & \text{reinstall } q \\
11 \ & \text{end for} \\
12 \ & \text{if } \exists q \in Q \text{ that connects a } v \text{ mapped onto } n^v \text{ with a } v \text{ mapped onto a physical core } c \notin n^{(k)} \text{ then} \\
13 \ & \text{send } (\text{fault}, n^{(k)}) \text{ to superior network} \\
14 \ & \text{end if}
\end{align*}
\]

6. EXPERIMENTAL RESULTS

In this section, we provide evaluation results by means of a prototype implementation of DAL. The goal is to demonstrate that a) the hierarchical control mechanism has low overhead, b) the proposed scenario-based model of computation enables the design of complex embedded systems, and c) the proposed hybrid mapping optimization strategy outperforms static mapping approaches and results in a maximum utilization that is close to the one of the optimal (local) mapping.

6.1 Control Mechanism

To measure the overhead of the hierarchical control mechanism, we developed a prototype implementation of DAL targeting an Intel i7-2720QM processor with four cores running Linux. The system is configured to form an architecture with three communication layers and only one tile so that the hierarchical control system consists of two controllers. The workload between the two controllers is split so that the MASTER controller is aware of the applications and the SLAVE controller is responsible for installing and removing processes and FIFO channels. We selected a different splitting as described in Section 5.2 to individually measure the overhead generated by the behavioral dynamism and by the interaction with the operating system.

The application set consists of the fullload application and the pulse application. The fullload application computes a predefined set of operations before stopping. Therefore, its execution time only depends on other processes running on the same core. The pulse application sleeps for a certain time interval, the so-called switching time. Then it sends an event to the run-time manager that tells the controller to stop and restart the pulse application. Each application is mapped onto a POSIX thread and scheduled by the operating system’s scheduler. The overhead of the control mechanism is estimated by comparing the absolute execution times of the fullload application for different mappings, see Table 2 for the detailed mapping configurations.

In Fig. 8, the absolute time to execute the fullload application is compared for four mapping configurations and different switching times. The switching time defines the

<table>
<thead>
<tr>
<th>Mapping</th>
<th>Core 0</th>
<th>Core 1</th>
<th>Core 2</th>
<th>Core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>M</td>
<td>s</td>
<td>fullload</td>
<td>pulse</td>
</tr>
<tr>
<td>B</td>
<td>M</td>
<td>s</td>
<td>fullload</td>
<td>-</td>
</tr>
<tr>
<td>C</td>
<td>M</td>
<td>s</td>
<td>fullload</td>
<td>s</td>
</tr>
<tr>
<td>D</td>
<td>M</td>
<td>s</td>
<td>fullload</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 8: Comparison of the time to execute the fullload application in different mapping configurations. The different mapping configurations are detailed in Table 2.
work of the VCD application is depicted in Fig. 10 and the three different video decoder applications are summarized in Table 3. The motion JPEG (MJPEG) decoder is able to decode a certain number of frames in parallel. In particular, the ss (“split stream”) process reads the video stream from a playout buffer and dispatches single video frames to subsequent processes. The sf (“split frame”) process unpacks and predicts DCT coefficients so that the dec (“decode”) process can decode one DCT block per activation. Finally, the mf (“merge frame”) process collects the DCT blocks, and the ms (“merge stream”) process collects the decoded frames.

All three video decoders read their playout buffers at a constant rate of 25 frames/second. The maximum execution time of a process, and the data volume per time unit and channel has been determined by running the applications on Intel’s SCC processor with as in Section 6.3.

6.2.2 Mapping Optimization

Next, we show how the hybrid mapping optimization strategy compares to other mapping strategies. To this end, we extended the PISA framework [2] to solve the mapping optimization problem proposed in Section 5.1. In particular, PISA is extended to calculate the collection $M$ of optimal mappings so that exactly one mapping $m$ of this collection is valid for each pair of an application and a scenario. Violations of the bandwidth constraints are avoided by imposing a big penalty on the maximum utilization. PISA solves the mapping optimization problem by either generating 1000 random solutions and selecting the best of them as overall solution, or using the evolutionary algorithm (EA) SPEA2 [27].

In the following, we compare the performance of four different mapping strategies when minimizing the maximum core utilization for different numbers of available cores:

- The **dynamic optimal** mapping strategy represents the hybrid design-time / run-time mapping optimization strategy solved using EAs.
- The **dynamic random** mapping strategy represents the hybrid design-time / run-time mapping optimization strategy solved by selecting the best of 1000 random solutions.
- The **global optimal** mapping strategy calculates a single static mapping for the system, i.e., it does not make use of the different execution scenarios. The **global optimal** strategy is solved using EAs.
- The **local optimal** mapping strategy calculates a single mapping for each scenario. Individually calculating a single mapping for each scenario might lead to the situation where an application has a different mapping in two connected scenarios, thus, the **local optimal** strategy requires run-time support for process migration. EAs are used to solve the **local optimal** strategy.

The results of this comparison are plotted in Fig. 11 for the scenarios $SD / VCD$ and $HD$. Utilizations larger than one imply that the mapping strategy is unable to find a schedulable mapping for the considered number of available cores.

As expected, the **local optimal** mapping strategy reduces the maximum utilization the most as it calculates the local optimal mapping for just one scenario. However, the unavoidable run-time support for process migration leads to non-negligible costs in terms of time and system overhead. The hybrid design-time / run-time mapping optimization strategy, i.e., the **dynamic optimal** mapping strategy, results for the $SD / VCD$ and $HD$ scenarios in a utilization that is on average 0.01 and 0.05 larger than the utilization calculated by the **local optimal** mapping strategy.

PISA is unable to find a valid mapping for the $HD$ scenario when the **global optimal** mapping strategy is used and less than 39 cores are available. On the other hand, PISA is already able to find a valid mapping for 30 cores and the $HD$ scenario when the **dynamic optimal** mapping strategy is used. Compared to the **global optimal** mapping strategy, the **dynamic optimal** mapping strategy reduces the utilization on average by 0.51 for the $SD / VCD$ scenario and 0.16 for the $HD$ scenario.

### Table 3: Configuration of the three video decoder applications of the PiP software.

<table>
<thead>
<tr>
<th>app</th>
<th>resolution</th>
<th>pixels / frame</th>
<th># processes</th>
<th># chan.</th>
</tr>
</thead>
<tbody>
<tr>
<td>HD</td>
<td>1280 × 720</td>
<td>921600</td>
<td>98</td>
<td>128</td>
</tr>
<tr>
<td>SD</td>
<td>720 × 576</td>
<td>414720</td>
<td>50</td>
<td>64</td>
</tr>
<tr>
<td>VCD</td>
<td>320 × 240</td>
<td>76800</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>
As the dynamic optimal mapping strategy does not only optimize a single scenario, the utilization might be increased with the number of available cores. For example, the maximum utilization of scenario HD is slightly increased when moving from 32 to 33 available cores. Finally notice that the selection of the solver has a high influence on the performance of hybrid design-time / run-time mapping optimization strategy. Selecting the best of 1000 random solutions might even result in a performance that is worse than the global optimal mapping strategy.

6.3 Real World Deployment

Based on Intel's SCC processor, we discuss a prototype implementation of the scenario-based model of computation. The goal is to evaluate the effort of a real world deployment of both the hierarchical control mechanism and KPNs according to the presented semantics. As a Linux operating system is running on each core of the SCC processor, a separate slave controller is assigned to each core to ensure the communication between the hardware, the operating system, and the run-time manager. In addition, the master controller is running on a dedicated core. Processes are mapped onto POSIX threads and scheduled by the operating system’s scheduler. Inner-core communication is realized by local FIFO buffers and the RCKMPI library [5] is used for inter-core communication. Depending on the mapping, the controllers decide at run-time which communication type is suitable. As controllers are just KPN processes, no additional software is required to run them. All KPN processes are stored as shared objects and loaded on request of a slave controller. In addition, SLAVE controllers have the ability to destroy running processes, and to install and remove communication channels.

The compiled code requires about 26 KB and 275 KB of static memory for the SLAVE and MASTER controller on Intel’s SCC processor. This shows that the run-time system only introduces a small overhead in terms of memory usage.

7. CONCLUSION

In this paper, we proposed the distributed application layer (DAL), a scenario-based design flow for mapping streaming applications onto heterogeneous on-chip many-core systems. Applications are modeled as Kahn process networks and a finite state machine is used to specify different execution scenarios. Behavioral events generated by either running applications or the run-time system trigger transitions between scenarios. The proposed mapping optimization strategy consists of two components. The design-time component calculates for each application a set of optimized mappings individually valid for a subset of scenarios. At run-time, hierarchically organized controllers monitor events, choose an appropriate mapping, and finally execute the required actions. We demonstrated that the hybrid design-time / run-time mapping optimization strategy outperforms global mapping optimization strategies and that the proposed scenario-based model of computation enables the design of complex embedded systems.

Acknowledgments

This work was supported by EU FP7 project EURETILE under grant number 247846.

References


