# AdaPNet: Adapting Process Networks in Response to Resource Variations

Lars Schor, Iuliana Bacivarov, Hoeseok Yang[♯], and Lothar Thiele
Computer Engineering and Networks Laboratory, ETH Zurich, Zurich, Switzerland
[♯]Dept. of ECE, Ajou University, Suwon, South Korea
firstname.lastname@tik.ee.ethz.ch hyang@ajou.ac.kr

## ABSTRACT

A widely considered strategy to prevent interference issues on multi-processor systems is to isolate the execution of the individual applications by running each of them on a dedicated virtual guest machine. The amount of computing power available to a single application, however, depends on the other applications running on the system and may change over time. A promising approach to maximize the performance under such conditions is to adapt the application's degree of parallelism when the resources allocated to the application are changed. This enables an application to exploit not more parallelism than required, thereby reducing inter-process communication and scheduling overheads. In this paper, we introduce AdaPNet, a run-time system to execute streaming applications, which are modeled as process networks, efficiently on platforms with dynamic resource allocation. AdaPNet responds to changes in the available resources by first calculating a process network that maximizes the performance of the application on the new resources. Then, AdaPNet transparently transforms the application into the alternative network without discarding the program state. Targeting two many-core systems, we demonstrate that AdaPNet outperforms comparable run-time systems, which do not adapt the degree of parallelism, in terms of speed-up and memory usage.

## Categories and Subject Descriptors

C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems; D.1.3 [**Programming techniques**]: Concurrent programming—*parallel programming*

## General Terms

Algorithm, Design, Performance

## Keywords

Streaming applications, process networks, run-time adaptivity, optimization, multi-processor systems

## 1. INTRODUCTION

Modern embedded systems offer a tremendous amount of processing power that enables multiple applications to share the system. This trend comes with the need for advanced strategies to manage the allocation of resources between applications. A widely considered strategy is to isolate the execution of the individual applications by running each of them on a dedicated virtual guest machine [4, 15]. A virtualization manager, commonly referred to as a hypervisor, is in charge of managing the computing power and allocating resources for each application. The amount of computing power available to an application, however, depends on the other applications running on the system and may change over time. For instance, if an application stops, the hypervisor may reallocate the released resources to another application whose performance can thereby be improved. We particularly assume that the number of processing elements (PEs) available to an application can change over time.

A promising approach to maximize the performance under such conditions is to adapt the application when the available resources change. However, if the application is statically specified at compile-time, the system can only respond to resource variations by adapting the mapping of parallel processes to PEs. This limits the maximum number of PEs that an application can simultaneously utilize to the number of processes. On the other hand, if resources are withdrawn so that parallel processes must share the remaining PEs, inter-process communication and scheduling overheads are likely to limit the performance.

To overcome these limitations, the application's degree of parallelism must be adapted at run-time. Specially, we address this challenge in the context of streaming applications, which are modeled as process networks. In fact, previous works showed that large performance gains in terms of throughput [17, 20] or energy consumption [2] can be obtained if the process network is refined at compile-time.

However, compared to compile-time strategies, adapting the application's degree of parallelism at run-time involves not only the calculation of an alternative process network in limited time, but also the transparent transformation of the application into this network. These steps are not trivial if stateful processes are executed asynchronously and the amount of data being produced or consumed by a process is not known beforehand. Previous works, therefore, limited their approaches to programming models with statically specified data production and consumption rates [2] or to the replication of stateless processes [3].

In this paper, we discuss the efficient execution of streaming applications on (virtual) platforms with dynamic resource allocation. We propose AdaPNet, an adaptive run-time system to execute stateful process networks on multi-processor platforms. It responds to resource variations by first calculating an alternative process network that maximizes the end-to-end throughput while preserving the original functionality. Afterwards, it transforms the application into this network without discarding the program state.

Instead of adhering to a static programming model, thereby limiting the possible structural changes to the replication of stateless processes, we specify applications as expandable process networks (EPNs) [17]. EPNs extend conventional Kahn process networks (KPNs) [10] in the sense that several process networks with different degrees of parallelism are abstracted in a single specification. In particular, an application specified as an EPN has a top-level process network defining the initial network. The initial network can be refined by hierarchically replacing stateful processes by other process networks. This enables AdaPNet to explore different degrees of parallelism automatically by expanding and contracting the network. We implemented AdaPNet for the distributed application layer (DAL) [16] that allows to execute applications adhering to the KPN and EPN model of computation (MoC) efficiently on multi-processor systems. Targeting two many-core platforms, we show that AdaPNet outperforms a comparable run-time system, which does not adapt the degree of parallelism, in terms of speed-up and memory usage. Moreover, measurements of the throughput during the transformation show that AdaPNet is able to transform an application seamlessly into an alternative process network so that its throughput is never lower than that at the beginning and end of the transformation.

The contributions of this paper are summarized as follows:

- We propose a novel run-time algorithm that reacts to changes in the available resources by determining an alternative process network and its mapping to the available resources so that the throughput is maximized.
- We propose a novel technique that transparently transforms the application into an alternative process network without discarding its program state.
- We integrate the proposed concepts into AdaPNet, a run-time system to execute streaming applications on (virtual) platforms with dynamic resource allocation.

The paper continues with a description of the considered problem and the approach to solve it. Afterwards, in Section 3, the application model is revised. In Section 4, the proposed run-time system is detailed. The results of the performed case studies are presented in Section 5. Finally, we review related work in Section 6.

## 2. PROBLEM AND APPROACH

In this paper, the efficient execution of stateful streaming applications on (virtual) platforms with dynamic resource allocation is considered. In particular, we assume that the number of PEs available to a single application dynamically depends on the presence of other applications. The goal is to maximize the end-to-end throughput of the application for all possible resource allocations.

To achieve this goal, the application's degree of parallelism and its mapping must be revised after a change in the available resources. This requires that *a*) the application has multiple implementations, all with the same functionality but with potentially different degrees of parallelism, and *b*) the application is able to switch from one implementation to another one without discarding the program state.

Both requirements come with their own challenges. On the one hand, calculating a different implementation for each possible resource allocation is impractical at compile-time, in particular, if the number of possible resource allocations is large. Therefore, alternative implementations must be calculated at run-time. On the other hand, transforming a stateful application into another implementation requires the specification of how the application's state is transferred from one implementation to another one. However, this is usually not trivial as the following example shows.

EXAMPLE 1. *Assume that an application has two implementations with different degrees of parallelism. Impl. 1 consists of process $v_1$ and impl. 2 consists of processes $v_2$ and $v_3$ that are connected by FIFO channel c. If enough hardware parallelism is available, the application might be transformed from impl. 1 to impl. 2. One possible way to do so is as follows: $v_1$ is stopped immediately and the new processes and channels are installed. To maintain the program state, a transformation procedure would be used that takes as input the program counter and all variables of $v_1$ and generates the program counters and the variables of $v_2$ and $v_3$, as well as the content of channel c. Once variables and program counters have been assigned, $v_2$ and $v_3$ are started. Programming such a procedure is typically complicated and it is even more laborious to derive a procedure that performs the opposite operation, i.e., generates the variables and the program counter of $v_1$. However, such a procedure is needed to transform the application back to impl.* 1*, e.g., if resources are taken away from the application.*

To tackle the above challenges, we propose a solution outline as follows:

- We define a high-level application-programming interface (API) to specify applications according to the EPN MoC, i.e., we specify an application as a process network that can be refined by hierarchically replacing processes by other process networks (so-called refinement networks).
- We design a run-time system that responds to variations in the assigned resources by calculating an alternative network and transforming the application into this network.
- We stepwise transform the application into the alternative network. In each step, we replace either a process by its refinement network or the processes and channels of a refinement network by their origin process. We call the first operation *expansion* and the second one *contraction.*
- We restrict the points in time for expansion and contraction: A process / refinement network must reach a normal state in order to be expanded / contracted.
- We describe a scheduling strategy that brings a process or refinement network to a normal state.
- We extend the API by two procedures that transform the state of a process into the state of its refinement network, and vice versa. Due to these procedures, a stateful process can be replaced transparently by its refinement network.

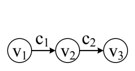
Considering the concepts and constraints described above, the application specification and the proposed run-time system are described in the following sections.

## 3. APPLICATION MODEL

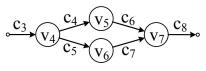
In this paper, we represent an application as an EPN [17], i.e., the application is specified as a top-level process network, which can be refined by hierarchically replacing processes by other process networks. In this section, we formally specify the application model. First, we discuss the semantics of EPNs. Then, we describe the considered execution model and propose a high-level API for EPNs.

### 3.1 Specification and Refinement Strategies

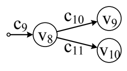
*Application specification.* The base element of an EPN is the process network. A process network is a network of autonomous processes, which can only communicate through



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



(b) Refinement network of $v_2$.



(c) Refinement network of $v_3$.

**Figure 1: Example specification of EPN $N$.**

unbounded point-to-point FIFO channels. Formally, a process network is a tuple $n = \langle V,\ C_{int},\ C_{in},\ C_{out} \rangle$ where $V$ is the set of processes and $C_{int}$, $C_{in}$, and $C_{out}$ are the sets of internal, input, and output channels, respectively. For every $c \in C_{int} \cup C_{in}$, there is one $v \in V$ that reads from it and for every $c \in C_{int} \cup C_{out}$, there is one $v \in V$ that writes to it.

The functionality of a process is specified by a deterministic imperative procedure named FIRE that performs destructive blocking reads and non-blocking writes on the input and output channels, respectively, and that is repeatedly called by the run-time system. In addition, the process may have a refinement network that defines the functionality of the process as another process network.

Clearly, channels with unbounded capacity cannot be realized in physical implementations. However, an implementation with the same semantics can use channels with finite capacity [5, 7].

An EPN is a tuple $N = \langle P,\ u,\ l,\ p_{org} \rangle$, where $P$ is a set of process networks, $u$ is the refinement function, $l$ is the channel mapping function, and $p_{org}$ is the top-level process network from which processes may be further refined. Function $u$ maps a process $v$ to the corresponding refinement network $p = u(v) \in P$ and function $l$ maps the input and output channels of the refinement network $u(v)$ to the corresponding input and output channels of $v$.

EXAMPLE 2. *Consider the EPN* $N = \langle \{p_{org},\ p_{v_2},\ p_{v_3}\},\ u,\ l,\ p_{org} \rangle$ *shown in Fig. 1.* $p_{org}$ *consists of three processes. Two of them have a refinement network:* $u(v_2) = p_{v_2}$ *and* $u(v_3) = p_{v_3}$. *The corresponding channel mapping is defined as* $l(c_3) = c_1$, $l(c_8) = c_2$, *and* $l(c_9) = c_2$.

*Refinement strategies.* An application specified as an EPN abstracts several possible granularities in a single specification. The top-level process network can be refined by hierarchically replacing processes by their refinement network. Each process replacement results in a new process network that has the same functionality as the top-level network.

EXAMPLE 3. *Consider the EPN introduced in Fig. 1. The top-level process network can be refined by replacing process* $v_2$ *by* $p_{v_2}$ *and process* $v_3$ *by* $p_{v_3}$*, see Fig. 2.*

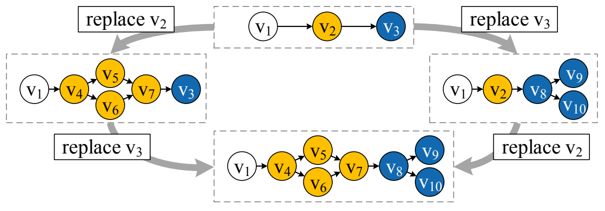


**Figure 2: Possible process networks of EPN $N$, which is specified in Fig. 1.**

## 3.2 Execution Model

As motivated in Section 2, the goal is to come up with an execution model that requires the programmer to specify only two additional transformation functions, namely one for the expansion and one for the contraction. To repeat, expansion denotes the replacement of a process by a refinement network and contraction denotes the replacement of a refinement network by its origin process. However, the programmer should not have to deal with program counters and implicit state that is on the stack when designing the transformation functions. Therefore, we restrict the points in time for the expansion and the contraction: A process can only be expanded if it has finished its FIRE procedure and a process network can only be contracted if all of its processes have finished their FIRE procedure and if its internal channels contain a statically specified number of tokens.

In the following, we discuss the considered execution model individually for channels, processes, and refinement networks. Based on that, we define the characteristics that must hold for processes that can be refined by a refinement network and for process networks that act as refinement networks so that the above stated goal can be achieved. This forms the basis for a novel transformation technique that is proposed in Section 4 and transparently transforms a process network into a new network.

*Channel.* A channel $c$ of network $n$ contains valued tokens that are read and written in FIFO order. The number and values of the tokens determine the state $s_c \in S_c$ of $c$ whereby $S_c$ is the set of channel states of $c$ that are admissible in any correct execution trace of $n$.

*Process.* During the execution of its FIRE procedure, process $v$ reads tokens from its input channels and writes tokens to its output channels, thereby modifying its state $s_v$ taken from the set of admissible states $S_v$ of $v$. As previously motivated, $v$ can only be expanded if its FIRE procedure has reached its end, but is not yet re-started. We call any admissible state of $v$ where $v$ has finished its FIRE procedure a *normal state* of $v$.

DEFINITION 1. *Assume that process $v$ is a node of a process network $n$ and $v$ either is part of a refinement network or can be replaced by a refinement network. Then, the following characteristics must hold for $v$ and $n$:*

1. *(*BOUNDEDNESS*) Assume that the input channels of $v$ contain a finite number of tokens. Then, the execution of $v$ is finite, i.e., after a bounded number of executions, its* FIRE *procedure is blocked on reading from an empty channel.*
2. *(*TERMINATION*) There exists a constant $L$ such that for any admissible state of $v$ and for any admissible states of the input channels of $v$ with $L$ tokens each, the* FIRE *procedure terminates, i.e., it reaches its end.*
3. *(*DEADLOCK FREE*) $v$ can infinitely often execute its* FIRE *procedure, i.e., network $n$ does not contain a deadlock concerning $v$.*
4. *(*NO DEAD INPUT CHANNEL*) During each correct execution trace of network $n$ of infinite length, an unbounded number of tokens is written into each input channel of $v$.*

Note that the first two conditions guarantee that the FIRE procedure never runs into an infinite execution, i.e., the FIRE procedure is not allowed to enter an infinite loop.

*Refinement network.* A refinement network $p$ has a state $s_p$, which consists of the states of all its processes and internal channels. A state $s_p \in S_p$ is admissible if it appears with a legal input sequence of $p$, whereby $S_p$ is the set of admissible states of $p$.

Refinement network $p$ can only be contracted by its origin process if all of its processes have finished their FIRE procedure and each of its internal channels contains a statically specified number of tokens. We call these numbers the *normal token distribution* of $p$ and any state of $p$ that fulfills the above stated conditions a *normal state* of $p$.

In order to transparently replace a process $v$ by its refinement network $p = u(v)$ (and vice versa), the programmer has to define how the state is passed from $v$ to $p$ and from $p$ to $v$. She does that by specifying the two transformation functions EXPAND and CONTRACT. The EXPAND function $E_v$ maps any normal state $s_v$ of $v$ to a normal state $s_p = E_v(s_v)$ of $p$. The CONTRACT function $C_p$ maps any normal state $s_p$ of $p$ to a normal state $s_v = C_v(s_p)$ of $v$. Using the above notation, we can define the required characteristics of a refinement network.

DEFINITION 2. *The following characteristics must hold for a refinement network $p = u(v)$ of process $v$:*

1. *(*BOUNDEDNESS*) Assume that the input channels of $p$ contain a finite number of tokens. Then, the execution of $p$ is finite, i.e., after executing the* FIRE *procedures of its processes a bounded number of times, each of its* FIRE *procedures is blocked on reading from an empty channel.*
2. *(*SYNTACTICAL EQUIVALENCE*) For each input and output channel of $v$, there is a corresponding input and output channel of $p$, i.e., $\forall c \in C_{in}$, $l(c)$ is an input channel of $v$ and $\forall c \in C_{out}$, $l(c)$ is an output channel of $v$.*
3. *(*FUNCTIONAL EQUIVALENCE*) Assume that the input channels of $v$ and $p$ have the same channel state, i.e., the same finite number of tokens with the same values.*
   - *If $v$ is initially in a normal state $s_v$ (i.e., the* FIRE *procedure reached its end, but is not yet re-started), if the state of $p$ initially satisfies $s_p = E_v(s_v)$, and if $v$ and all processes of $p$ then execute their* FIRE *procedure iteratively until being blocked on reading from an empty channel, the sequences of tokens written by $v$ and $p$ to the corresponding output channels are the same.*
   - *If $p$ is initially in a normal state $s_p$ (i.e., the* FIRE *procedures of all its processes are finished and each of its internal channels contains a statically specified number of tokens), if the state of $v$ initially satisfies $s_v = C_v(s_p)$, and if $v$ and all processes of $p$ then execute their* FIRE *procedure iteratively until being blocked on reading from an empty channel, the sequences of tokens written by $v$ and $p$ to the corresponding output channels are the same.*
4. *(*REACHABILITY*) There exists a constant $K$ such that for any normal state of $p$ and for any admissible state of the input channels of $p$ with $K$ tokens each, there exists an ordering of complete executions of the* FIRE *procedures*[1] *of the processes of $p$ such that $p$ has again a normal token distribution. Furthermore, the* FIRE *procedure of every process in $p$ is executed at least once in such an ordering.*

These conditions guarantee that both the expansion of process $v$ by refinement network $p = u(v)$ and the contraction of $p$ by $v$ do not change the functionality of the whole process network. Furthermore, the reachability condition states that there is at least one schedule that brings network $p$ from a normal state to another normal state.

In many situations, the above stated conditions do not impose severe restrictions. Consider a process of a video processing application. Its functionality can often be split into sub-steps or rewritten so that it first splits large data blocks into smaller data blocks and then operates on these tokens. Both described refinements do not modify the functionality, the execution is still bounded, and, if enough tokens are available in their input channels, the refinement networks can be scheduled so that they enter a normal state. Finally, note that the proposed MoC is still more general than the synchronous dataflow (SDF) [12] MoC that has been applied in previous works to adapt the application structure.

[1]The execution of a single FIRE procedure does not have to be atomic; it can be interrupted by another FIRE procedure at any time.

**Listing 1: Example of an implementation of an EPN process $v_1$, which has a refinement network $p = \langle V = \{v_2, v_3\}, C_{int} = \{c_1\}, C_{in}, C_{out}\rangle$.**

```
// process state declaration of $v_1$, $v_2$, and $v_3$
struct StateV1        struct StateV2        struct StateV3
  variables;            variables;            variables;
end struct            end struct            end struct

// initialization of state and output channels
procedure INIT(StateV1 *V1)
  V1 = initializeState();
  writeInitialTokensToOutgoingChannels();
end procedure

// behavioral description of the process
procedure FIRE(StateV1 *V1)
  manipulate(); // communication and computation
end procedure

// generate process state of $v_2$ and $v_3$, and write initial tokens
procedure EXPAND(StateV1 *V1, StateV2 *V2, StateV3 *V3,
        Channel *C1)
  V2 = generateStateOfV2(V1);
  V3 = generateStateOfV3(V1);
  writeInitialTokens(C1, V1);
end procedure

// generate process state of $v_1$
procedure CONTRACT(StateV1 *V1, StateV2 *V2, StateV3 *V3,
         Channel *C1)
  channelState = readChannelState(C1);
  V1 = generateStateOfV1(V2, V3, channelState);
end procedure
```

## 3.3 High-Level API

Based on the above discussed considerations, we propose the high-level API illustrated in Listing 1 to specify applications adhering to the EPN MoC. After a one-time initialization that is specified by the INIT procedure, the FIRE procedure is repeatedly invoked by the system scheduler. The scheduler is part of the run-time system, which is described in the next section. Procedures EXPAND and CONTRACT implement functions $E_v$ and $C_v$. Procedure EXPAND generates the state of the refinement network. Procedure CONTRACT reads the remaining tokens from all internal channels of the refinement network (the number of tokens is statically specified by the normal token distribution). Then, it generates the process state. It is the programmer's responsibility to ensure that the characteristics stated in Definitions 1 and 2 hold for all processes and refinement networks.

In addition, we specify the structure of the application in an XML format following the formal specification of an EPN. The XML document defines the processes and channels, specifies the refinement networks, and specifies the refinement and channel mapping functions. Finally, it also specifies the normal token distribution.

## 4. PROPOSED RUN-TIME SYSTEM

In this section, we propose AdaPNet, our adaptive runtime system to execute applications on platforms with dynamic resource allocation. When the resource allocation changes, AdaPNet determines an alternative process network and transforms the application into this network.

Figure 3 illustrates the execution flow. The application runs on a platform with resources A, and starts its execution with network 1 and mapping 1. If the resource allocation changes, the application is transformed into a new network and its mapping is revised. In the example, the new network and mapping are named network 2 and mapping 2. The process of transforming the application into network 2

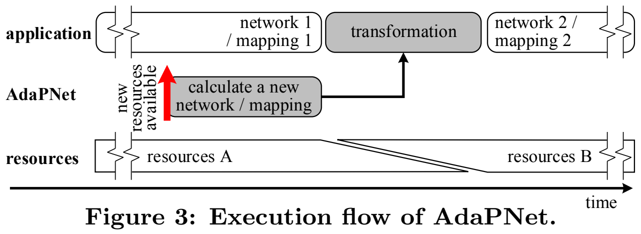


**Figure 3: Execution flow of AdaPNet.**

and mapping 2 consists of two steps. First, AdaPNet calculates an alternative process network and mapping using the algorithm detailed in Section 4.1. Then, it transforms the application into the alternative network; a technique to do so is proposed in Section 4.2. At the same time, processes whose mapping has changed are migrated. A process migration technique is summarized in Appendix A. While being transformed, the application uses resources A and B simultaneously; however, after completing the transformation, resources that are no longer needed are released.

In this paper, we aim to maximize the application's throughput. However, note that the overall approach is not restricted to this optimization goal, but can also be used if other performance metrics like, for instance, the energy consumption, are optimized.

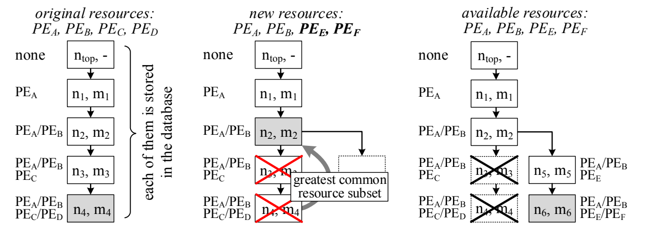
## 4.1 Alternative Process Network Calculation

AdaPNet reacts to changes in the available resources by re-calculating the process network structure in order to maximize the application's throughput while preserving the original functionality. In addition, AdaPNet must also determine a mapping for the new network. However, as the procedure is performed at run-time, AdaPNet has only limited time to react to resource variations so that the number of optimization steps must be limited.

The basic idea is to expand the application if more resources are available and to contract it if fewer resources are available. However, comprehensive strategies to contract an application are usually expensive, particularly as it is expensive to determine which process should be contracted to achieve the largest performance gain. Assume, for instance, that the processes belonging to refinement network $p = u(v)$ are distributed among the PEs. Estimating the performance gain that can be achieved by contracting $v$ involves not only the contraction, but also the calculation of a new mapping as the processes may not be evenly distributed anymore.

AdaPNet divides the re-calculation of an alternative process network, as follows. If PEs are removed or replaced, it uses backtracking to find a process network for (a subset of) the remaining PEs. Otherwise, if new PEs are available, Alg. 1 is used to expand the network so that all PEs are exploited. To enable backtracking, intermediate results obtained during the expansion of the network are stored in a database. Each entry consists of the network, its mapping, and the particular PEs for which the network and mapping have been obtained. By backtracking, the first entry in the database whose target architecture is a subset of the new PEs is identified. If the platform has multiple PEs of the same type (they are functionally identical), it only saves the PE type. Example 4 illustrates the proposed algorithm.

Example 4. *In Fig. 4, each box represents a network $n$ and mapping $m$ obtained for the resources written next to it. Assume that $PE_A$, $PE_B$, $PE_C$, and $PE_D$ have been originally assigned to the application, see Fig. 4a. Network $n_4$ and mapping $m_4$ have been obtained by stepwise increasing the number of PEs. After each step, the result has been saved in the database. Next, assume that $PE_C$ and $PE_D$ are removed and $PE_E$ and $PE_F$ are assigned to the application.*


original resources: PE_A, PE_B, PE_C, PE_D
none
n_top, -
PE_A
n_1, m_1
PE_A/PE_B
n_2, m_2
PE_A/PE_B PE_C
n_3, m_3
PE_A/PE_B PE_C/PE_D
n_4, m_4
each of them is stored in the database
new resources: PE_A, PE_B, PE_E, PE_F
greatest common resource subset
available resources: PE_A, PE_B, PE_E, PE_F
n_5, m_5
PE_A/PE_B PE_E
n_6, m_6
PE_A/PE_B PE_E/PE_F

(a) Original assignment.
(b) Phase 1: backtracking.
(c) Phase 2: calculate new network / mapping.

**Figure 4: Example of the steps to calculate an alternative network and mapping using backtracking.**

*Starting with network $n_4$ and mapping $m_4$, AdaPNet finds the database entry for $PE_A/PE_B$ as being the entry with the greatest common resource subset (Fig. 4b). Then, it calculates an alternative network and mapping for $PE_A$, $PE_B$, $PE_E$, and $PE_F$ using network $n_2$ and mapping $m_2$ as starting point. As shown in Fig. 4c, network $n_6$ and mapping $m_6$ are finally used to execute.*

Once AdaPNet has identified a certain network and mapping as the starting point for further refinements, its uses Alg. 1 to calculate an alternative network and its mapping. The basic idea of the algorithm is to use graph partitioning, which was previously proposed to find good replication degrees [8], to balance the workload between the available PEs. It first identifies the PE with the highest utilization and its process $v_{work}$ with the largest amount of work. Then, it virtually adds $v_{work}$ to all PEs and selects the PE with the lowest utilization. It stops if the ratio between highest and lowest utilization is lower than a predefined balance factor that is specified as an input to the algorithm. Selecting a good balance factor might be difficult. However, our experiments have shown that a balance factor of 1.2 generates good results in general. As extensive expansion increases the communication overhead, Alg. 1 primarily migrates processes to PEs with low utilization. In fact, $v_{work}$ is migrated to the PE with the lowest utilization if the maximum utilization over all PEs can be reduced. Otherwise, if the maximum utilization cannot be reduced anymore by migrating pro-

**Algorithm 1** Calculating a new network and mapping.

```
Input: PEs, initNetwork, initMapping, balanceFactor
Output: new process network and mapping
addProcessesToPEs(PEs, initNetwork, initMapping)
while True do
    sortPEsByUtil(PEs)                    ▷find PE with max. utilization
    maxPE ← PEWithMaxUtil(PEs)
                                 ▷find process with largest computing demand
    process ← largestProcess(maxPE)
                           ▷find PE with lowest utilization after adding process
    minPE ← PEWithMinUtilAfterAdding(PEs, process)
                                       ▷check the overall balance of the system
    if util(maxPE) < util(minPE) ∗ balanceFactor then
        finish()
    end if      ▷migrate process to minPE or expand the process
    if utilAfterAdding(minPE, process) < util(maxPE) then
        addTo(process, minPE)                          ▷migrate process
        removeFrom(process, maxPE)
    else if process can be expanded then   ▷expand process, . . .
                     ▷. . . uniformly distribute processes to maxPE, minPE
        removeFrom(process, maxPE)
        subProcesses ← expand(process)
        distributeAndAdd(subProcesses, maxPE, minPE)
    else if process is the only process assigned to maxPE then
        finish()                       ▷no more refinements are possible
    else     ▷migrate remaining processes of maxPE to minPE
        remaining ← removeAllExceptOneFrom(process, maxPE)
        addTo(remaining, minPE)
    end if
end while
```

cesses, Alg. 1 expands $v_{work}$ making the largest indivisible unit of work smaller. The new processes are distributed between the PE of $v_{work}$ and the PE with the lowest utilization so that both PEs have a balanced workload. In case that $v_{work}$ cannot be expanded, the algorithm stops or, if there are further processes assigned to the PE of $v_{work}$, it migrates these processes to the PE with the lowest utilization.

Calculating a new network might be done stepwise by executing Alg. 1 multiple times. In each step, only a subset of the new PEs is added. In Example 4, one PE has been added per step so that more entries are available in the database.

The complexity of Alg. 1 depends on the application, the number of PEs that are added, and the total number of PEs that are available. The first two parameters affect the number of iterations that must be performed and the total number of PEs determines the complexity of a single iteration. In fact, the complexity of a single iteration mainly depends on the used sorting technique (Line 3). However, as each iteration only changes the utilization of two PEs, the sorting algorithm can use the result of the previous iteration and just change the position of two entries. The complexity of an iteration is therefore $O(\#PEs)$ with #PEs being the number of PEs.

## 4.2 Application Transformation

After identifying a process network that improves the throughput of the application, AdaPNet transparently transforms the current process network into this network. AdaPNet does that in multiple steps and in each step, it either expands or contracts one process.

*Expand a process.* Assume that process $v$ is a node of network $n$. It follows from Definition 2 that the expansion of $v$ by network $p = u(v)$ does not change the functionality of $n$ if $v$ is expanded after finishing its FIRE procedure and if the state of $p$ is initially $s_p = E_v(s_v)$, whereby $s_v$ is the state of $v$ after finishing the FIRE procedure. A prerequisite for the expansion is therefore that $v$ finishes its FIRE procedure. However, it follows from the characteristics stated in Definition 1 that $v$ will do that if $n$ is executed long enough. Algorithm 2 summarizes the steps to expand a process.

**Algorithm 2** (EXPANSION) Replace process $v$ by $p = u(v)$.

```
01 install all processes and channels of $p$
02 connect the channels of $p$ to the corresponding processes of $p$
03 stop process $v$ at the end of its FIRE procedure
04 use the EXPAND procedure to generate the process states of all
   processes of $p$ and the initial tokens of all channels of $p$
05 connect incoming and outgoing channels of $v$ to the corresponding
   processes of $p$
06 start all processes of $p$ and remove process $v$
```

*Contract a process.* Assume again that process $v$ is a node of process network $n$ and that $v$ has the refinement network $p = u(v)$. It follows from Definition 2 that contracting the refinement network $p = u(v)$ by $v$ does not change the functionality of $n$ if $p$ is stopped in a normal state $s_p$ and if the state of $v$ is initially $s_v = C_v(s_p)$. $p$ is in a normal state if all of its processes have finished their FIRE procedure and if each internal channel contains a statically specified number of tokens, also known as the normal token distribution.

Therefore, a prerequisite for the contraction is that $p$ is in a normal state. However, if the FIRE procedures of its processes are executed iteratively in a greedy manner, $p$ might never enter such a state. Thus, in the following, we will first describe a scheduling strategy that executes the processes of network $n$ such that the refinement network $p$ enters a normal state. The basic idea of the scheduling strategy shown in Alg. 3 is that each process of the refinement network observes the number of tokens in its input and output channels and only starts its FIRE procedure if certain conditions are fulfilled. In fact, if a channel has more tokens than its normal number of tokens (defined by the normal token distribution), the reader process continues its execution. The writer process continues its execution if a channel has less tokens than its normal number of tokens.

**Algorithm 3** Scheduling strategy to bring a refinement network $p$ being part of process network $n$ to a normal state.

```
01 execute all processes of $n$ except those of $p$ in a greedy manner,
   i.e., execute their FIRE procedures iteratively
02 execute all processes of $p$ in a greedy manner. However, do only
   restart the FIRE procedure of a process $v$ in $p$ if at least one of the
   following conditions holds:
03   • $v$ has an internal input channel that contains more tokens than
       its normal number of tokens,
04   • $v$ has an internal output channel that contains less tokens than
       its normal number of tokens, or,
05   • the FIRE procedure of another process in $p$ is directly or indi-
       rectly blocked on an output channel $c$ of $v$. A process is directly
       blocked on $c$ if the process is the reader process of $c$. A process
       is indirectly blocked on $c$ if the process is blocked on an input
       channel of $p$ whose writer process itself is blocked by some other
       process and if this chain finally ends in $c$
06 stop if all processes of $p$ have finished their FIRE procedures and
   no FIRE procedure can be restarted
```

Note that the strategy of Alg. 3 must only be used to schedule the network when a particular refinement network is supposed to be contracted. Example 5 emphasizes the role of Line 5, which is used to resolve deadlocks that are imposed by the scheduling strategy (the processes of the refinement network might be blocked artificially, i.e., they cannot necessarily restart their FIRE procedure).

EXAMPLE 5. *Assume that process $\tilde{v}$ of refinement network $p$ must restart its* FIRE *procedure as either the rule on Line 3 or the one on Line 4 holds for one of its input or output channels. Then, $\tilde{v}$ may block on reading from another input channel that does not contain enough tokens and whose writer process is also blocked. If the writer process is also in $p$, the* FIRE *procedure of $\tilde{v}$ is directly blocked on an output channel of $v$. On the other hand, if the writer process is not in $p$, it must again be blocked by some other process as only processes in $p$ can be blocked artificially. In fact, the thereby created chain must end at some process $v$ in $p$ so that $\tilde{v}$ is indirectly blocked on an output channel of $v$ and the block can only be resolved if $v$ restarts its* FIRE *procedure.*

THEOREM 1. *Assume that network $p$ is a refinement of process $v$, part of process network $n$, and the characteristics stated in Definitions 1 and 2 hold for $v$, $p$, and $n$. After executing the* FIRE *procedures of its processes for a finite number of times, network $n$ including $p$ is scheduled according to the rules stated in Alg. 3. Then, $p$ will eventually enter a normal state.*

PROOF. We know that we originally replaced in network $n$ process $v$ by a correct refinement network $p = u(v)$. Due to Definitions 1 and 2, replacing $v$ by $p$ did not change the functionality of $n$ so that no deadlock can occur in $p$ if the FIRE procedures of all processes of $n$ are executed iteratively. Given this property, the basic idea of the proof is to observe the sequences of tokens in the input channels of $v$ in the origin network $n$ with $v$ instead of $p$. Then, we use these sequences to determine a schedule for the processes of $p$. Unless this schedule leads to a deadlock, $p$ can be executed according to this schedule even if $p$ is embedded into $n$.

Let us first consider the point in time when we started to schedule $n$ by Alg. 3. Assume that the highest number of executions of a FIRE procedure in any process of $p$ was $f$, i.e., no process in $p$ executed the FIRE procedure more than $f$ times. As all processes of $n$ are deterministic, the sequences of tokens in the input channels of $p$ are independent of the execution order of the processes. Due to the functional equivalence of the expansion, the sequences are the same as that of the origin network $n$ with $v$ instead of $p$. Let us observe the input channels of $v$ and the corresponding sequences that would have occurred if we had not done the expansion, starting from the instance of the expansion. As $v$ has no dead input channels (see Definition 1), we stop the observation if each sequence contains at least $f \cdot K$ tokens whereby $K$ is defined as in the reachability condition of Definition 2.

Let us go back to the refined network. We know from the reachability condition in Definition 2 that starting from any normal state of $p$, there exists an ordering of complete executions of the FIRE procedures of the processes of $p$ such that $p$ is again in a normal state. In such a sequence, each process in $p$ executes its FIRE procedure at least once. Now assume that we go from one such state to the next one $f$ times. Clearly, there would be enough tokens in the input channels to allow for this schedule and the FIRE procedure of each process in $p$ must be executed at least $f$ times. As defined above, no process in $p$ executed its FIRE procedure more than $f$ times when starting to schedule $n$ by Alg. 3. Therefore, in order to reach the final normal state from the current state, there are executions of the FIRE procedures still left for some processes, but no process executed its FIRE procedure more often than necessary in order to reach the final normal state (after $f$ iterations).

As all processes of $n$ are deterministic, the ordering of executing the FIRE procedures does not matter. In other words, if we start from any state and execute the FIRE procedures by a certain scheduling method a given number of times, then we can reach the same state by any other scheduling method provided that we do not execute the FIRE procedure more often than this number of times. However, no online scheduling strategy knows the number of times that the FIRE procedures should be executed in order that $p$ reaches the final normal state. Therefore, in order to prove that $p$ enters a normal state if $n$ is scheduled according to the rules of Alg. 3, we have to show that the scheduling strategy *a*) does not execute the FIRE procedure of a process in $p$ more often than the number of times that is necessary to reach the final normal state and *b*) does not lead to deadlocks, whereby $p$ initially started in a normal state and a greedy scheduler with an upper bound on the number of FIRE executions for each process in $p$ would enter the final normal state.

First, we show that Alg. 3 only starts a FIRE procedure if the greedy scheduler would also do so. Assume that an internal channel contains a smaller number of tokens than its normal number, then the writing process needs to execute its FIRE procedure at least once. Assume that an internal channel contains a larger number of tokens than its normal number, then the reading process needs to execute its FIRE procedure at least once. If the FIRE procedure in one of the above mentioned cases is blocked due to an internal input channel, the process writing to this channel must also execute its FIRE procedure. Clearly, Alg. 3 covers these cases.

Now, let us show that the scheduling strategy of Alg. 3 does not lead to a deadlock. Assume towards a contraction that there is a deadlock in $p$, i.e., the number of tokens in the internal channels does not yet correspond to the normal token distribution, but no process can proceed anymore. Consequently, at least one process in $p$ is blocked on reading from an internal or input channel of $p$ and the remaining processes of $p$ have completed their FIRE procedure, but are not eligible to restart it. However, the case that a process is blocked on reading from an internal channel is resolved as the process that causes the blocking starts its FIRE procedure. In case that a process is blocked on an input channel of $p$, the execution will only block forever if the writing process (which is not part of $p$) is blocked itself and a process of $p$ connected to an output channel of $p$ must execute its FIRE procedure at least once more to resolve this blocking. However, this case is resolved as a process that causes an indirect blocking on an output channel restarts its FIRE procedure. □

Finally, the steps to contract a refinement network are summarized in Alg. 4.

**Algorithm 4** (CONTRACTION) Replace the processes and channels of refinement network $p = u(v)$ by process $v$.

```
01 install process $v$
02 use the strategy of Alg. 3 to stop the refinement network $p$ in a
   normal state
03 use the CONTRACT procedure to generate the process state of $v$
04 connect the input and output channels of $p$ to process $v$
05 remove all processes and channels of $p$, and start process $v$
```

## 5. EXPERIMENTAL RESULTS

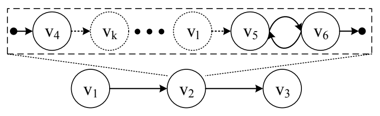
In this section, we evaluate the performance of AdaPNet. The goal is to answer the following questions: *a*) How expensive is the transformation into an alternative process network? *b*) Can AdaPNet outperform run-time systems that do not adapt the application's degree of parallelism? *c*) How does the transformation procedure affect the performance of an application during the actual transformation? — To answer these questions, we use AdaPNet to execute synthetic and real-world applications on two many-core platforms.

### 5.1 Experimental Setup

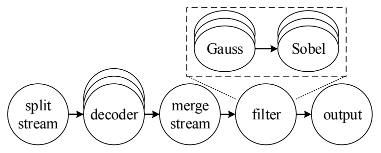
*Hardware setup.* We implemented AdaPNet on Intel's Xeon Phi and Single-chip Cloud Computer (SCC) [9]. Though not designed for embedded applications, for the purpose of the experimental evaluation, they are representatives of future many-core systems-on-chip. The Xeon Phi has 60 physical cores (the hyper-threading capability is not used), which are clocked at 1.2 GHz, and hosts a Linux operating system. The SCC has 48 cores that are clocked at 533 MHz and each of them hosts its own Linux operating system. AdaPNet runs on top of Linux and uses the POSIX library to execute multiple processes in parallel. In fact, processes are stored as dynamic libraries, which are loaded and linked dynamically when the process is started. As each core of the SCC represents an individual computing environment, channels between processes running on different cores are implemented using MPI. The implemented software synthesis tool chain follows the DAL design flow [16]. We suppose that each application is running in isolated guest machines [4, 15] so that the effectiveness of AdaPNet can be studied for each application individually.

*Benchmark applications.* We evaluate the performance of AdaPNet based on the three benchmarks shown in Fig. 5.

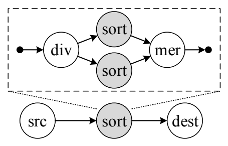
- **Synthetic.** The synthetic application has a top-level process network with three processes. Process $v_2$ has a refinement network with a variable number of processes.
- **Video-processing.** The video-processing application first decodes a motion-JPEG video stream and then applies several filters to the decoded frames.
- **Sorting.** The sorting application uses quicksort to sort arrays of 128 elements. Multiple sub-arrays can be sorted in parallel by recursively expanding the SORT process.



(a) Synthetic application.



(b) Video-processing application.



(c) Sorting application.

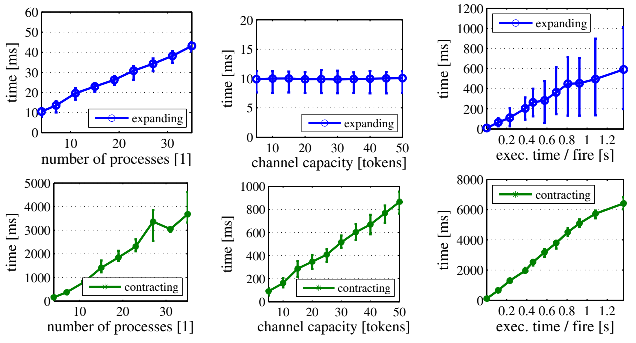
**Figure 5: Benchmark applications.**

## 5.2 Transformation Costs

To evaluate the costs of transforming an application into an alternative process network, we measure the time to expand and contract process $v_2$ of the synthetic application. When measuring the transformation time, we vary either the number of processes of the refinement network, the capacity of the channels, or the workload of processes $v_2$ and $v_5$. If not varied, the refinement network has three processes, the channels have a capacity of five tokens, and a process just reads one token from all of its input channels and writes one token to all of its output channels. For brevity, we only report the results for the Xeon Phi; however, the results for the SCC exhibit similar trends.

Figure 6 shows the time to expand and contract process $v_2$. The reported numbers are the mean of 70 runs and the bottom and top of the error bars are the 16-th and 84-th percentile. The time to expand process $v_2$ increases linearly with the number of processes of the refinement network and highly depends on the work performed per invocation of the FIRE procedure. However, the time is independent of the channel capacity as there is always only one process involved in the expansion. The time to contract process $v_2$ increases linearly with all investigated parameters. In fact, contracting takes up to ten times longer than expanding, mainly as the procedure of bringing a refinement network to an admissible state requires more steps than the procedure of bringing a single process to an admissible state.

Next, we measure the time to expand, contract, and replicate various processes of the video-processing and sorting application. Replication means that a process is replaced by a refinement network that consists of a fork process, multiple replicas of the process, and a join process. The maximum and average times measured over ten repetitions are listed in Table 1. The reported numbers confirm the trends observed with the synthetic application. Moreover, if the transformation includes multiple expand or contract operations, the



(a) Processes. (b) Channel size. (c) Workload.

**Figure 6: Time to expand and contract the process network of the synthetic application.**

**Table 1: Time in milliseconds to expand, contract, and replicate processes of two benchmarks.**

| bench-mark | transformation | Xeon Phi max | Xeon Phi avg | SCC max | SCC avg |
|---|---|---|---|---|---|
| video-processing | expand $v_{dec}$ | 151 | 78 | 667 | 440 |
| | contract $v_{dec}$ | 155 | 103 | 1054 | 893 |
| | expand $v_{filter}$ | 144 | 86 | 740 | 587 |
| | contract $v_{filter}$ | 747 | 428 | 1923 | 1852 |
| | replicate $v_{dec}$ $5x$ | 540 | 461 | 3357 | 2965 |
| | replicate $v_{gauss}$ $3x$ | 205 | 134 | 5340 | 5260 |
| | replicate $v_{sobel}$ $3x$ | 213 | 169 | 1961 | 1761 |
| | expand $v_{filter}$, replicate $v_{gauss}$ and $v_{sobel}$ $3x$ | 855 | 786 | 8017 | 7857 |
| | contract $v_{dec}$, expand $v_{filter}$ | 275 | 195 | 1423 | 1398 |
| sorting | expand $v_{sort}$ $1x$ | 15 | 8 | 115 | 104 |
| | expand $v_{sort}$ $3x$ | 73 | 61 | 505 | 470 |
| | contract $v_{sort}$ $1x$ | 93 | 51 | 52 | 42 |
| | contract $v_{sort}$ $3x$ | 596 | 425 | 795 | 675 |

measured transformation times are about the same as the sum of the times to perform the individual transformations. The results also show that installing new processes is more costly on the SCC than on the Xeon Phi. This might be due to the framework's ability to load processes dynamically from the file system, which is more costly on the SCC.

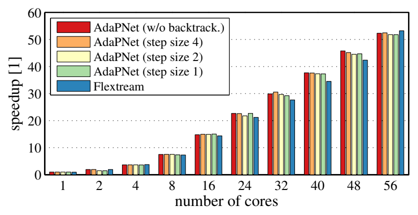
In summary, the results demonstrate that transforming an application can take up to several hundreds of milliseconds. However, the time to transform a process strongly depends on the granularity of the refinement network and the work performed per invocation of the FIRE procedure.

## 5.3 Refinement Algorithm

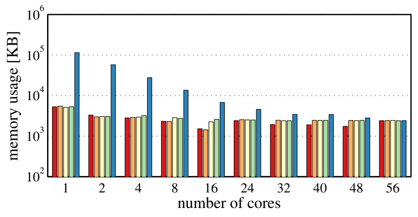
Next, we investigate the question whether AdaPNet outperforms run-time systems that only adapt the application's mapping. To this end, we compare the performance of AdaPNet with that of Flextream [8] in terms of throughput, memory usage, and time to calculate an alternative mapping and process network (if applicable). Flextream refines the process network already at compile-time by using the largest possible resource allocation as target platform. At run-time, it assigns the processes that have been originally assigned to cores that are not available, to the remaining cores. For comparability, we extended Flextream's compile-time algorithm to also support EPNs. AdaPNet calculates the new network and mapping either with or without using the ability to backtrack to previous results. If it is configured to save intermediate results in the database, it adds either one, two, or four cores per step. We call this number the *step size*. Furthermore, the balance factor of Alg. 1 is set to 1.2.

Figure 7 plots the performance of the video-processing application when executing it on the Xeon Phi with the number of available cores being varied from one to 56. The speed-up versus the throughput of the top-level process network executed on one core is shown in Fig. 7a. The speed-up achieved with AdaPNet is up to 10 % higher than that achieved with Flextream. AdaPNet achieves speed-ups close to the theoretical maximum for any investigated number of available cores. In particular, if no backtracking is used and for the maximum number of available cores, AdaPNet achieves the same speed-up as Flextream (the difference is within the limits of measurements). In all other cases, the speed-ups achieved with Flextream can be lower than that of AdaPNet if the processes cannot be evenly distributed among the available cores. This effect is particularly evident for 48 cores. AdaPNet does not suffer from this effect, as it adapts the application's degree of parallelism.

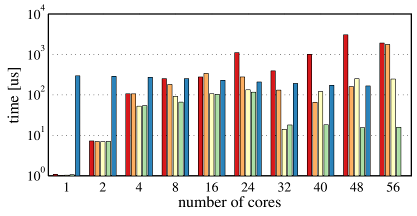
As AdaPNet adapts the number of processes to the number of available cores, the average memory usage per core is almost constant with AdaPNet, see Fig. 7b. In fact, it is



(a) Speedup.



(b) Memory usage per core.



(c) Time to calculate the new mapping and process network (if applicable)[2].

**Figure 7: Performance of a video processing application for different resource allocations.**

up to 22.5 x less than with Flextream, which uses the same process network for all possible resource allocations.

The required time to calculate the alternative mapping and process network is shown in Fig. 7c. For Flextream, we only report the time that is needed at run-time to assign the processes that have been originally assigned to cores that are not available, to the remaining cores. This is why it takes Flextream $0\,\mu s$ to calculate a new mapping when the largest possible resource allocation (56 cores) is used as target platform. The reported numbers for AdaPNet are measured when the number of cores previously allocated to the application distinguishes from the new number by the step size. For instance, for a step size of 4 and 16 available cores, we assume that the database has an entry for 12 cores. Even for tens of cores, the measured time is still in the order of a few milliseconds for both investigated run-time systems.

We conclude that AdaPNet outperforms run-time systems that do not adapt the application's degree of parallelism. Compared with Flextream, AdaPNet achieves up to 10 % higher throughput and has up to 22.5 x less memory usage. In fact, AdaPNet mainly benefits from the better suited application parallelism that reduces inter-process communication and scheduling overheads.

## 5.4 A Run-Time Scenario

The results presented so far indicate that reasonable speed-ups can be obtained by transforming the application into an alternative network. However, we have also seen that the transformation can take several hundreds of milliseconds. Next, we investigate how the performance of the application is affected during the transformation.

For this purpose, we measure the frame rate of the video-processing application when the available cores are changed every 40 s, see Fig. 8. All resource variations except the one from four to six cores cause a transformation into an alternative network. During the transformation, the frame rate basically stays between the rate at the beginning and end of the transformation. However, it can happen (e.g., when changing the number of available cores from two to five) that several frames arrive at the output process almost at the same time. This happens if multiple replicas simultaneously start to process.

Transforming the application takes the most time when

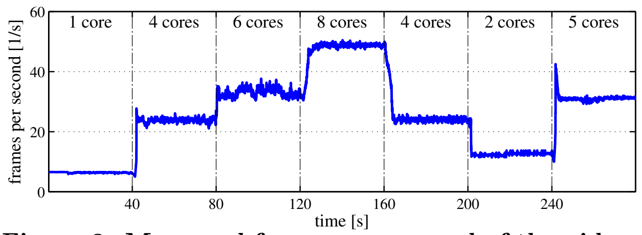


**Figure 8: Measured frames per second of the video-processing application.**

the available cores are increased from six to eight (3.9 s) and reduced from eight to four (4.2 s). In both situations, AdaPNet changes the number of replicas of the processes $v_{dec}$, $v_{gauss}$, and $v_{sobel}$. In particular, in the first situation, it replicates $v_{dec}$ three times, changes the number of replicas of $v_{gauss}$ from ten to 19, and replicates $v_{sobel}$ two times. As the application is assigned the same process network for four and six cores, AdaPNet just reverses the previous expansion when the number of available cores is reduced to four.

Overall, the results show that AdaPNet is able to transform the application into an alternative network seamlessly so that the throughput is never lower than the throughput at the beginning and end of the transformation.

## 6. RELATED WORK

Various compile-time techniques to refine a process network have been proposed in literature. Most of these techniques use the concept of fusion and fission operators [6] to change the number of replicas of stateless processes. In [20], "just-enough parallelism" is exploited by replicating processes of synchronous dataflow (SDF) graphs [12]. In [18], the throughput of an SDF graph is maximized by replicating and merging processes. A machine learning based approach to predict the ideal partition of a dataflow graph among the available resources is presented in [19].

When the available resources may change at run-time, techniques are needed that are able to adapt the mapping or even the application's parallelism. Flextream [8] is a flexible compilation framework to adapt the mapping of a streaming application dynamically. However, as shown in Section 5.3, the memory usage of the application is virtually independent of the available PEs and the application might have a considerable scheduling overhead on a single PE. Run-time task duplication is used in [3] to maximize the application's throughput. The technique replaces stateless processes by a master thread that distributes the actual work among its sibling threads. When the available PEs are changed, the number of sibling threads is increased or decreased to improve the throughput. In contrast, our work proposes the expansion of processes by process networks as a mechanism to also refine stateful processes. A dynamic scheduling approach for streaming applications specified as SDF graphs is proposed in [13]. It uses the fusion and fission operators to generate a schedule that maximizes the throughput of the application. StreaMorph [2] is a technique to adapt SDF graphs at run-time by performing a reverse sequence of executions to bring the graph into a known state. In contrast to the previous two approaches, our technique does not assume a static schedule and is therefore applicable to more complex MoCs than SDF graphs. Furthermore, the mech-

[2]For 56 cores, Flextream has in fact $0\,\mu s$ to calculate the new mapping as it refines the process network at compile-time using the largest possible resource allocation, i.e., 56 cores, as target platform.

anism proposed in Section 4.2 supports stateful processes, a key characteristic of general process networks that is not supported by the previously discussed techniques.

Finally, a different approach to achieve dynamic load balancing, the overall goal of run-time adaptivity, is the concept of task stealing [1]. Even though efficient implementations for shared-memory systems have been presented (e.g., [11]), task stealing approaches still suffer from communication overheads, in particular on distributed memory systems [14]. However, this overhead can be reduced if tasks are assigned to specific PEs. In addition, task stealing only provides limited options to exploit pipeline parallelism. In contrast, our technique uses pipeline parallelism to split large (stateful) tasks into sub-tasks.

## 7. CONCLUSION

In this paper, we demonstrated that stateful process networks can be executed on platforms with dynamic resource allocation efficiently. To achieve this goal, we proposed AdaPNet, an adaptive run-time system to execute streaming applications on multi-processor systems. AdaPNet does not exploit more application parallelism than required, thereby reducing unnecessary inter-process communication and scheduling overheads. It responds to resource variations by calculating an alternative process network that preserves the application behavior, but maximizes the performance on the new resources. Afterwards, AdaPNet transparently transforms the application into the alternative process network without discarding its program state. Evaluations on two many-core systems have shown that AdaPNet outperforms run-time systems that do not adapt the degree of parallelism in terms of speed-up and memory usage. Moreover, AdaPNet is able to transform the application seamlessly into an alternative network so that the throughput is never lower than that at the beginning and end of the transformation.

*Acknowledgement.* This work was supported by EU FP7 project EURETILE under grant number 247846 and by the UltrasoundToGo project from Nano-Tera.ch with Swiss Confederation financing.

## References

[1] R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *J. ACM*, 46(5):720–748, 1999.

[2] D. Bui and E. A. Lee. StreaMorph: A Case for Synthesizing Energy-Efficient Adaptive Programs Using High-Level Abstractions. In *EMSOFT*, pages 20:1–20:10, 2013.

[3] Y. Choi et al. Adaptive Task Duplication using On-Line Bottleneck Detection for Streaming Applications. In *CF*, pages 163–172, 2012.

[4] A. Fedorova et al. Cypress: A Scheduling Infrastructure for a Many-Core Hypervisor. In *MMCS*, pages 1–7, 2008.

[5] M. Geilen and T. Basten. Requirements on the Execution of Kahn Process Networks. In *Programming Languages and Systems*, volume 2618 of *LNCS*, pages 319–334. Springer, 2003.

[6] M. I. Gordon et al. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs. *SIGPLAN Not.*, 41(11):151–162, 2006.

[7] W. Haid et al. Efficient Execution of Kahn Process Networks on Multi-Processor Systems using Protothreads and Windowed FIFOs. In *ESTIMedia*, pages 35–44, 2009.

[8] A. Hormati et al. Flextream: Adaptive Compilation of Streaming Applications for Heterogeneous Architectures. In *PACT*, pages 214–223, 2009.

[9] J. Howard et al. A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. In *ISSCC*, pages 108–109, 2010.

[10] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In *IFIP*, pages 471–475, 1974.

[11] N. M. Lê et al. Correct and Efficient Work-stealing for Weak Memory Models. In *PPoPP*, pages 69–80, 2013.

[12] E. Lee and D. Messerschmitt. Synchronous Data Flow. *Proc. IEEE*, 75(9):1235–1245, 1987.

[13] H. Lee et al. Dynamic Scheduling of Stream Programs on Embedded Multi-core Processors. In *CODES+ISSS*, pages 93–102, 2012.

[14] S. Li et al. Asynchronous Work Stealing on Distributed Memory Systems. In *PDP*, pages 198–202, 2013.

[15] 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.

[16] L. Schor et al. Scenario-Based Design Flow for Mapping Streaming Applications onto On-Chip Many-Core Systems. In *CASES*, pages 71–80, 2012.

[17] L. Schor et al. Expandable Process Networks to Efficiently Specify and Explore Task, Data, and Pipeline Parallelism. In *CASES*, pages 5:1–5:10, 2013.

[18] A. Stulova et al. Throughput Driven Transformations of Synchronous Data Flows for Mapping to Heterogeneous MPSoCs. In *SAMOS*, pages 144–151, 2012.

[19] Z. Wang and M. F. O'Boyle. Partitioning Streaming Parallelism for Multi-Cores: A Machine Learning Based Approach. In *PACT*, pages 307–318, 2010.

[20] J. T. Zhai et al. Exploiting Just-Enough Parallelism when Mapping Streaming Applications in Hard Real-Time Systems. In *DAC*, pages 170:1–170:8, 2013.

## APPENDIX

## A. PROCESS MIGRATION

As shown in Section 4, some processes must be migrated to different PEs during the transformation. In the following, we summarize a technique to do so. It migrates a process adhering to the API described in Section 3 from one PE to another one whereby both PEs might belong to different computing environments.

Algorithm 5 shows the pseudo-code to migrate an individual process from one PE to another one on a platform with distributed memory. The basic idea of the algorithm is to have a hand-shaking protocol that informs the process to be migrated, $v_{mig}$, that no more incoming tokens are generated. Afterwards, all in-flight tokens (tokens written by a parent process but not yet received by the child process) are collected so that process $v_{mig}$, as well as all incoming and outgoing channels of $v_{mig}$, can be migrated.

**Algorithm 5** Pseudo-code to migrate process $v_{mig}$ of process network $p$ from $PE_{src}$ to $PE_{dst}$.

```
                                ▷ stop $v_{mig}$, pause parent and child processes of $v_{mig}$
stop process $v_{mig}$ before it starts a new firing
for all $c = \langle v_{src}, v_{dst} \rangle$ s.t. $v_{src} <> v_{mig}$ and $v_{dst} == v_{mig}$ do
    pause process $v_{src}$
    wait until all in-flight tokens of $c$ arrived at destination
end for
for all $c = \langle v_{src}, v_{dst} \rangle$ s.t. $v_{src} == v_{mig}$ and $v_{dst} <> v_{mig}$ do
    pause process $v_{dst}$
    wait until all in-flight tokens of $c$ arrived at destination
end for
install process $v_{mig}$ on $PE_{dst}$                ▷ move process to new PE
move process state $S_{v_{mig}}$ from $PE_{src}$ to $PE_{dst}$
remove process $v_{mig}$ on $PE_{src}$
                                ▷ re-instantiate incoming and outgoing channels
for all $c = \langle v_{src}, v_{dst} \rangle$ s.t. $v_{src} <> v_{mig}$ and $v_{dst} == v_{mig}$ do
    install channel $c$ between $PE(v_{src})$ and $PE_{dst}$
                                ▷ with $PE(v_{src})$ being the PE of $v_{src}$
    transfer tokens from old to new instance of $c$
    remove old instance of $c$
    resume process $v_{src}$
end for
for all $c = \langle v_{src}, v_{dst} \rangle$ s.t. $v_{src} == v_{mig}$ and $v_{dst} <> v_{mig}$ do
    install channel $c$ between $PE_{dst}$ and $PE(v_{dst})$
                                ▷ with $PE(v_{dst})$ being the PE of $v_{dst}$
    transfer tokens from old to new instance of $c$
    remove old instance of $c$
    resume process $v_{dst}$
end for
start process $v_{mig}$ on $PE_{dst}$ ▷ re-start the process on target PE
```