ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Semester Thesis
at the Department of Information Technology
and Electrical Engineering

# Task Migration for Multi-Processor Systems

AS 2011

Tobias Scherer

Advisors: Lars Schor
Devendra Rai
Professor: Prof. Dr. Lothar Thiele

Zurich
20th March 2012

# Abstract

The ever-increasing demand for computational power leads embedded systems to shift from single processor systems to *multi-processor systems-on-chip*. However, it is difficult to exploit the full potential for this new class of processors. In particular, dynamically reacting to changes in the environment is an open problem for distributed memory systems.

In this thesis, we propose a method for individual task migration of single processes on a distributed memory system. Task migration will allow the system to remap single processes during runtime to other processors. The designed approach allows light-weight task migration suitable for embedded systems. Furthermore, it will guarantee successful task migration independent from the network topology for *Kahn process networks*. The approach was implemented on a distributed Linux environment using the *message passing interface* as a communication framework. An enhanced case study serves as proof of concept of the proposed methods.

# Acknowledgements

First of all I would like to express my sincere gratitude to Prof. Dr. Lothar Thiele for giving me the opportunity to write this semester thesis in his research group.

I wish to express my warm and sincere thanks to my advisors Devendra Rai and Lars Schor for their many enriching discussions and their extensive support during the thesis. It was a pleasure to work with you and also to contribute to your future research.

Furthermore I would like to thank to my family, my friends and also my girlfriend Tanja for their constructive motivation and patience during this thesis.

# Contents

# List of Figures

# 1 Introduction

## 1.1 Motivation

Embedded systems are more and more executing real-time multimedia and signal processing applications, which typically require high computational power. Thus, embedded systems are shifting from single processor systems to *multi-processor systems-on-chip* (MPSoC). These architectures offer the required computing power, are small, and very efficient in terms of power consumption. However, up to now, it is still not possible to exploit the full potential of these systems. One remaining challenge is for example, to optimally utilise all processors, in terms of performance or temperature. The missing key functionality is the ability to react to environmental changes during runtime. A transparent runtime environment should be able to decide where to run processes and eventually change the process to processor assignment at runtime in case of an external event. State-preserving task migration is needed to enable this funcionality.

## 1.2 Contributions

In this semester thesis, we propose a method to migrate a single process on a distributed memory system in case of an external event. The designed approach allows light-weight task migration which is suitable for embedded systems. Furthermore, it will guarantee successful task migration independent of the network topology for *Kahn process networks* (KPN) [1]. The

approach is implemented on a distributed Linux environment using the *message passing interface* (MPI) [2] as a communication layer. A case study serves as proof of concept of the proposed methods.

## 1.3 Related Work

The considered applications are formally defined as KPN [1] which is a special class of *reactive process networks* (RPN), for which Geilen and Basten [3] discussed the operational semantics. Their implementation was based on *YAPI* [4], a programming interface to model signal processing applications as process networks.

*MPI Checkpointing* [5] allows task migration on MPI level, that is, the migration of whole virtual processor instances. This apporach lacks in flexibility, since it is not possible to migrate individual processes. Mapping each process to one processor would be the only way to circumvent this restriction. However, this would create a big overhead and is therefore not a feasible solution for embedded systems.

Kalé et al. proposed a programming language called *CHARM++* [6][7]. They also offer an communication framework called *adaptive MPI* and a memory management system called *isomalloc*. However, the proposed methods will not give any guarantee, that deterministic task migration is possible, since the migration of blocked processes is not intended.

Furthermore, the reconfiguration overhead of different task migration approaches is compared in [8].

## 1.4 Outline

In Chapter 2, the considered model of computation is introduced and the objective is defined. In Chapter 3, we describe the proposed approach for task migration. In Chapter 4, we describe the implementation of the approach followed by a case study, which serves as proof of concept. Finally, we conclude in Chapter 5, and present an outlook for further work.

# 2 Objective

The objective of this thesis is to design a method to dynamically change the assignment of processes to processors at runtime, in order to react to environmental changes. A static mapping of processes onto processors lacks in flexibility, because the assignment of processes has to be fixed at compile-time and can never change. Hence, the designated functionality is remapping of individual processes during runtime from one specific processor to another. This shall work without restarting processes, meaning, the migrated process' state has to be preserved.

## 2.1 Model of Computation

Similar to the *distributed application layer* (DAL) [9] framework, we describe an application as a KPN. These process networks are mapped onto a distributed Linux environment running on a MPSoC architecture designed for embedded systems. All processes are connected by a built-in *network-on-chip* (NoC) and there is no shared memory. On top of the NoC, the MPI is used as a communication framework.

### 2.1.1 Kahn Process Networks

Applications are modelled using the KPN specification, which describes an application as a network of concurrent autonomous processes. The communication is performed by using point-to-point FIFO channels. From now on,

the KPN is denoted as $(V, E)$, where $V$ identifies the set of processes, and $E$ the set of channels. Correspondingly, $v \in V$ and $e \in E$ denote a specific entity in the respective set. In the original specification, the `READ` semantic is described as blocking and the `WRITE` semantic is non-blocking. However, this implies either the use of unbounded memory, or the loss of tokens in case that the FIFO is full. Since there is only bounded memory in practice, the model must be refined. In order to preserve the syntax of the KPN specification, channels are bounded in size with blocking `READ` and `WRITE` semantics [10] [11]. Blocking, in this sense, means that a process stalls on the attempt of reading from an empty channel, or writing to a full channel. Only applications which show a consistent behaviour over time are considered in this thesis, i.e. there is no mechanism to deal with erroneous applications or processors.

### 2.1.2 Multi-Processor System-on-Chip Architecture

Each of the KPN processes is mapped onto one specific processor of the MPSoC architecture. The KPN edges are modelled as FIFO channels with bounded size. Without loss of generality, we assume, that we have one FIFO on both interconnected processors [12]. These FIFOs are connected over the NoC. We assume, that the NoC has some storage capacity, meaning that there might be data in transit. In our model we only allow synchronous communication schemes for the NoC.

## 2.2 Requirements

The DAL is ported to this specific model of computation. The main focus of DAL is found in embedded system applications, which yields to some important requirements concerning the task migration implementation.

A solution for task migration:

- should allow the individual migration of every single KPN process,
- has to be lightweight in terms of memory usage,
- must preserve the application in a defined state, and
- shall be transparent to the developer.

All the before-mentioned prerequisites yield to a general problem description, which is detailed in the following chapter.

## 2.3 Motivational Example

Nowadays, the reliability of embedded systems is threatened by exceeding threshold temperatures due to increasing on-chip power density. Embedded systems - as the term suggests - are usually small and integrated into a larger context. The increasing demand of performance has lead to a shift towards multi-core processors. These kind of processors provide great performance per unit area, but face various thermal issues. For example, the temperature might increase at one processor, because of unbalanced workload. The main idea suggests to distribute the load of the concerned core regions, such that the processor temperature is more equalised over the whole MPSoC area, or to temporarily move a certain task to cool down the affected region. In order to react to temperature changes, task migration is needed; it must be possible to remap individual processes during runtime. This is one specific example among many others, where task migration is an important requirement for solving various problems.

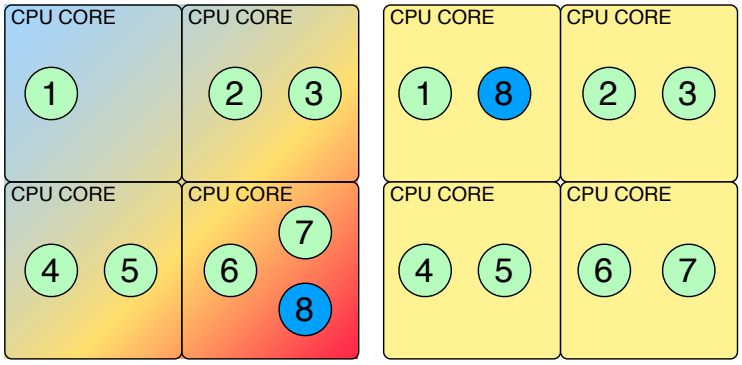



*Figure 2.1:* Temperature distribution for MPSoC is unbalanced [left] and about to exceed the threshold temperature of this chip region. Task migration of one specific process (8) allows to cool down that region and to get a more equalised temperature distribution [right].

# 3 Approach

Successful task migration of individual KPN processes (KP) requires at least that the process to be migrated ($v_{mig}$) is stopped on the original processor and finally restarted on the targeted processor. By the definition of a KPN with bounded buffer size, the successor ($V_{suc}$) and predecessor ($V_{pre}$) processes have to be blocked at their `READ` or `WRITE` statements during the time $v_{mig}$ is being migrated. The terminologies are illustrated in Fig. 3.1 and formally defined as:

$$
\begin{array}{lll}
v_{mig}: & \textit{process to be migrated} & \\
V_{pre} := & \{\ u \in V & \mid \exists\ (u, v_{mig}) \in E \quad \} \\
E_{pre} := & \{(u, v_{mig}) \in E & \mid u \in V_{pre} \quad \} \\
V_{suc} := & \{\ u \in V & \mid \exists\ (v_{mig}, u) \in E \quad \} \\
E_{suc} := & \{(v_{mig}, u) \in E & \mid u \in V_{suc} \quad \}
\end{array}
\tag{3.1}
$$

A three-way task migration approach is elaborated in this chapter. First, the setup and the preconditions are defined, and then, the subsequent task migration steps are detailed.

## 3.1 Setup

A known set of applications is mapped onto the MPSoC architecture. The applications are modelled as KPNs consisting of a distinct set of processes. Processes can either communicate over data channels with their predecessors and successors or exchange messages over a special event channel $e$ as shown in Fig. 3.2. In this thesis, only event messages from the set {*Stop, Restart,*

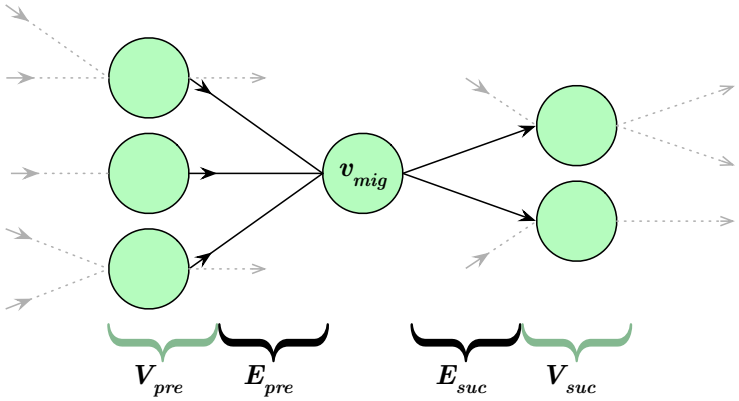



*Figure 3.1:* Partial KPN with one node to be migrated.

*Pause, Resume*} are considered. They are used to control the process model that can be illustrated as a finite state machine as shown in Fig. 3.4.

### 3.1.1 Conditions

The approach described in this chapter relies on some important assumptions:

1. All treated KPNs do not contain any loops.
2. All buffers have finite size.
3. KPNs are consistent and connected at any time.
4. Each KP can eventually release its blocked `READ` and `WRITE` statements to execute the task migration logic.

## 3.2 Stopping a KPN Process

The simplest way of stopping a KP, that is, interrupting the process between two consecutive firings, shows some major flaws. Since the definition of a KPN allows almost every reasonable topology, this approach highly depends on the fact that one firing is completed at some future point in time. Furthermore can neither timing bounds be given, nor is it possible to give a formal proof that stopping works for any KPN. Any approach dependent on the network topology is therefore discarded from the very beginning.

From $v_{mig}$'s point of view, the network topology is defined by its `READ` and `WRITE` statements, that allow the process to communicate with $V_{pre}$ and $V_{suc}$. Therefore, we are looking for the ability to stop a KP at each of the before-mentioned communication semantics. However, this feature must not

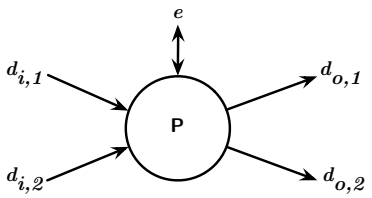



*Figure 3.2:* A single KP which has two input and two output data channels ($d_{i,x}$ and $d_{o,x}$), and one special event channel ($e$).

only allow to stop a process, but also to reenter at the corresponding statement on another processor. In our approach we use the well-established Protothreads [13] framework to enable the desired functionality. At compile time, C address labels are introduced at each of the `READ` and `WRITE` statements. The labels are named with the corresponding line number of the source code and the line number is stored within the local state of $v_{mig}$ during the stopping stage.

### 3.2.1 Releasing Channels

Task migration shall be deterministic, therefore all channels must not contain any data. Releasing the predecessor channels ($E_{pre}$) is performed by collecting all remaining data tokens as shown in Fig. 3.3. This data is eventually migrated with the process as well. The collection is only possible if we can define an upper bound on the amount of data remaining in $E_{pre}$. Hence, all processes in $V_{pre}$ have to be stopped to give the guarantee that no more tokens are written into $E_{pre}$. Though, the stopping problem then also applies to $V_{pre}$ (and could be solved as described for $v_{mig}$). In this case we would finally pause the whole network. But, in contrast to $v_{mig}$ it is not necessary to actually stop $v \in V_{pre}$, i.e. it is sufficient to pause $v \in V_{pre}$. In other words, it must only be guaranteed, that the process does not continue to write data into $d \in E_{pre}$. Using this property as a prerequisite, the upper bound of data ($N_{max}$ Bytes) can be calculated as:

$$N_{max}(d) = 2 \cdot N_{FIFO}(d) + N_{NoC}(d) \leq 3 \cdot N_{FIFO}(d) \qquad \forall d \in E_{pre} \; . \quad (3.2)$$

In Section 2.1.2 we decided to use equally sized FIFOs ($N_{FIFO}$ Bytes) at both ends of the channels. For any synchronous communication scheme, the data in transit ($N_{NoC}$) is smaller or equal to $N_{FIFO}$.
The upper bound calculated in (3.2) will allow releasing all $d \in E_{pre}$ and then collecting the remaining data tokens. The suggested proceeding is detailed in the subsequent sections.

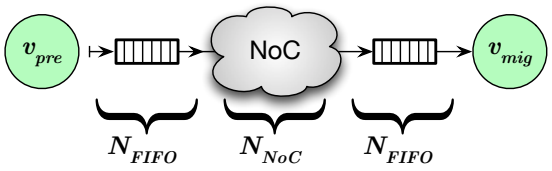



*Figure 3.3:* Disconnected KPN with remaining data tokens on the channel.

### 3.2.2 Stop Token

Once $v_{mig}$ has stopped, the processor hosting $v_{mig}$ will launch a new cleanup process ($v_{mig}^{clean}$), which is responsible for flushing the channel by collecting and storing the remaining tokens. Then $v_{mig}^{clean}$ waits for all $u \in V_{pre}$ to be paused (will be signalled over the event channel $e$ at some point in the future). On the other hand, all $u \in V_{pre}$ will ultimately write a unique data token ($ST$) onto the channel to flag the last token on each $d \in E_{pre}$. Then, $v_{mig}^{clean}$ will receive data from all $d \in E_{pre}$ and store the remaining data tokens in the memory. The memory needed is bounded by (3.2), we decided to use a backup buffer of the very same size. Once $v_{mig}^{clean}$ has received $ST$ on all channels $d \in E_{pre}$, all data has been collected. We neglected so far that $d \in E_{suc}$ might also contain data. These channels must also be flushed before migration. Luckily, we can apply the same principles which were discussed here to all $u \in V_{suc}$. However, we do not actually want to stop $u \in V_{suc}$. Important here is the discrimination between stopping and pausing of a process as denoted in Fig. 3.4. Stopping is used for $v_{mig}$. $v_{mig}$ terminates itself and will therefore not exist anymore. Pausing on the contrary will only guarantee, that a process does not continue until it is finally triggered to resume. If this discrimination cannot be done, all predecessor and successor nodes have to be stopped as well, which will ultimately result in stopping the whole KPN.

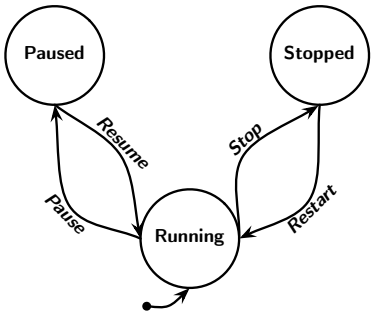



*Figure 3.4:* Processes (KP) can be either running, paused or stopped modelled with this finite state machine.

### 3.2.3 Summary of the Stopping Stage

The previous sections have explained a procedure to stop one single KP of a KPN in order to enable the migration of that process. The shown approach is independent of the network topology for the considered class of KPN. However, the proposed approach can be applied to any class of KPN if all processes are stopped to migrate a single process.
As a summary, the approach used the following subsequent steps to stop one single KP:

1. Stop the process to be migrated; Pause the predecessor and successor processes.
2. Insert a unique stop token to flag the last token on each channel.
3. Start a cleanup process to collect the remaining data tokens on all channels.
4. Receive data from channels until the unique stop token arrives.

The three basic properties which were used in this stage are:

- The ability to stop a KP at every `READ` and `WRITE` semantic and consequently restart the KP at the very same line.
- The discrimination between stopping and pausing of a process as modelled in Fig. 3.4.
- The prerequisite that the steps are subsequent, i.e. coordination happens among the KPs over the event channel $e$ defined as in Fig. 3.2.

## 3.3 Migration

In this stage, the relevant data is taken and then sent to the target processor. All preliminary steps were already done in the stopping stage. To recap shortly, the following data is at least needed to enable a deterministic restart of $v_{mig}$.

- The line number of the corresponding `READ` or `WRITE` statement, where the process has actually stopped.
- The local variables of the process.
- The buffer content of all `READ` FIFOs.

Having this data, it is possible to properly restart the KP and correctly reconnect the KPN. Therefore, the actual task migration stage consist of first storing and then transferring the before-mentioned data to the target processor.

## 3.4 Restarting

The restart stage will finally allow the KPN to continue its operation. It only takes the stored data from the previous migration stage. Since $v_{mig}$ has been migrated to another processor, the channels ($V_{pre}$ and $V_{suc}$) have to be reconnected. This is not a problem, because all of their data was collected during the stopping stage. Therefore, they can only be re-instantiated and reconnected.
The required steps are:

1. Reinitialise $v_{mig}$.
2. Restore the local variables of $v_{mig}$.
3. Refill the FIFO buffers with the data that has been collected by $v_{mig}^{clean}$.
4. Jump to the line number, where the process has stopped.
5. Reconnect the KPN ($V_{pre}$ and $V_{suc}$).
6. Restart $v_{mig}$ and resume all $u \in \{V_{pre} \cap V_{suc}\}$.

After this, the process is running from the very same statement as it has left and the application just continues its operation in a consistent behaviour as it was proposed in the objective.

# 4 Implementation

The implementation part of this thesis serves as proof of concept of the proposed task migration approach.

## 4.1 Control Hierarchy

The DAL framework elects one of the MPSoC processors as a *master*. All other processors are then called *slaves*. The *master* coordinates the *slaves* and controls the execution of KPs. In particular, the *master* decides where to start and stop processes and is also able to trigger task migration of an individual KP. Furthermore, it is the *master*'s duty to set up the channels and to reconnect them after task migration.

## 4.2 Protothreads

In the approach we assumed to have the ability to release each blocked `READ` and `WRITE` semantic. Once, these semantics are released we use the well-established Protothreads framework [13] to store the line number of the corresponding `READ` or `WRITE` statement. The release functionality must be added as an additional feature of the `READ` and `WRITE` functions triggered by a signal. Once, the process received the release signal, it must not continue its computations, but has to store the line number and terminate itself. The protothread library is used to implement the latter functionality. At compile time, C address labels are introduced at each of the `READ` and `WRITE` statements. The labels are named with the corresponding line number of the source code. The line number is then stored within the local state of $v_{mig}$

during the stopping stage.

### 4.2.1 Possible Obstacles

Protothreads enables some very neat functionalities, but it also entails some difficulties. In this section, we discuss possible obstacles and the remedies.

| **Possible Obstacles** | **Remedies** |
|---|---|
| No threads can be used within the process itself. | Instead of using threads, one should use DAL processes and let the DAL code generator decide where to run them. |
| No support for dynamic memory; i.e. no `malloc()` or `calloc()`. This also implies, that one should not use C++ objects. | Usage of a unique system-wide virtual memory address space per process. A special `DAL_malloc()` and `DAL_free()` macro could then take care of the occupied memory regions. For task migration, one would then copy these regions to another processor and insert it at the very same address. For more details, refer to the `isomalloc()` principle of CHARM++ [6][7]. |
| Data tokens cannot be read within functions. | Precompiler could resolve this. |

## 4.3 Coordinated Shutdown

For simplification, we did not discriminate between stop and pause for processes. Therefore, we are pausing the whole KPN in the prototype implementation. The stopping of the KPN requires some synchronisation among the *slaves*. In the first phase, all KPs have to terminate. Then, in the second phase, the remaining data tokens are collected. The size of this data is only bounded if the first phase, that is, stopping of all KPs, has finished at that point. Therefore, the *slaves* report to the *master* after each phase and will then be triggered again to continue to the next phase. The last confirmation will inform the *master* that stopping was successful. Finally, the *master* can trigger the next stage, namely the task migration. So it is important to have at least these two different barriers:

1. Confirmation that $V_{pre}$, $V_{suc}$, and $v_{mig}$ have been terminated.
2. Confirmation that a cleanup process $v^{clean}$ has collected all the remaining tokens on the incoming channels of each $v \in V$.

## 4.4 Case Study

We tested our task migration approach with a 5-KP MJPEG application, where each KP is assigned to a different *slave* processor. In Figures 4.1 and 4.2 the output of each KP can be seen. The whole network is started and asynchronously stopped several times. This means, that the *master* triggers each *slave* to stop its KP without any assumption on timing or ordering constraints. In the period when all KPs are stopped, the *slaves* persist as homogenous instances. Finally, after some random delay, the *master* triggers the *slaves* again to restart the KPs. In our testing we did not consider the actual migration due to the lack of time. But since the *slaves* are homogeneous, it will certainly also work after a process has been migrated to another *slave* processor.

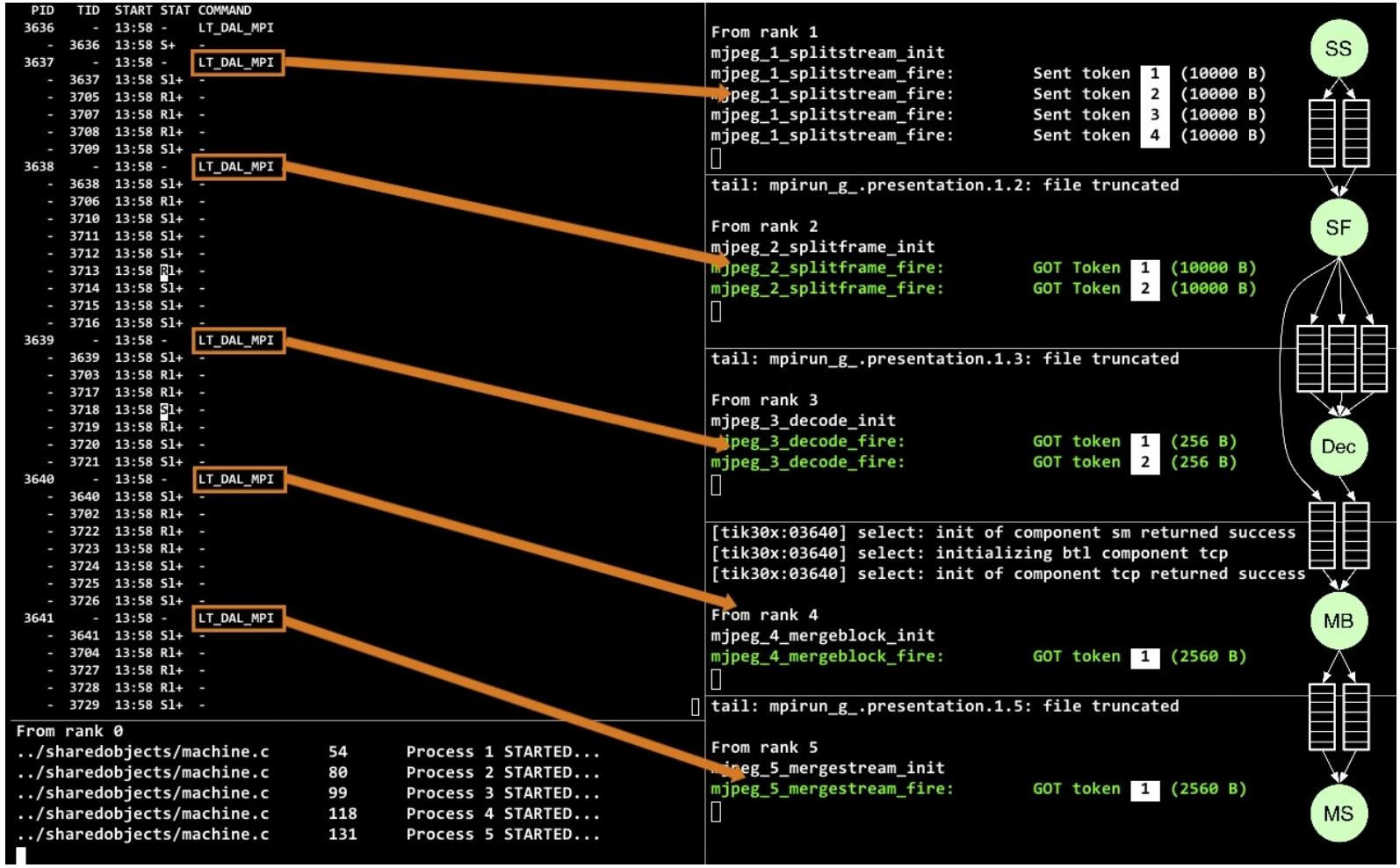



*Figure 4.1:* Proof of concept with a 5-KP MJPEG application [right]. Process table [top left] shows running KPs. All KPs run on a different virtual MPI processor on the same physical six-core machine. Therefore each of the processes in the process table corresponds to one virtual processor runtime environment.


*Figure 4.2: Master* [bottom left] triggered all KPs to stop. Process table shows that only *slave* processes are running, the KP threads are gone (compare to Fig. 4.1).

# 5 Conclusion and Outlook

## 5.1 Conclusion

This semester thesis proposed a state-preserving task migration approach for an MPSoC architecture with distributed memory. In order to guarantee task migration independent of the *Kahn process network* topology, the approach must be able to unblock processes at their `READ` and `WRITE` semantics. We proposed a method, that allows to migrate individual processes by making sure that the predecessors of the corresponding process to be migrated are not writing tokens to their channels. This allowed us to set an upper bound on the amount of memory used to collect the remaining data on the channels. A unique data token is used to flag the last token on each channel. In the migration phase, we transfer the line number, the collected data, and the state information of the process to be migrated to another processor. The process is restarted on its new processor and the application can continue the execution where it has left.
The overhead for the proposed task migration consists mainly of the line number, the buffer content, the local variables, and the overhead of the communication layer. Finally, a prototype implementation was developed to show the viability of the considered approach.

## 5.2 Outlook

There are several ways to extend the current work. First, the proposed approach can be integrated to the DAL design flow to automatically provide task migration into all DAL applications. Second, a task migration interface

can be implemented with the knowledge gained from this thesis. Third, it would be nice to have detailed performance measurements test the timing requirements. Furthermore, it would be interesting to extend the approach to any class of KPN and to formally prove the task migration approach.

# A
# List of Acronyms

| | |
|---|---|
| API | Application Programming Interface |
| DAL | Distributed Application Layer |
| FIFO | First-In First-Out |
| KPN | Kahn Process Network |
| RPN | Reactive Process Network |
| KP | Kahn Process |
| MJPEG | Motion JPEG |
| MPI | Message Passing Interface |
| MPSoC | Multiprocessor System-on-Chip |
| NoC | Network on Chip |

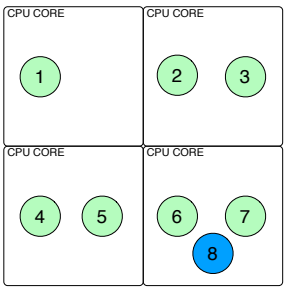
# B
# Presentation Slides

# Task Migration for Multi-Processor Systems

Semester Thesis of Tobias Scherer
Advisors: Lars Schor, Devendra Rai



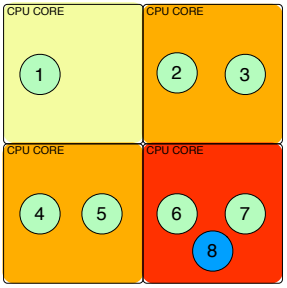
## Motivation







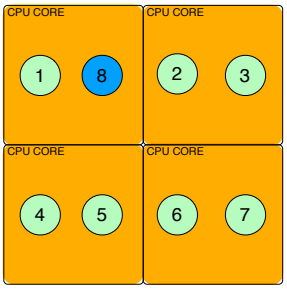

## Motivation








## Motivation







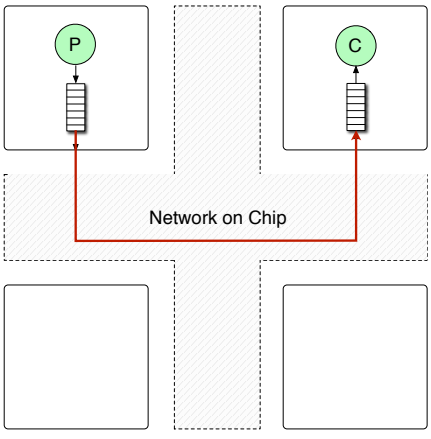

## Objective - Task Migration

- Kahn Process Network (KPN)
- Distributed Linux environment
- Message Passing Interface (MPI) for communication of KPN nodes
- No shared memory between KPN nodes

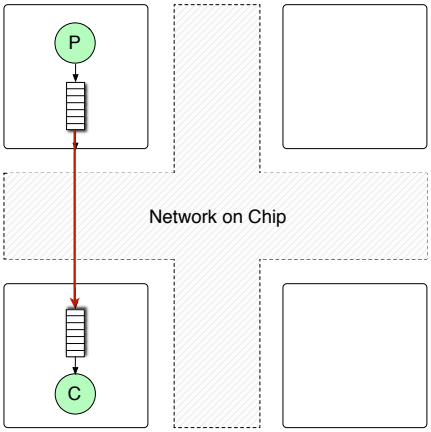




## Objective - Task Migration



## Objective - Task Migration








## Related Work

- CHARM++ with AMPI
  - ➡ Assign unique virtual address range for all processes and then copy the this segment to another processor
  - ➡ Big overhead → not suitable for embedded applications
- MPI Checkpointing
  - ➡ Migrates virtual processor instances in one piece





## Content

- Motivation and Objective
- **Basic Principles**
- **Approach for Task Migration**
- Demo
- Limitations
- Conclusion





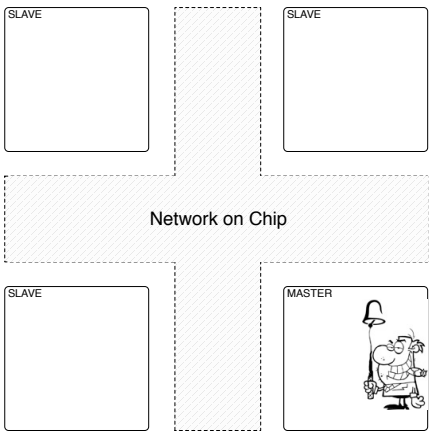
## Basic Principles

- Protothreads
  - Well established
  - Being able to stop a process at every READ or WRITE statement
- Controlled and managed stopping
  - No lost tokens
  - Coordination is crucial
  - Killing threads is no valid possibility





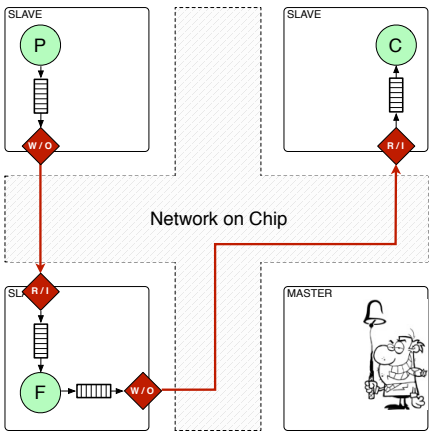
## Approach - Stop KPN



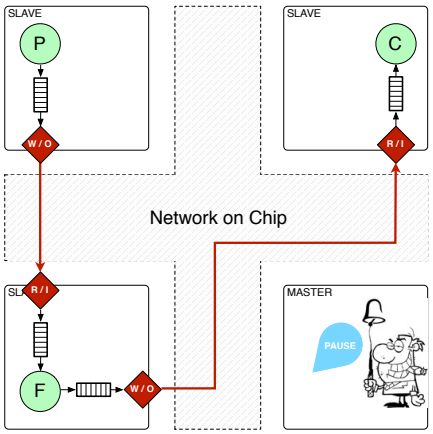





## Approach - Stop KPN



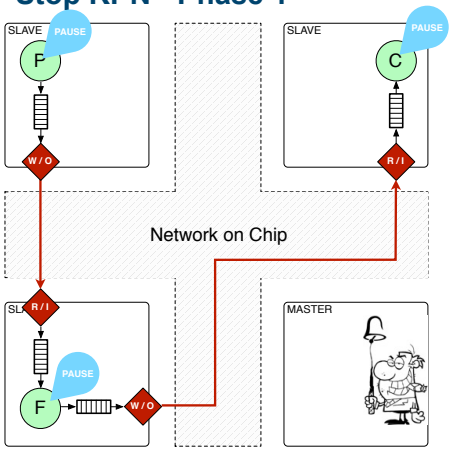
# Approach - Stop KPN - Phase 1







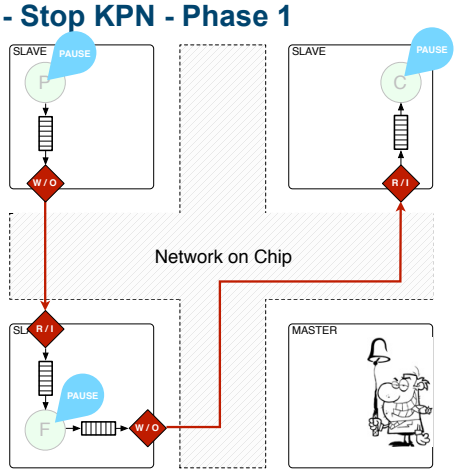

# Approach - Stop KPN - Phase 1







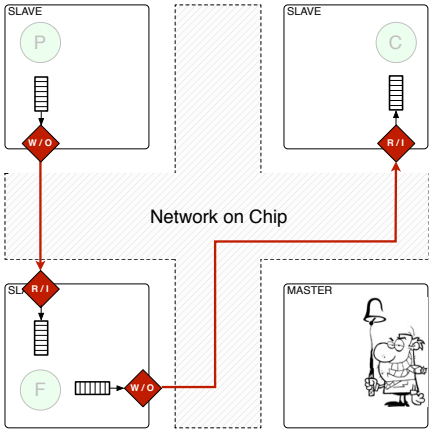

# Approach - Stop KPN - Phase 1







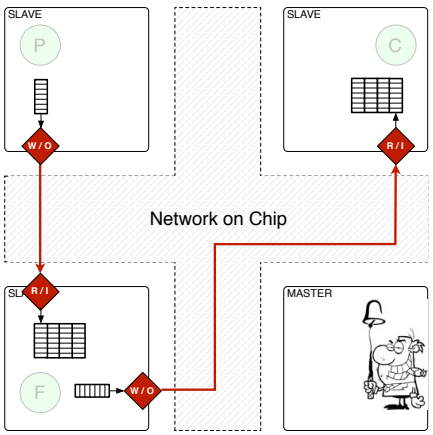

# Approach - Stop KPN - Phase 2







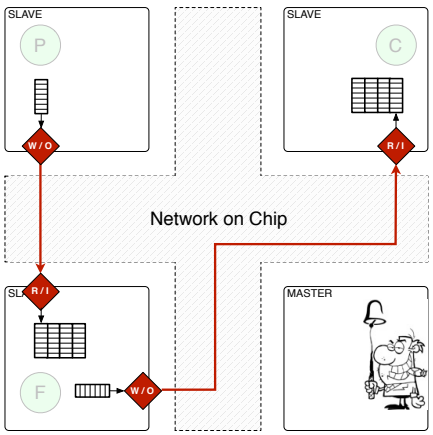

# Approach - Stop KPN - Phase 2



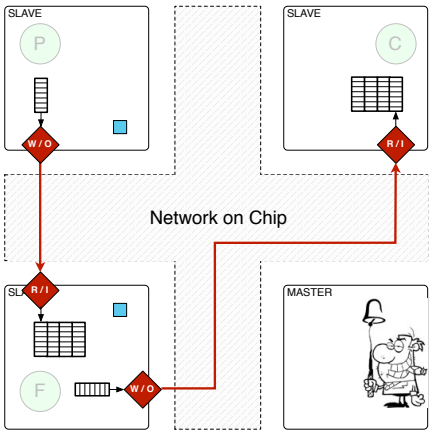





# Approach - Stop KPN - Phase 3



# Approach - Stop KPN - Phase 3






ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

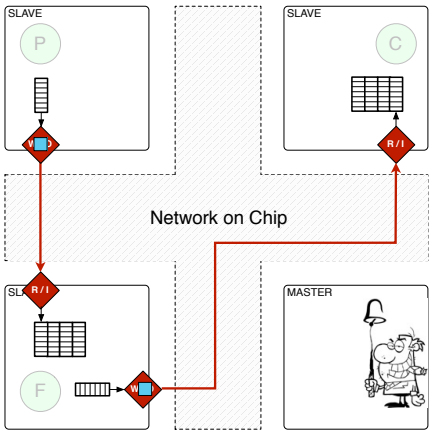
Introduction
- Motivation
- Objective
- Related work

Approach
- Basic Principles
- Stop
- Migrate
- Restart

Case study
- Demonstration

Conclusion
- Limitations
- Conclusion

# Approach - Stop KPN - Phase 3






ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

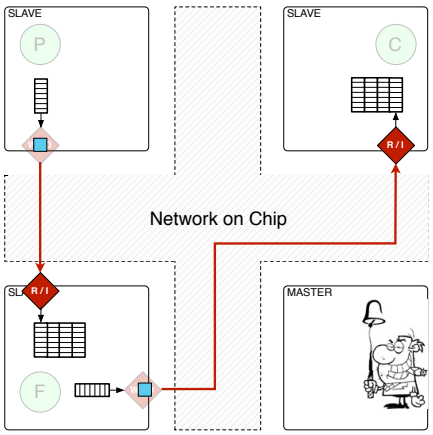
Introduction
- Motivation
- Objective
- Related work

Approach
- Basic Principles
- Stop
- Migrate
- Restart

Case study
- Demonstration

Conclusion
- Limitations
- Conclusion

# Approach - Stop KPN - Phase 3






ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

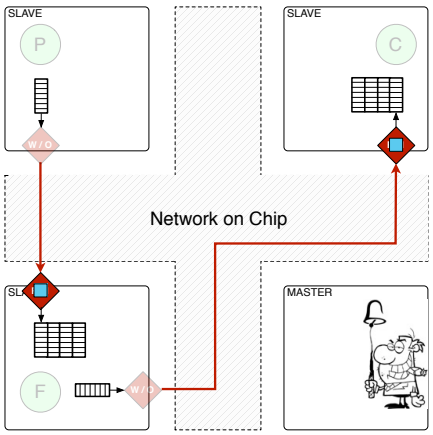
Introduction
- Motivation
- Objective
- Related work

Approach
- Basic Principles
- Stop
- Migrate
- Restart

Case study
- Demonstration

Conclusion
- Limitations
- Conclusion

# Approach - Stop KPN - Phase 3






ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

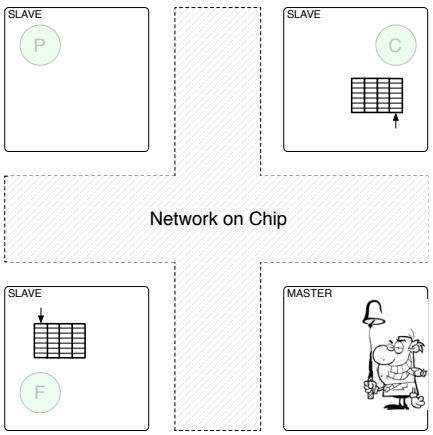
Introduction
- Motivation
- Objective
- Related work

Approach
- Basic Principles
- Stop
- Migrate
- Restart

Case study
- Demonstration

Conclusion
- Limitations
- Conclusion

# Approach - Stop KPN - Phase 3






ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

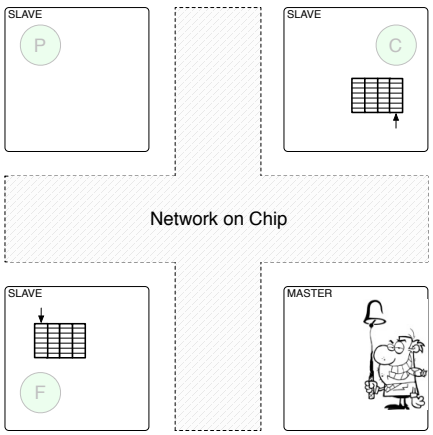
Introduction
- Motivation
- Objective
- Related work

Approach
- Basic Principles
- Stop
- Migrate
- Restart

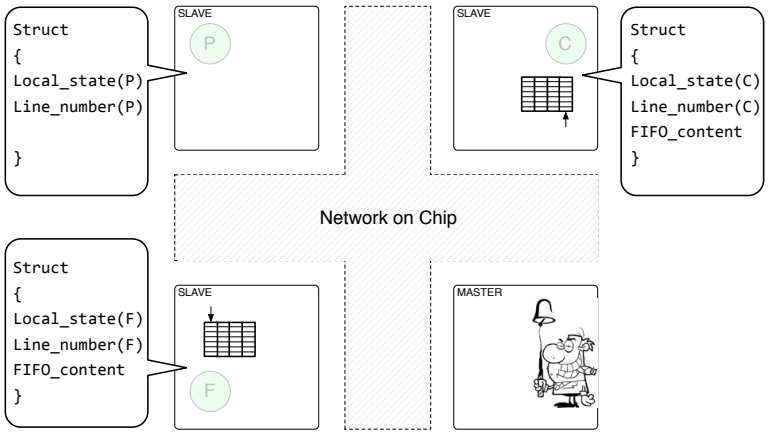
Case study
- Demonstration

Conclusion
- Limitations
- Conclusion

# Approach - Stop KPN - Phase 4



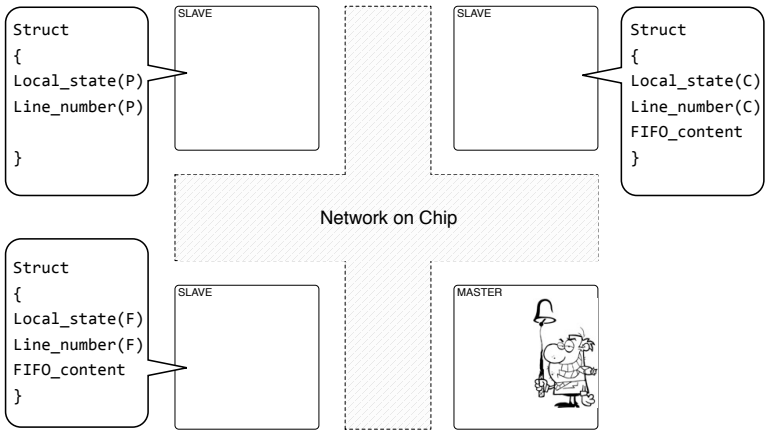
# Approach - Stop KPN - Phase 4






ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich
Introduction: Motivation, Objective, Related work
Approach: Basic Principles, Stop, Migrate, Restart
Case study: Demonstration
Conclusion: Limitations, Conclusion

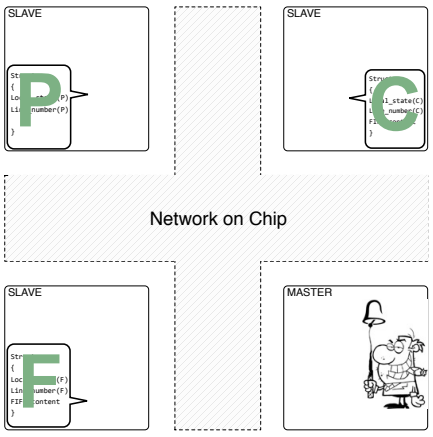
# Approach - Stop KPN - Phase 4






ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich
Introduction: Motivation, Objective, Related work
Approach: Basic Principles, Stop, Migrate, Restart
Case study: Demonstration
Conclusion: Limitations, Conclusion

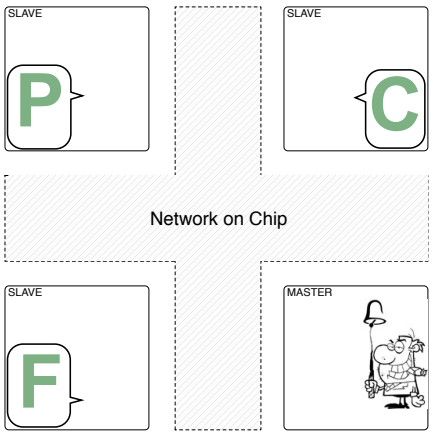
# Approach - Stop KPN - Phase 4






ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich
Introduction: Motivation, Objective, Related work
Approach: Basic Principles, Stop, Migrate, Restart
Case study: Demonstration
Conclusion: Limitations, Conclusion

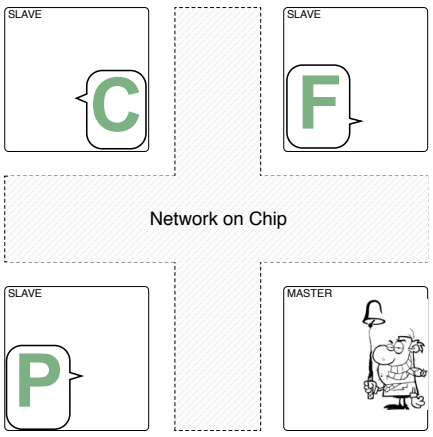
# Approach - Migration






ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich
Introduction: Motivation, Objective, Related work
Approach: Basic Principles, Stop, Migrate, Restart
Case study: Demonstration
Conclusion: Limitations, Conclusion

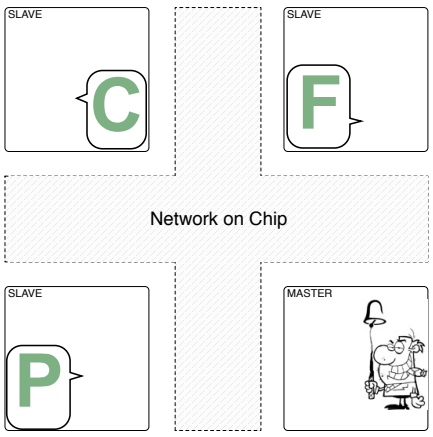
# Approach - Migration



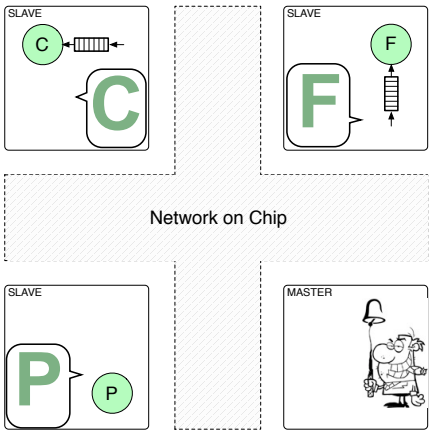



ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich
Introduction: Motivation, Objective, Related work
Approach: Basic Principles, Stop, Migrate, Restart
Case study: Demonstration
Conclusion: Limitations, Conclusion

# Approach - Restart



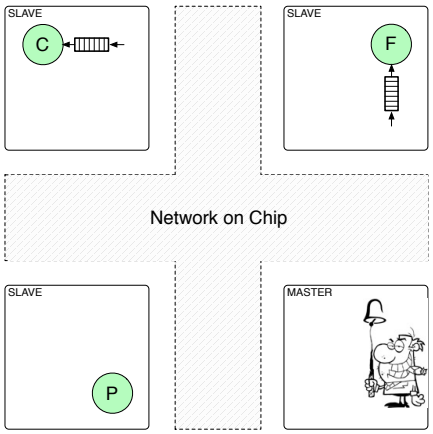
## Approach - Restart







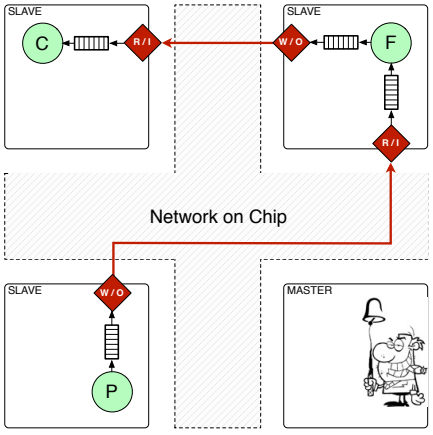

## Approach - Restart







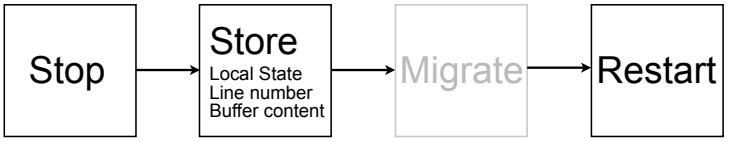

## Approach - Restart








## Case Study - Recap

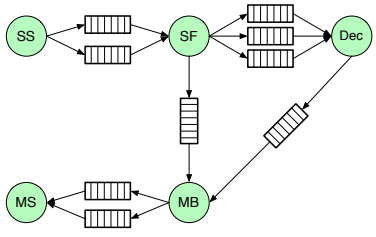







## Case Study - Implementation

- MJPEG Kahn Process Network
  - One network node per processor



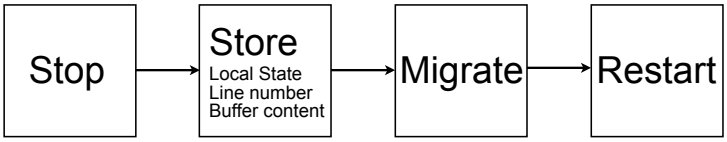





## Limitations

- No threads within KPN nodes
  - ➡ Parallelise application and let DAL code generator chose where to execute process
- No support for *malloc* / *calloc* within KPN nodes
  - ➡ Hard to deal with unknown, unbounded memory within embedded applications
  - ➡ Decision to support lightweight applications
- Tokens cannot be read or written inside functions
- Explicit RETURN statements in the process implementation are not supported
- All local variables have to be in the process state

## Conclusion




- Coordinated, controlled and managed shutdown of KPN triggered by a MASTER node.
- Ability to terminate from within a KPN process at every READ and WRITE statement.





## Prototheads

- Using C macros for the READ and WRITE functionality
- C address labels to jump to a specific line number (only at entry points)
- The FIFO *pop function* must have the ability to unblock

# Bibliography


[1] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," *Information Processing*, 1974.

[2] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*, 1999.

[3] M. Geilen and T. Basten, "Reactive Process Networks," in *Proc. ACM Int'l Conf. on Embedded Software (EMSOFT)*. ACM, 2004, pp. 137–146.

[4] E. De Kock, W. Smits, P. van der Wolf, J. Brunel, W. Kruijtzer, P. Lieverse, K. Vissers, and G. Essink, "YAPI: Application Modeling for Signal Processing Systems," in *Proc. Design Automation Conference (DAC)*. ACM, 2000, pp. 402–405.

[5] S. Sankaran, J. Squyres, B. Barrett, V. Sahay, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman, "The LAM/MPI Checkpoint/Restart Framework: System-initiated Checkpointing," *International Journal of High Performance Computing Applications*, vol. 19, no. 4, pp. 479–493, 2005.

[6] L. Kalé, B. Ramkumar, A. Sinha, and A. Gürsoy, "The CHARM Parallel Programming Language and System: Part I-Description of Language Features," 1995.

[7] B. Ramkumar, A. Sinha, V. Saletore, and L. Kale, "The Charm Parallel Programming Language and System: Part II-The Runtime System," *IEEE Transactions on Parallel and Distributed Systems*, 1994.

[8] S. Bertozzi, A. Acquaviva, D. Bertozzi, and A. Poggiali, "Supporting Task Migration in Multi-Processor Systems-on-Chip: A Feasibility Study," in *Proc. Design, Automation and Test in Europe (DATE)*, 2006, pp. 15–20.

[9] "Distributed Application Layer," 2012. [Online]. Available: http://www.tik.ee.ethz.ch/~euretile/

[10] T. Parks, "Bounded Scheduling of Process Networks," Ph.D. dissertation, University of California, 1995.

[11] M. Geilen and T. Basten, "Requirements on the ecution of Kahn Process Networks," *Programming Languages and Systems*, pp. 319–334, 2003.

[12] K. Huang, W. Haid, I. Bacivarov, and L. Thiele, "Coupling MPARM with DOL," ETH Zurich, Technical Report, Tech. Rep., 2009.

[13] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: Simplifying Event-driven Programming of Memory-constrained Embedded Systems," in *Proc. Int'l Conf. on Embedded Networked Sensor Systems*. ACM, 2006, pp. 29–42.