Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

MASTER THESIS
AT THE DEPARTMENT OF INFORMATION TECHNOLOGY
AND ELECTRICAL ENGINEERING

A Modular Trace-based Simulation Framework for Multiprocessor Systems

Jun Liu

Advisors: Iuliana Bacivarov, Kai Huang
Professor: Prof. Dr. Lothar Thiele

August 12th, 2007

Abstract

The increasing complexity and heterogeneity in multiprocessor SoCs call for new performance evaluation methodologies in order to meet the severe time-to-market constraints. This thesis aims at designing a performance evaluation approach for multiprocessor SoCs at early stages in the design space exploration. The challenge is to speed-up the simulation and at the same time to maintain accuracy for making design decisions.

To overcome the challenge, we present a modular trace-based simulation framework, in which the application functionality is abstracted as high-level traces in order to achieve efficiency and allow for fast design space exploration. Compared with existing trace-based approaches, it has the following features: 1) the traces are generated automatically without any manual instrumentation; 2) both the application and the architecture are scalable and modular; 3) the framework can explore a variety of parameters, e.g. different resource bindings, different scheduling policies including preemption, complex communication paths, different communication buffer locations and different atomic transaction data sizes.

We implement the framework in the context of Distributed Operation Layer, which aims at optimizing the mapping of parallel applications onto multiprocessor architectures. To verify our framework's modularity, efficiency and capability to explore various parameters, we extensively test it on two case studies: a producer-consumer application and an MPEG-2 decoder.

Contents

1 Introduction

This thesis aims at designing a performance evaluation approach with both accuracy and efficiency at system level. The first section describes the context of our work and the reason of making use of simulation method at system level. After that the motivations and the various factors that need to be considered in the system level simulation are illustrated. In Section 1.3, we proposed a modular trace-based simulation framework, which can help to solve the challenges in mapping parallel applications onto multiprocessor architectures. Finally the organization of this thesis is presented.

1.1 Context

The advancement in electronic technologies have led to the emergence of the System-on-Chip (SoC), which enables more and more functionality to be integrated on a single chip. With the increasing complexity of modern applications, traditional single processor architecture can no longer meet the demanding performance requirements. Nowadays, the trend of embedded system design is moving from single processor architecture toward heterogeneous multiprocessor SoC architecture. This shift calls for new methodologies because traditional ad-hoc approaches fall short for dealing with the complexity and heterogeneity of multiprocessor SoCs.

One primary goal in the process of system development is to meet the severe time-to-market constraints. Due to the fact that the multiprocessor SoC has opened up a large design space, having an efficient and accurate performance estimation method is mandatory for the design space exploration, especially in the early design stage. Traditional cycle-accurate cosimulations are impractical anymore at this scenario, because the time for constructing a synthesizable system as well as simulating softwares on it using multiple Instruction Set Simulators (ISS) is not affordable, which may take days of work, even months. Therefore, performance evaluation at a higher abstraction level, e.g. system level, is necessary to minimize the modeling effort, time-to-market, as well as to get the best trade-off between accuracy and speed.

The context of the work is the development of Distributed Operation Layer (DOL) [1], which aims at optimizing the mapping of parallel applications onto multiprocessor architectures. The DOL takes the application specification, the architecture specification as inputs and generates the optimized mapping specification. During the design space exploration, the mapping optimization procedure makes use of various performance evaluation strategies at different abstraction levels. The formal performance analysis method has the highest speed, yet its bounds are not tight. It can be used to select a set of possible alternatives when the design space is large. The system level simulation framework can then be used to validate the results of the formal performance analysis method. More importantly, it is used to optimize the mappings and confine the design candidates to a small sphere.

1.2 Motivation

The challenge of system level performance evaluation for mapping an application onto a multiprocessor embedded system is to speed-up the simulation, and at the same time maintain sufficient accuracy for making design decisions, which implies that the impact engendered by the architecture should be taken into account. The basic principles that should be considered for system level performance evaluation can be summarized as follows: effect of the sharing of a resource, effect of communication architectures, efficiency, accuracy, modularity and automatization.

Effect of Sharing of Resource

In a multiprocessor System-on-Chip architecture, several processes may be mapped onto the same computation resource. They have to compete with each other to get access to the CPU and to carry out their own functionalities. Yet in some performance analysis methodologies, the effect of sharing computation resources is neglected. It assumes the largest parallelism among the processes mapped onto the same processor to the extent that no process will block because of access conflicts, which does not reflect the actual situation. Therefore, the performance evaluation framework should provide various arbitration mechanisms to assign mutually exclusive access to the computation resource.

Similarly, the effect of sharing communication resources, e.g. the communication bus, should also be taken into consideration. The communication delays through one bus result not only from the bandwidth, but also from the unavailability of the bus. If the bus is taken by another resource at the moment, the communication requests can only be satisfied until the bus is released and free again.

Effect of Communication Architecture

In DOL, an application is modeled as process network, the processes of which will be mapped onto different processors and to cooperate together by means of inter-process communications in order to accomplish the overall functionality. There are a variety of ways for these processes to send and receive data between each other. One possibility is that two processes are mapped onto the same processor. Under such circumstance, these two processes are able to exchange data through internal communication. Where the data transfer between processes can be conducted through local buffers and does not need to access the two external bus. The overhead will be much less, compared with the case when the processes communicate through an external communication media. In the external communication, the data from the sender process will have to traverse at least one bus to reach the destination. The time delay on the bus due to the limitation of bandwidth or access conflicts is considerable. Additionally, the communication between two processes can as well be carried out via a series of buses. The overhead stemming from going through all these buses is more significant.

Moreover, another important aspect of the communication architecture is the location of the memory that is used to buffer the communication data. Considering the case of an external communication via one bus, if the memory is located on one side of the two processors involved in the communication, e.g. origin processor, only the target processor who tries to read data from the memory will have to access the bus. In this case, the speed for the origin processor to access the memory will be much faster. In contrast, if the buffer memory is implemented on a shared memory linked to the bus, both write requested from origin processor and read requested from target processor ought to occupy the bus, resulting in slower data transfer speed.

Therefore, the variety and complexity of communication architectures in multiprocessor SoCs can produce a vast range of alternatives for implementation. Therefore, the communication plays a key role and can exert considerable influence on the final system performance. This requires that a system level performance evaluation tool should be capable of exploring various candidates and derive meaningful performance estimation for each alternative of the communication architecture.

Efficiency and Accuracy

In the initial design stages, the design space is extraordinarily large that the efficiency of the performance evaluation is vital to guarantee the time-to-market constraints. Hence any method that can lead to an improvement of the simulation speed merits serious consideration. The dilemma, however, is that efficiency and accuracy are usually contradictory. Thus it is difficult to improve these two ingredients at the same time. As a matter of fact, it is very likely that the measures to increase efficiency can degrade accuracy, and vice versa. To overcome this contradiction, the performance framework should provide embedded system designers with flexible choices between accuracy and efficiency, in accordance with their actual needs. When the number of alternative solutions is large, the efficiency may take precedence over accuracy. When the candidates are scaled down to a small number after initial explorations, the designers can then increase the accuracy of the performance evaluation to obtain the best solutions.

Modularity and Automatization

In the domain of multiprocessor SoCs design, both applications and architectures can vary significantly. On the one hand, the number of processes, their respective definition and the communication among them differ from application to application. On the other hand, the number of the processors, the processor type and the communication architecture connecting them can also be flexible. Accordingly, the performance evaluation framework should be made modular to the largest extent and be able to cope with different scenarios. It should not be limited to a specific application or hardware architecture platform, which means the performance evaluation framework needs little or no modification when given a set of valid specification for the application, architecture and mapping.

Moreover, since the system level performance evaluation is used to investigate a vast rang of possible solutions, it is important to make it automatic in order to release the system designers from burdensome task of manual operations, as well as to increase efficiency. in a performance estimation framework, the system designers will be harassed by these manual operations which are error prone and even infeasible when the scale of the applications grow increasingly larger.

1.3 Contribution

In this thesis, we present a modular trace-based simulation framework to solve the challenges faced by system level performance evaluation. This framework is based on execution traces in order to achieve high efficiency and allow for fast design space exploration. In the simulation framework, the application functionality is represented as a set of traces, in which the computation and communication behaviors are abstracted as computation and communication events. Its features are listed as follows:

- The trace generation is completely automatic, which releases the designers from tedious manual instrumentation.
- Both the application and architecture modular, which means that without any modification, the framework can deal with mappings with different applications and architectures specified under certain schemes.
- It can explore different resource bindings.
- To model the sharing of computation and communication resources, different arbitration mechanisms, e.g. TDMA, FIFO and Fixed Priority (FP) with preemption, are implemented for the resource sharing.
- It can explore complex communication path via a sequence of communication resources.
- It can explore different communication buffer locations, i.e. internal memory at original processor or target processor and shared memory.
- It can explore different atomic transaction data size which represents different granularities of the packet size. The atomic transaction data size offers flexibility for designers with choices between accuracy and efficiency.

The proposed methodology is briefly illustrated in Figure 1.1. It mainly consists of two phases. In the first phase, we modify the existing functional simulation to enable the automated generation of traces which are independent of target architectures and scheduling policies. It has application specification as input and execution traces as output. In the second phase, with the execution traces, the architecture and the mapping as input,

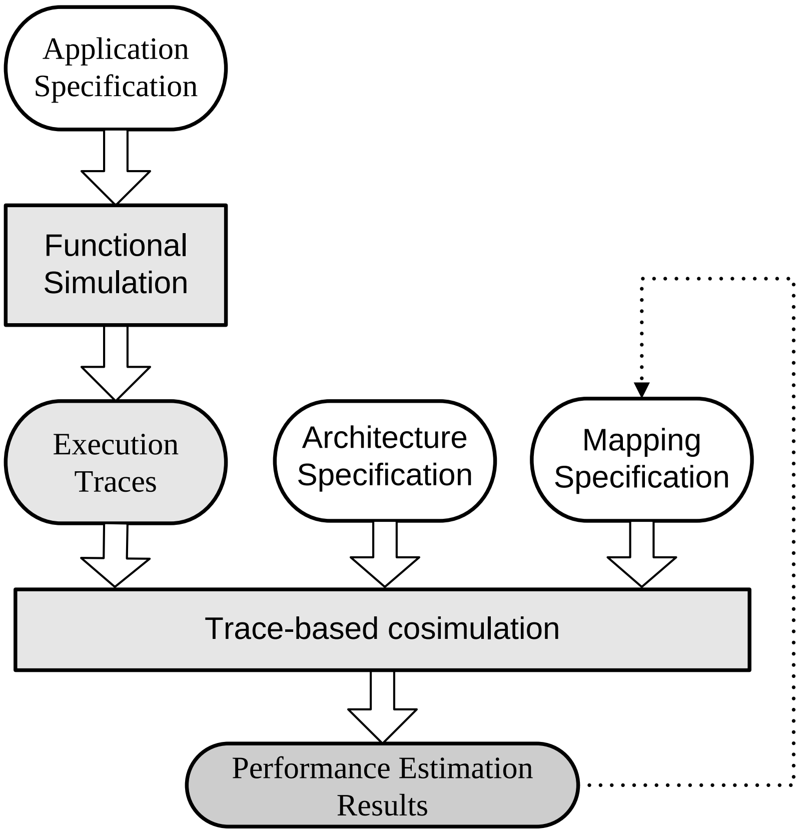


Figure 1.1: Proposed methodology.

the trace-based framework we built is used to obtain the performance estimation results, such as execution time, processor load and bus load, which can provide hints for either refining the existing mapping or designing a new mapping. In addition, the performance evaluation methodology is implemented and testified in the context of the Distributed Operation Layer. We take advantage of SystemC [2] to build the basic components. Besides, we use XML to configure and specify the architecture and the mapping. To validate its efficiency and capability to explore various parameters, we extensively test the framework on two case studies: the producer-consumer application and an MPEG-2 decoder.

1.4 Organization of the Thesis

The remainder of the thesis is organized as follows. In the next chapter, the related work with regard to high level simulation is explored. In the third chapter, we introduce our trace-based simulation framework, including the basic concepts and methodologies. The implementation of our framework in the context of DOL is described in detail in Chapter 4, while in Chapter 5, case studies on a simple producer-consumer application and an MPEG-2 decoder are used to test the framework. In the last chapter, we give the conclusions of this thesis, as well as the future work.

2
Related Work

In this chapter, we investigate the related work about system level performance evaluation methodologies, including timed functional simulation and the trace-based performance evaluation. In section 2.1, the state-of-art approaches in timed functional simulation are presented and their respective drawbacks are analyzed. Section 2.2 desribes the major achievements in the trace-based performance evaluation, i.e. trace-based simulation and trace-based analysis. Finally by comparing with other trace-based methodologies, the main characteristics of our approach are presented.

2.1 Timed Functional Simulation

As a good compromise between formal performance evaluation methods and cycle-accurate simulation, the timed functional simulation uses timing to model the running effect of an application. It is faster than cycle-accurate simulation and is more accurate than the formal performance evaluation method. But it needs to execute the functional code iteratively during the design space exploration, which degrades the efficiency. Some of the state-of-art timed functional simulation approaches are listed as follows:

The work in [3] addresses the problem of performance estimation for multiprocessor systems at the system level. It is based on a back-annotation approach to increase efficiency, which means that the timing elements needed for performance estimation are generated once and for all. The application code needs to be analyzed to isolate the basic computation blocks. The types of scheduling policies of the computation node are limited and the processes of a computation node are scheduled according to a uniform random distribution. Additionally, this framework does not investigate complex communication architecture.

Chronosym [4] uses a timed native execution model to simulate the execution of the software on the target architecture. Both the OS and software code are executed on the simulation host. The software code is annotated with execution delays to model the timing effect on the target processor. Since it does not need to simulate the processor functionality as ISS-based simulation does, Chronosym is much faster than the ISS-based simulation. Yet in the process of exploring a large design space, the software code and OS code should be executed iteratively, which limits its efficiency.

2.2 Trace-based Performance Evaluation

Trace-based performance evaluation methodologies are developed to overcome the drawback of the timed functional simulation. The speed-up is achieved by abstracting the computation behaviors as high-level execution traces. Therefore, to simulate the computation behavior effects, the target architecture will not need to run the actual programming code. Besides, in trace-based performance evaluation, the tedious analysis of branches in the source code is no longer required since the execution has already been flattened.

The work in [5] uses transaction level SystemC modeling technique to simulate the system. Since the functionality has been abstracted in the traces, it allows for a very fast investigation of SoC architectures. Yet the approach is not generic. It is mainly used to explore network processor architectures consisting of a variable number of processing units communicating via only one bus. Besides, it is designed to solve specific application area, namely packet processing. It assumes that an application is used to deal with different types of packets which will receive their corresponding processing.

Another trace-based simulation framework SoCExplore [6] is used for exploring communication-centric design space of complex SoCs with network based interconnects. It uses a communication event simulator SoCNet to model the communication and the execution order of the trace events. Since it does not make use of a well defined application model of computation, e.g. Kahn, an explicit concurrency extraction is needed to separate the application space from the system space. This is similar to the compilers that try to extract parallelism in the code. In addition, in order to generate traces, manual instrumentation is required to track the execution of the computation and communication behavior.

Sesame [7] tries to separate the application modeling and architecture modeling. An explicit mapping step is required to associate them together for trace-based simulation. The configurations of the application, architecture and mapping are all specified in parameter-based files. It supports gradual architecture model refinement. The Kahn Process Network model of computation [8] is utilized to specify the application. The traces are generated dynamically, by running the application model and are dispatched to the architecture model using a UNIX IPC-based interface, which means that the computation programming code in the application is executed iteratively for multiple mappings. Besides, manual annotation to the original source code is needed in order to track the computation and communication actions.

The trace-based performance analysis presented in [9] makes use of the communication analysis graph (CAG) to evaluate the performance of on-chip communication architectures. An initial cosimulation step is needed to generate traces, from which the CAG can be extracted. The CAG is then analyzed statically without dynamic simulation, resulting in faster design space exploration. But it is mainly used to investigate communication architectures, e.g. its topology and associated protocols. It does not support concurrent computation and communication mapping exploration. The sharing of computation resources is not considered. Besides, it does not contain various scheduling policies for scheduling the requests on the bus, and only a priority-based scheme is used. In addition, the granularity of the communication is coarse. In the CAG a communication event may contain read or write of large size and it will be executed atomically.

2.3 Proposed trace-based Simulation Framework

Our framework also adopts the trace-based approach to achieve efficiency. It aims at evaluating the system performance at a high level and can be used to explore design space at early stages. Compared with the foregoing trace-based performance evaluation methodologies, it has the following characteristics:

- The application is specified using a restricted version of Kahn process network. We extend the KPN by adding an extra constraint, i.e. the FIFO channel is bounded by a limited size. Consequently, both read and write will block when the channel is empty and full, respectively. Yet the deterministic characteristic of the KPN is maintained.

- The traces are generated automatically without any manual instrumentation.
- Once the traces are generated, they can be reused for different mappings of the same application. This means we only need to run the functional model once and the time for executing the computation programming code is saved during the trace-based simulation.
- It is able to explore arbitrary communication structures, e.g. internal communication, external communication, communication via a path. The communication path may consist of several communication resources.
- When the communication channel is mapped onto the architecture, one important factor that can affect the system performance is the location of software channel buffer. Our framework is capable of investigating the various scenarios of implementing the software channel buffer on the origin processor, target processor or shared memory. According to our investigation, this is the first time that the implementation of software channel buffer is considered in the system level performance evaluation.
- Various scheduling policies, including TDMA, FIFO and FP with preemption, have been implemented to provide exclusively mutual access to the resources.
- Trace refinement is conducted by dividing the previous communication event into smaller atomic communication events. The size of the atomic communication event is flexible to provide a balance between accuracy and efficiency.
- The framework is generic. The application is not limited to a specific area. Any application written under the process network specification scheme can be handled. Besides, the architecture in the framework can be very complex with arbitrary processors and buses.

3 Framework

In this chapter, the trace-based simulation framework we have proposed is presented in more detail. In Section 3.1, an overview of the framework is given, including the formal description and the system model. In the next sections, the fundamental blocks of the framework and their underlying mechanisms are described. For example, in Section 3.2, the way the application is specified and how the traces are generated from functional simulation are illustrated, while Section 3.3 explains the approach of modeling architecture, including the uniform definition of resources and the algorithms for various scheduling policies. The last section describes the way to simulate the computation event and communication event for certain mapping configuration.

3.1 Overview

Our framework aims at solving the problem for performance evaluation at high level. It takes application specification, architecture specification and mapping specification as input and derives from the simulation the system performance statistics, such as processor load, bus load, and overall system execution time. They can be used to guide the designers to improve the design of the system.

3.1.1 Problem Description

The application in our framework is previously specified by means of process network, a restricted version of KPN. It defines the functional behaviour for each process and how they communicate with each other via software channels. The details about process network will be explained in section 3.2.1. The architecture used in our framework is a tailored description of the target hardware platform, with only some major parameters concerning the hardware components. The major components are computation resources and communication resources. The mapping specification determines how the processes are mapped onto computation resources, as well as how the software channels are mapped onto communication resources.

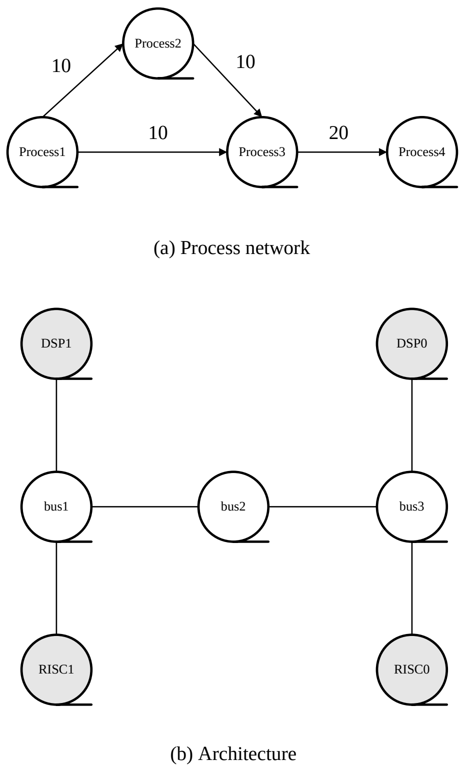


Figure 3.1: Process network and architecture.

As can be seen from Figure 3.1, an application can be described as a weighted directed acyclic graph $G_{\text{pn}} = (V_{\text{process}}, E_{\text{sw\_channel}})$, in which $E_{\text{sw\_channel}} \subseteq \langle V_{\text{process}} \times V_{\text{process}} \rangle$ is a partial order.

- V_{process} is the set of vertices in G_{pn}. $v_{\text{process}} \in V_{\text{process}}$ denotes one process in the process network. In each process the corresponding functional behavior is defined.

In the mapping, each process is a atomic mapping unit, indicating that it can only be mapped to one processor.

- $E_{\text{sw\_channel}}$ is the set of edges in G_{pn}. $e_{\text{sw\_channel}} \in E_{\text{sw\_channel}}$ denotes one software channel in the process network. The software channel is a bounded FIFO channel. It is used by two processes in the process network to communicate with each other. The direction of an edge denotes the data flow direction of the software channel. $S_{\text{buf\_size}}$ is the set of weights of the edges in G_{pn}. $s_{\text{buf\_size}} \in S_{\text{buf\_size}}$ edge denotes the buffer size of a corresponding software channel.

The architecture can be described as an undirected graph $G_{\text{arch}} = (V_{\text{resource}}, E_{\text{hw\_connection}})$.

- V_{resource} is the set of vertices in G_{arch}, and $V_{\text{resource}} = V_{\text{comp\_res}} \bigcup V_{\text{comm\_res}}$, where $V_{\text{comp\_res}}$ stands for the set of computation resource vertices and $V_{\text{comm\_res}}$ stands for the set of communication resource vertices. $v_{\text{resource}} \in V_{\text{resource}}$ denotes one resource in the architecture.
- $E_{\text{hw\_connection}}$ is the set of edges in G_{arch}. $e_{\text{hw\_connection}} \in E_{\text{hw\_connection}}$ denotes one connection between two hardware resources.

We define P_{comm} as the subset of paths in G_{arch}, and for each $p_{\text{comm}} \in P_{\text{comm}}$, both the start vertex and the last vertex should be computation resources. It denotes a possible communication path in the architecture level. For example, in Figure 3.1, path $< DSP1, bus1, RISC1 >$ is one elements in P_{comm}

The computation mapping is defined as a many-to-one function: $M_{\text{comp}} : V_{\text{process}} \longrightarrow V_{\text{comp\_res}}$. According to the definition, several processes can be mapped onto the same computation resource. But for each process, it can and only can be mapped onto one computation resource at a time.

The communication mapping is defined as a many-to-one function: $M_{\text{comm}} : E_{\text{sw\_channel}} \longrightarrow P_{\text{comm}}$. So it is possible that several software channels are mapped onto the same communication path. Yet for one software channel, it can only be mapped to one communication path each time.

The performance estimation is defined as a metric:

$T_{\text{perf}}\,(bus\_load, processor\_load, estimated\_execution\_time)$. The bus load_stands for the time the bus is in use for actual data transfer. The processor_load stands for the time the processor is in use for actual computation. The estimated_execution_time measures the estimated time needed for executing the application on the target architecture.

The goal of the trace-based simulation framework is to derive the performance statistics which will guide the designers to find the best mapping solution. According the foregoing definitions, we define our simulation framework as a function: $F_{\text{sim}} : (G_{\text{pn}}, G_{\text{arch}}, M_{\text{comp}}, M_{\text{comm}}) \longrightarrow T_{\text{perf}}$. It takes application G_{pn}, architecture G_{arch}, computation mapping M_{comp} and communication mapping M_{comm} as input. After the simulation, the performance statistics T_{perf} is obtained.

3.1.2 System Model

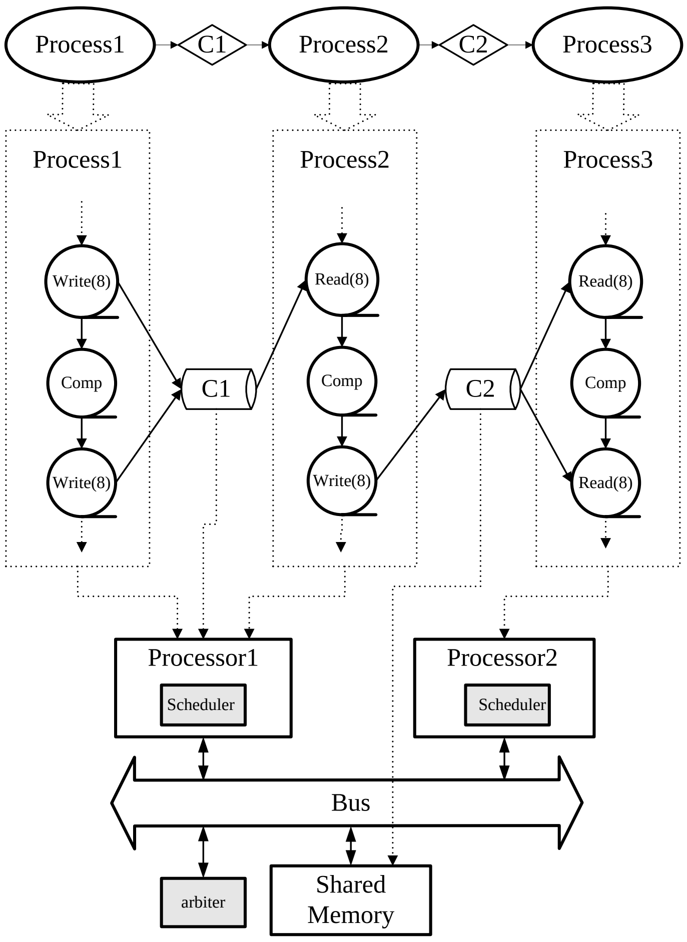


Figure 3.2: Trace-based simulation framework.

The infrastructure of the trace-based simulation framework is illustrated in Figure 3.2, where we make use of a concrete application and architecture as an example to explain the framework while in fact they are both modular and can be easily extended. As can be seen in the figure, the application consists of three processes and two software channels, i.e. software channel *C1* connecting *Process1* and *Process2*, software channel *C2* connecting *Process2* and *Process3*. The architecture contains two processors, one bus, and a shared memory. For computation mapping, *Process1* and *Process2* are both mapped to *Processor1*, *Process3* to *Processor2*. For communication mapping, software channel *C1* is mapped to *Processor1* because both *Process1* and *Process2* are mapped onto *Processor1*, and *C2* mapped to the path $< Processor1, Bus, Processor2 >$.

There are two phases in our simulation framework: trace generation and trace-based simulation. In the first phase, the traces are generated only once via functional simulation. In the original application specification, the behaviours of the processes are described with sequential programming language, e.g. C. By conducting functional simulation on the process network, pure functional execution traces are produced. The traces generated from this step are architecture and scheduling independent. In the second phase, trace-based simulation is executed each time a new design needs evaluation while the traces can be reused. Another input for the simulation is the hardware architecture, which is derived from the architecture specification. The resource in the architecture can simulate the performance consequence of the trace event and schedule trace events from different processes according to a given policy (e.g. TDMA, FIFO, and fixed priority). The time needed to finish a trace event is related with the parameters of the resource, e.g. type of processor and clock frequency. A mapping specification links the application and the architecture. For certain mapping, the system is simulated and the performance estimation results are obtained.

3.2 Application Modeling

As is explained in Figure 3.2, the application in the trace-based simulation framework is represented as execution traces produced from functional simulation. In this section, we mainly focus on explaining the process network and the execution traces.

3.2.1 Process Network

The application is initially modeled in the form of process network, a restricted version of the Kahn Process Network (KPN). In KPN, parallel processes communicate with each other through unbounded FIFO channels. For communication, only reading from channel may block for missing data, while writing to channel never stall. One major characteristic of KPN is that it is deterministic, which implies that the same application input will always produce the same outputs, regardless of implementation platform or scheduling. In the application specification of our framework, we extend the KPN by adding an extra constraint, i.e. the FIFO channel is bounded by a limited size. Consequently, writing to channel may block when the buffer is full. The reason for introducing blocking write is that on the one hand, the unbounded FIFO channel does not exist in reality, thus limiting the expression ability of KPN. On the other hand, possible deadlock caused by the communication between processes can be detected in our model. In a word, both blocking read and blocking write will be considered, yet the deterministic characteristic of KPN has retained, which lays the foundation for using function traces to represent the combination of certain input and the application and subsequently drive the underlying architecture model to simulate the system.

3.2.2 Execution Trace

Trace Generation

As it has been described in Figure 3.1, the process network can be represented as a weighted directed acyclic graph. The function of each process component is specified in C programming language, regardless of the future implementation. With the process network, we should do initial simulation to obtain the traces for later simulation. Simulation can be carried out at different levels, according to the degree of exposure of final implementation detail. The lower-level simulation, e.g. ISS-based simulation, is more accurate, yet its disadvantage is low speed. So we need to raise the abstraction level. As the functional simulation focuses on validating the algorithm of the application, it does not consider the impact of the target hardware architecture, such as the detailed execution on the target resource and how the communication is performed. No timing information can be collected from functional simulation and only the partial order of the computation behaviours and communication behaviours can be established. The high level of simulation makes it faster than other simulation methods. The fact that we are only concerned about the execution procedure of each process and the existing dependency among them makes functional simulation suitable for trace generation. The role played by the functional simulation can be depicted in Figure 3.3. The traces are generated only once and can be reused for each mapping in trace-based simulation.

Trace Representation

The trace events in traces are classified into three types: computation event, read event and write event.

- The computation event abstracts the function of a basic computation block by iden-

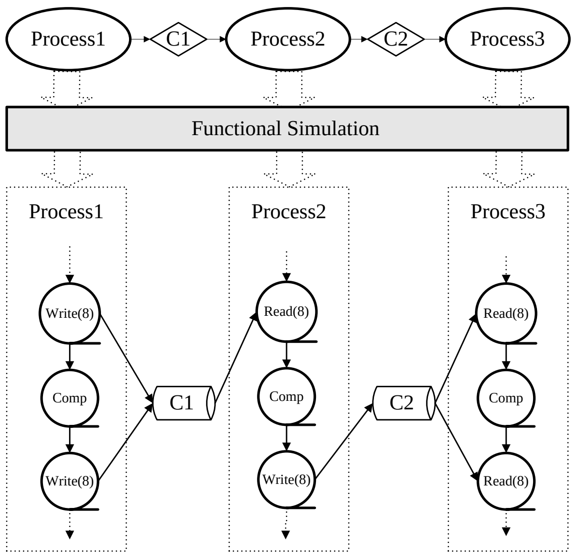


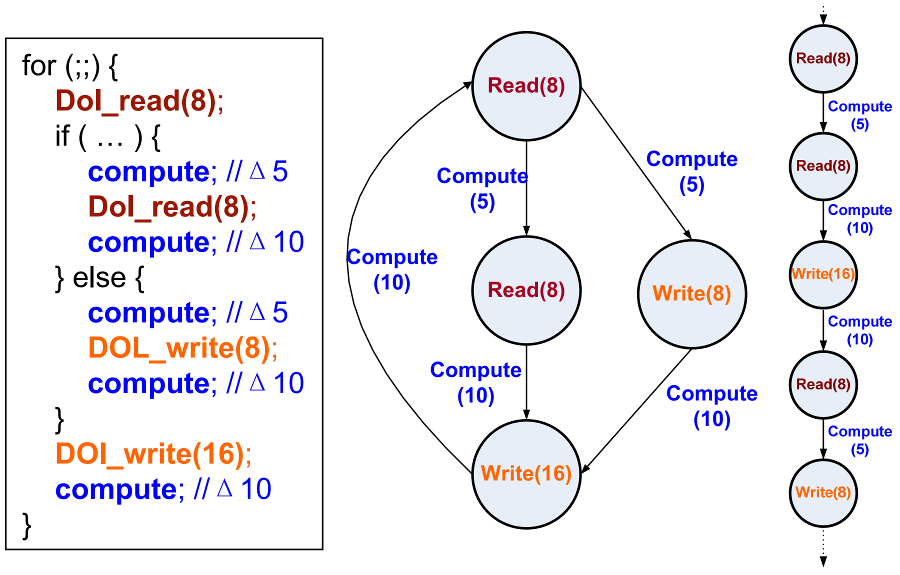
Figure 3.3: Functional simulation for trace generation.

tifying the corresponding code. When the computation resource simulates the consequence of running the computation event, it does not have to do the actual computation. Instead, it will only wait for some time delay, which is the time estimation for this computation running on a certain computation resource, e.g. DSP. Since some computation, e.g. DCT, may need significant time to finish on the simulation host, the abstraction of computation in this way can considerably increase the speed of our trace-based simulation.

- The write event contains how many data items and which software channel to write to.
- Similarly, read event records how many data items and which channel to read from.

The granularity of a trace event can be illustrated by Figure 3.4. In the box is the source code segment of a process. On the right is its abstract representation, i.e. a directed graph $G_{\text{process}} = (V_{\text{comm}}, E_{\text{comp}})$, in which the vertex V_{comm} corresponds to a read or write while the edge E_{comp} stands for the computation. The weight of the edge denotes the execution delays of the computation. We use the unfolded version of this graph to represent one possible execution path, namely trace.

Each process in the process network has its own trace, of which the trace events form a total order. Because of the existence of blocking read and blocking write, the execution of the trace events from different process may be interrelated. Therefore, as a whole, the



```
for (;;) {
   Dol_read(8);
   if ( … ) {
      compute; // Δ 5
      Dol_read(8);
      compute; // Δ 10
   } else {
      compute; // Δ 5
      DOL_write(8);
      compute; // Δ 10
   }
   DOl_write(16);
   compute; // Δ 10
}
```

Figure 3.4: Trace granularity.

trace events of the application form a partial order. Thus if there is no order between two events, they can run in parallel on the target architecture. The approach to express dependencies among trace events can be illustrated in Figure 3.3.

Before defining the dependency between two traces events, we introduce the following functions for each software channel $c_k \in C$ on trace event e_i, where C is the set of software channels in the application. The function W_{c_k} describes the writing of a trace event to software channel c_k, while R_{c_k} describes the reading of a trace event to software channel c_k.

$$W_{c_k}(e_i) = \begin{cases} d & \text{if } e_i \text{ writes d bytes to } c_k \\ 0 & \text{else} \end{cases} \tag{3.1a}$$

$$R_{c_k}(e_i) = \begin{cases} -d & \text{if } e_i \text{ reads d bytes from } c_k \\ 0 & \text{else} \end{cases} \tag{3.1b}$$

So if $W_{c_k}(e_i) = d$, the data available of c_k will increase by d bytes. And if $R_{c_k}(e_i) = -d$, the data available of c_k will decrease by d bytes.

We define $e_{p,m} \in E_p$ and $e_{q,n} \in E_p$, where E_p is the set of trace events for process p, E_q is the set of trace events for process q, $e_{p,m}$ is the mth trace event in E_p and $e_{q,n}$ is the nth trace event in E_q. We say that $e_{p,m}$ is directly dependent on $e_{q,n}$ if one of the following constraints is satisfied:

- $p = q$, and $m = n + 1$;
- $p \neq q$, and there exists $c_k \in C$,

$$\begin{cases} \sum_{i=1}^{m} R_{c_k}(e_{p,i}) + \sum_{j=1}^{n-1} W_{c_k}(e_{q,j}) < 0 \\ \sum_{i=1}^{m} R_{c_k}(e_{p,i}) + \sum_{j=1}^{n} W_{c_k}(e_{q,j}) \geq 0 \end{cases} \tag{3.2}$$

- $p \neq q$, and there exists $c_k \in C$,

$$\begin{cases} \sum_{i=1}^{m} W_{c_k}(e_{p,i}) + \sum_{j=1}^{n-1} R_{c_k}(e_{q,j}) > s_{c_k}, \text{where } s_{c_k} \text{ is the size of } c_k. \\ \sum_{i=1}^{m} W_{c_k}(e_{p,i}) + \sum_{j=1}^{n} R_{c_k}(e_{q,j}) \leq s_{c_k} \end{cases} \tag{3.3}$$

As a matter of fact, the first constraint implies that two trace events are in the same process. The second constraint stands for blocking read, in which the first formula means the read can not be satisfied and the second one means the read can be satisfied. The third constraint represents blocking write, in which the first formula means the write can not be satisfied and the second one means the write can be satisfied.

Trace Transformation

The communication events are responsible for modeling data transfer. For two communication events, e.g. one write event writing 64 bytes to the channel and the read event reading 64 bytes as depicted in Figure 3.6, it means in the trace-based simulation the read event can be scheduled only after the write event has finished. The granularity of the communication in this way is coarse. Yet it is highly possible that the buffer of the software channel is implemented in a pipelined way, indicating that the process in the receiving side of the channel can immediately start to read after the process in the sending side writes some data into the channel, e.g. 4 bytes. In this case, it will be more likely to reflect the actual situation if the original communication events are split into a series of atomic communication events, each one handling 4 bytes of data.

On the other hand, as the trace events will be simulated at the architecture level in the framework, the granularity of the communication of the traces should be closer to the hardware in order to increase accuracy. Take the architecture in Figure 3.5 for example. Consider 64 bytes of data are transferred from *processor1* to *processor2* via two buses with width of 32 bits, namely 4 bytes. After 4 bytes are transferred from *processor1* to the buffer of the bridge connecting *bus1* and *bus2*, *bus2* will start to transfer the 4 bytes to processor2, rather than wait until all 64 bytes are available. Therefore, the introduction of atomic communication event will facilitate the communication simulation at architecture level.

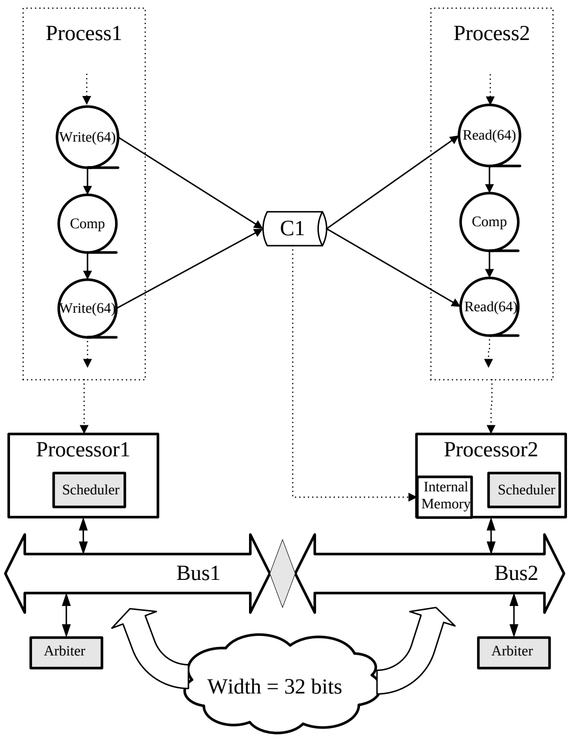


Figure 3.5: Architecture for trace refinement.

Another reason why we prefer to do trace transformation and split the events into atomic communication events is that it enables the implementation of preemption scheduling policy in an straightforward way. In the procedure of simulation, the atomic communication event should either be finished by the scheduler or not be executed at all. The details with regard to this topic will be presented in section 3.3.4.

The approach to do trace transformation is depicted in Figure 3.6. The atomic size of a communication event is fixed in one simulation. Yet it is configurable and can be changed by the user. In the process of trace transformation, each communication event containing data more than the atomic size will be divided into smaller ones, while other information about the trace events such as which software channel will maintain. For example, in Figure 3.6, we define the atomic size as 8 bytes. Thus the previous write event e_1 of *process1* with 64 bytes will be divided into 8 atomic trace events, each one containing 8 bytes. Yet the software channel binding information will retain and all the atomic trace events generated from e_1 will still write to *C1*. Additionally, as the atomic size gets smaller, the traces and will become larger. Because the trace-based simulation has to consider each trace event individually, the simulation speed will suffer from the trace transfromation. And the flexibility of atomic size provides the designer with a trade-off between accuracy and efficiency.

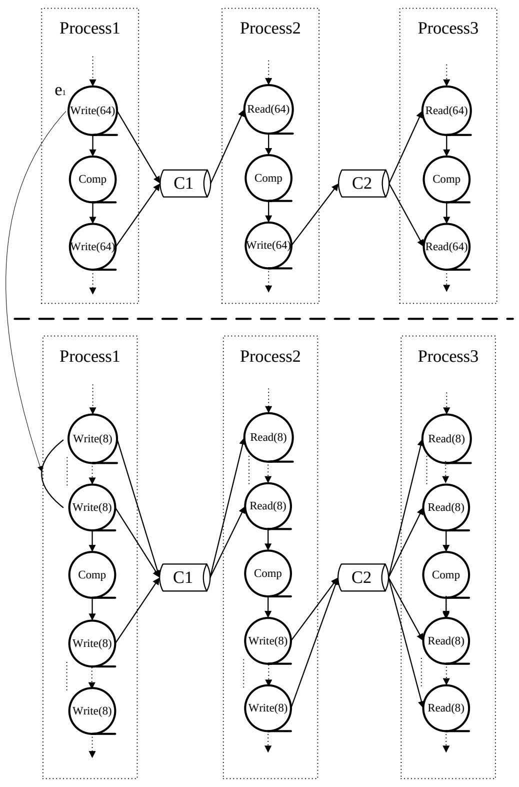


Figure 3.6: Trace transformation.

3.3 Architecture Modeling

The architecture is in charge of modeling the performance consequence of the trace events generated by means of functional simulation on process network. The organization of the architecture has been illustrated in Figure 3.1 and 3.2. It mainly consists of two types of resources, namely computation resource and communication resource. Their characteristics and how they communicate with each other are defined in a parameter-based scheme. At the initialization step of the simulation, all the resources are automatically extracted from the architecture design file and built from those parameters which specify the type of the resource, clock frequency and scheduling policy. The computation resource, e.g. RISC or DSP, is used for simulating the computation event. As the functional behaviour has already been captured in the process of generating traces, when handling the computation events, the computation resource only need to measure the time delays that need to be used for executing the computation on this processor, thus resulting in faster performance evaluation. Both the computation resource and communication resource can simulate the data transfer included in the communication event. For example, the internal communication takes place on the computation resource. Sometimes a single communication event should be executed consecutively by a series of resources in order to be finished, in accordance with the fact that in actual situation a data transfer from the origin processor needs to traverse several buses to arrive at the target processor. To provide exclusively mutual access to the resources, a set of arbitration mechanisms have been established, including TDMA, FIFO and fixed priority with preemption.

3.3.1 Architecture Resource

At the architecture level, all the components are modeled as resources, no matter whether it is a computation resource or communication resource. All the characteristics of the resources, such as clock frequency, are defined in a parameter-based way. Therefore, adding a new resource to the architecture or changing the characteristic of a resource only requires changing the corresponding parameters in the architecture file, which makes our modeling of architecture modular.

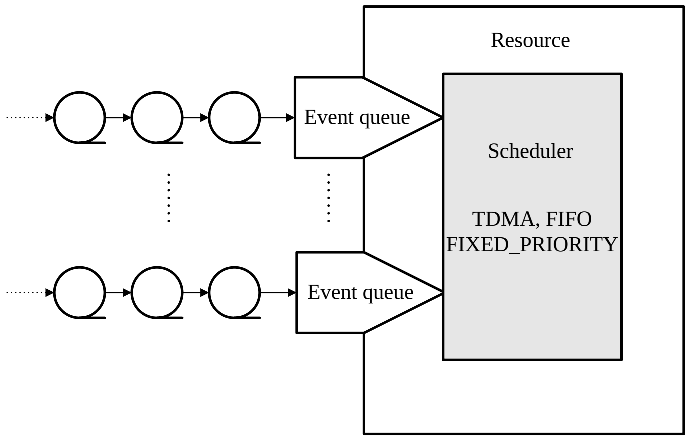


Figure 3.7: Skeleton of resources.

Figure 3.7 illustrates the uniform skeleton of resources. The architecture in our framework consists of a set of resources, including computation resources and communication resources. All the architecture resources are connected and attached through event queues which store the trace events arriving from another resource. Consider two processors linked to a bus in Figure 3.8. The bus will hold two different event queues, one for each processor. Therefore, when one processor delivers communication events to the bus in order to conduct reads or writes, they will be queued in their corresponding event queue of the bus. The event queues in the processors are used to store trace events from processes mapped onto them. In this way, the architecture can be easily extended and can form any complex hardware architecture.

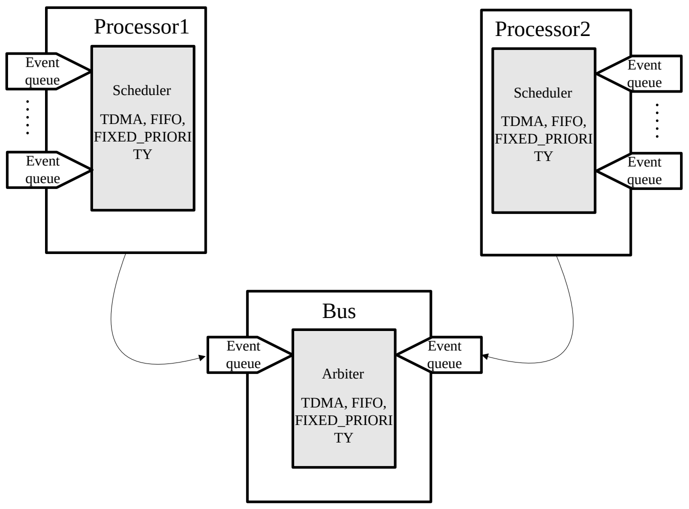


Figure 3.8: How to connect resources.

3.3.2 Processor

Processor is one type of computation resource onto which one or more processes of the application can be mapped. In our trace-based simulation framework, a processor is mainly responsible of scheduling the traces events delivered from processes according to a given scheduling policy. The also model the latency for processing a computation event. Moreover, a processor cooperates with other resources, e.g. buses, to simulate the performance consequence of a communication trace event. Two factors of the processor can affect the time needed to finish a trace event, as can be illustrated as follows: the processor type and its frequency.

Processor Type

In the first place, the latency caused by the same trace event on different processors may be different and is dependent on the type of the processors, e.g. DSP or RISC. It is very likely that an algorithm which is designed specifically for DSP can run much faster on DSP than on RISC. To reflect this difference and extract the corresponding performance estimation for a trace event on each type of processor, several methods can be used: low-level model of architecture, performance estimation tools, available documentation or experience of the designers.

Clock Frequency

In the second place, even for the same trace event mapped onto two processors of the same type, the latency used to model execution of the trace event can still be different because the two processors may have different clock frequencies. It is intuitive to note that the higher the frequency of the processor, the more powerful its processing capability can be. Assumes that it will take 5 cycles to accomplish the trace event on a RISC. In the architecture there are two RISC processors, whereas the clock frequency of the first processor is 100MHz and that of the second one is 200MHz. It will take about 50 ns for the first processor to finish the trace event while only 25 ns is needed for the latter one. Therefore, the latency T_{latency} ns caused by a trace event which requires N_{cycles} cycles in order to be finished on the target processor whose clock frequency is F_{clock} MHz can be calculated with:

$$T_{\text{latency}} = \frac{N_{\text{cycles}}}{F_{\text{clock}}} \times 10^3$$

3.3.3 Bus

The bus in the framework is one type of communication resources, used to transfer data between two different resources linked to it. In multiprocessor system, it plays the key role in communication. When two processes mapped onto two different processors need to exchange data, one common practice is to connect them through a bus. In our framework, we have modeled the bus as communication media. It is capable of dealing with communication trace events. For each resource linked to it, it maintains an event queue to buffer all the data transfer requests from that resource. The latency on the bus caused by trace event relies on how many data to transfer, which protocol to use, and the bus bandwidth. At the same time, there may be several requests form different resources to this bus. Hence an arbiter is required to schedule them and assign them exclusive access. Also, arbitration policies, such as TDMA, FIFO, fixed priority, are implemented in our framework.

The latency T_{latency} ns caused by a communication event with data quantity N_{data} bytes on a bus with bus width N_{width} bytes and bus frequency F_{clock} MHz under some protocol which requires T_{protocol} ns, can be expressed as:

$$T_{\text{latency}} = \lceil N_{\text{data}}/N_{\text{width}} \rceil \times \frac{1}{F_{\text{clock}}} \times 10^3 + T_{\text{protocol}} \tag{3.4}$$

3.3.4 Scheduling

Each resource in the architecture has its own scheduler to schedule the trace events dispatched onto it according to a given scheduling policy. The event queue is the entity that can be scheduled by the scheduler. For each scheduling policy, we design the corresponding algorithm. Yet the structure of the event queue is uniform for all types of schedulers.

Event Queue

An event queue is an infinite FIFO buffer used to store trace events delivered to it from other resources or processes. For processor, one event queue corresponds to one process mapped onto it. If the next trace event of the process's trace is ready to be executed, it is dispatched to the target processor and inserted at the end of the event queue. It will be scheduled by the scheduler if the traces events before it have all been finished and the time slice for this process is available. For communication resource, e.g. bus, one event queue stands for a resource with which it is connected, e.g. processor. It is for organizing reading or writing requests from that resource. In our framework, each event queue can be scheduled by the corresponding scheduler. As is depicted in Figure 3.9, it has three states, i.e. *ready*, *running* and *waiting*, in accordance with three different process states in scheduling. Take the event queue of TDMA scheduler for example. If there are trace events ready in the queue and the time slice is not available. This means that the process is waiting to be scheduled and it is not blocking on I/O and its state will be set to *ready*. After the time slice arrives, the scheduler will choose the earliest event from the event queue and simulate its execution by waiting for the corresponding amount of time. At this moment, the event queue's state is *running*. After all the trace events are finished, it is possible that the process is going to wait on I/O. No more trace event arrives and the event queue becomes empty, the state of the event queue is set to be *waiting*.

At present, three types of scheduling mechanisms have been implemented for our framework, which are TDMA, FIFO, fixed priority. Yet with similar mechanism, it is not difficult to implement any other scheduling policy (such as round-robin). The respective design details of the three scheduling policies are described as follows.

TDMA Scheduling

In Time-Division Multiple Access (TDMA) scheme, the time axis is divided into a number of time slots of a fixed length. Each event queue is allocated a fixed set of slots at which it can access the resource. This allows different event queues to share the processor or the bandwidth of the bus, thus providing some sort of fairness. Yet the overhead on this particular scheduler may be larger compared to other scheduling policies due to constant context switches. If the trace events in current event queue have not been finished before its time slot ends, the scheduler will switch to next event queue. This behaviour can be regarded as preemption in that the scheduler preempts current process while it is still running by taking CPU away from one process and give it to another. The state of an event queue in TDMA scheduler is illustrated in Figure 3.9. The procedure of state transition has already been described above. The algorithm to implement TDMA in our framework is illustrated in Algorithm 1.

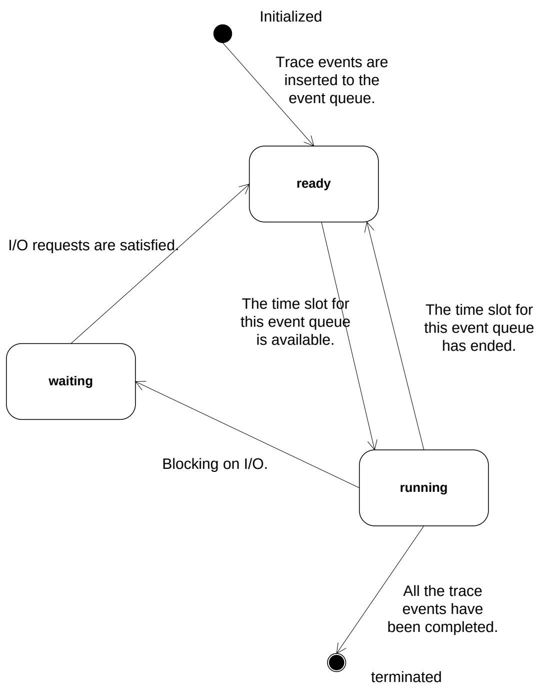

Initialized
Trace events are inserted to the event queue.
ready
I/O requests are satisfied.
The time slot for this event queue is available.
The time slot for this event queue has ended.
waiting
Blocking on I/O.
running
All the trace events have been completed.
terminated

Figure 3.9: Event queue states in TDMA scheduler.

Algorithm 1 TDMA SCHEDULER FOR PROCESSOR

```
1:  loop
2:    for all event queues $q_i \in Q$ in the scheduler do
3:      time slice left $t_i \leftarrow T_i$ which is the time slice assigned to this event queue
4:      while time slice left $t_i$ for $q_i$ has not expired do
5:        if the state $s_i$ of $q_i$ is WAITING then
6:          wait until $t_i$ ends or $s_i$ becomes READY
7:          if $t_i$ has ended then
8:            break
9:          end if
10:       end if
11:       fetch the next trace event $e_k$ from $q_i$
12:       if $e_k$ is a computation event then
13:         wait until $t_i$ ends or the performance estimation latency $l_k$ for $e_k$ ends
14:         update both $t_i$ and $l_k$
15:         if $l_k$ is 0 then
16:           finish $e_k$ and delete it from $q_i$
17:           if $q_i$ is empty then
18:             set $s_i$ to be WAITING
19:           end if
20:         end if
21:         if $t_i$ has ended then
22:           break
23:         end if
24:       else if $e_k$ is a write event then
25:         if $t_i$ is not sufficient for transfer data in $e_k$ then
26:           wait until $t_i$ ends
27:         end if
28:         wait for latency $l_k$ needed to transfer the data
29:         update $t_i$
30:         if the current resource is not the last one to handle $e_k$ then
31:           deliver $e_k$ to the next resource on the write path
32:         else
33:           update the channel buffer
34:         end if
35:         finish$e_k$ and delete it from $q_i$
36:         if $q_i$ is empty then
37:           set $s_i$ to be WAITING
38:         end if
39:         if $t_i$ has ended then
40:           break
41:         end if
42:       else if $e_k$ is a read event then
43:         if $t_i$ is not sufficient for transfer data in $e_k$ then
44:           wait until $t_i$ ends
45:         end if
46:         wait for latency $l_k$ needed to transfer the data
47:         update $t_i$
48:         if the current resource is the first one on the read path then
49:           update the channel buffer
50:         end if
51:         finish$e_k$ and delete it from $q_i$
52:         if $q_i$ is empty then
53:           set $s_i$ to be WAITING
54:         end if
55:         if $t_i$ has ended then
56:           break
57:         end if
58:       end if
59:     end while
60:     wait until the time slice $T_{\mathrm{i}+1}$ for $q_{\mathrm{i}+1}$ has come
61:   end for
62: end loop
```

FIFO Scheduling

The FIFO scheduler is based on the principle of a first-come, first-served behaviour. The event queue will be chosen to execute in the order they arrive. It is a non-preemptive scheduling algorithm so every event queue can execute until it gets blocked, which means it becomes empty. When an event queue becomes *ready* it is added to the tail of ready queue. The problem with FIFO is that the average waiting time can be long since it does not take into account short tasks before long tasks. Figure 3.10 describes the states transitions of an event queue in FIFO scheduler. If there are some trace events in the event queue waiting to be scheduled, its state is set to *ready* and is inserted at the end of the ready queue. Then if event queues arrived before it have all been finished by the resource or get blocked, it will be selected to execute by the scheduler and the its state is set to *running*. After that it holds the resource until all its trace events are finished and it will be waiting on I/O. The event queue will be *ready* again if I/O requests are satisfied. As can be seen in Figure 3.10, there is not state transition from *running* towards *ready* for the reason that in FIFO scheduling there is no preemption. The algorithm for implement FIFO scheduler is described in Algorithm 2.

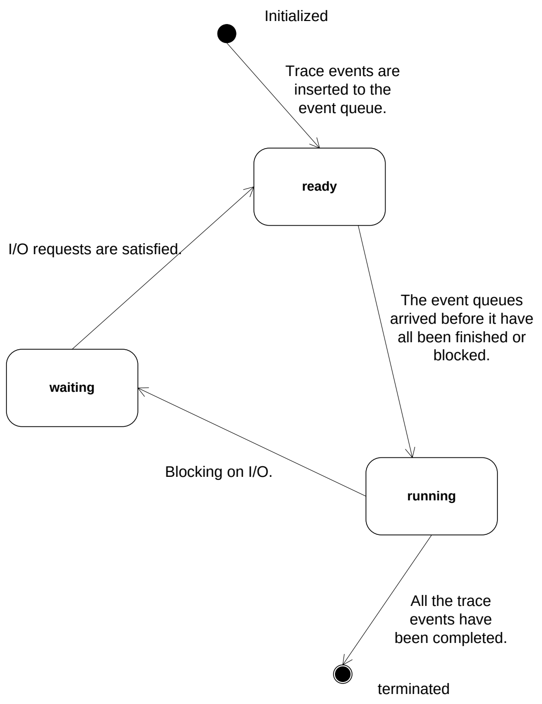

Initialized
Trace events are inserted to the event queue.
ready
I/O requests are satisfied.
The event queues arrived before it have all been finished or blocked.
waiting
Blocking on I/O.
running
All the trace events have been completed.
terminated

Figure 3.10: Event queue states in FIFO scheduler.

Algorithm 2 FIFO SCHEDULER FOR PROCESSOR

```
 1: loop
 2:   if the scheduling queue $q_{\text{schedule}}$ is empty then
 3:     wait until some event queue needs to be scheduled
 4:   end if
 5:   fetch from $q_{\text{schedule}}$ the first event queue $q_{\text{fst}}$
 6:   fetch the next trace event $e_k$ from $q_{\text{fst}}$
 7:   if $e_k$ is a computation event then
 8:     wait until the performance estimation latency $l_k$ for $e_k$ ends
 9:     finish $e_k$ and delete it from $q_{\text{fst}}$
10:     if $q_{\text{fst}}$ is empty then
11:       set the state $s_i$ of $q_{\text{fst}}$ to be WAITING
12:     end if
13:   else if $e_k$ is a write event then
14:     wait for latency $l_k$ needed to write the data
15:     if the current resource is not the last one to handle $e_k$ then
16:       deliver $e_k$ to the next resource on the write path
17:     else
18:       update the channel buffer
19:     end if
20:     finish$e_k$ and delete it from $q_{\text{fst}}$
21:     if $q_{\text{fst}}$ is empty then
22:       set $s_i$ to be WAITING
23:     end if
24:   else if $e_k$ is a read event then
25:     wait for latency $l_k$ needed to read the data
26:     if the current resource is the first one on the read path then
27:       update the channel buffer
28:     end if
29:     finish$e_k$ and delete it from $q_{\text{fst}}$
30:     if $q_{\text{fst}}$ is empty then
31:       set $s_i$ to be WAITING
32:     end if
33:   end if
34: end loop
```

Fixed Priority Scheduling

Fixed priority scheduling is previously known as real-time scheduling. The fixed priority scheduler guarantees that at any time the process which is ready to execute and has the highest priority can have access to the resource even if there is some other process running. In our framework, each event queue is set a fixed value to represent their respective priority. The event queue with highest priority always gets the resource as soon as its trace event arrives, thus may preempt the execution of other event queue. State transitions of an event queue in fixed priority scheduler can be seen in Figure 3.11. The preemption takes place when an event queue with higher priority is ready and the current event queue will be set to *ready* from *running*. If the event queues with higher priorities have all been finished or blocked, the event queue which was preempted before will resume its execution. The algorithm to implement fixed priority mechanism is illustrated in Algorithm 3.

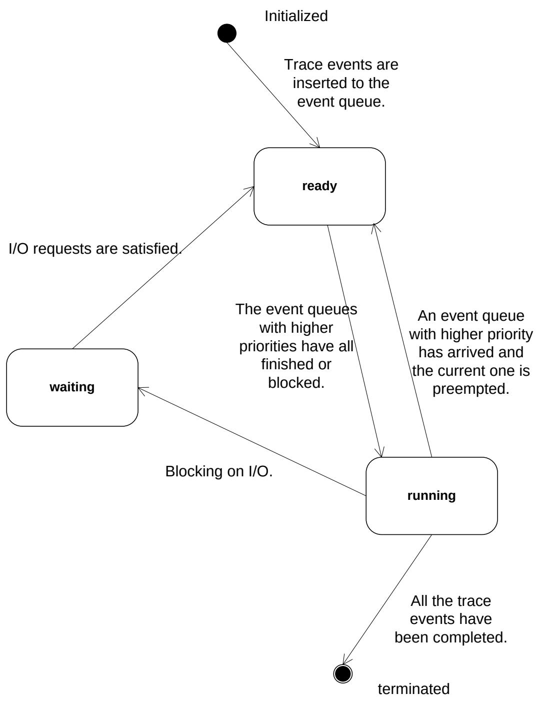

Initialized
Trace events are inserted to the event queue.
ready
I/O requests are satisfied.
The event queues with higher priorities have all finished or blocked.
An event queue with higher priority has arrived and the current one is preempted.
waiting
Blocking on I/O.
running
All the trace events have been completed.
terminated

Figure 3.11: Event queue states in fixed priority scheduler.

Algorithm 3 Fixed priority scheduler for processor

```
1: loop
2:   if the scheduling queue $q_{\text{schedule}}$ is empty then
3:     wait until some event queue needs to be scheduled
4:   end if
5:   fetch from $q_{\text{schedule}}$ the first event queue $q_{\text{fst}}$ with the highest priority
6:   fetch the next trace event $e_k$ from $q_{\text{fst}}$
7:   if $e_k$ is a computation event then
8:     wait until the performance estimation latency $l_k$ for $e_k$ ends or interrupt event is
       signaled
9:     update $l_k$
10:    if $l_k$ is 0 then
11:      finish $e_k$ and delete it from $q_{\text{fst}}$
12:      if $q_{\text{fst}}$ is empty then
13:        set the state $s_i$ of $q_{\text{fst}}$ to be WAITING
14:      end if
15:    end if
16:    finish $e_k$ and delete it from $q_{\text{fst}}$
17:  else if $e_k$ is a write event then
18:    wait until latency $l_k$ needed to write the data for $e_k$ ends or interrupt event is
       signaled
19:    if $e_k$ has not been finished then
20:      continue
21:    end if
22:    if the current resource is not the last one to handle $e_k$ then
23:      deliver $e_k$ to the next resource on the write path
24:    else
25:      update the channel buffer
26:    end if
27:    finish$e_k$ and delete it from $q_{\text{fst}}$
28:    if $q_{\text{fst}}$ is empty then
29:      set $s_i$ to be WAITING
30:    end if
31:  else if $e_k$ is a read event then
32:    wait until latency $l_k$ needed to read the data for $e_k$ ends or interrupt event is
       signaled
33:    if $e_k$ has not been finished then
34:      continue
35:    end if
36:    if the current resource is the first one on the read path then
37:      update the channel buffer
38:    end if
39:    finish$e_k$ and delete it from $q_{\text{fst}}$
40:    if $q_{\text{fst}}$ is empty then
41:      set $s_i$ to be WAITING
42:    end if
43:  end if
44: end loop
```

3.4 Mapping Modeling

In this section, three aspects concerning mapping are explained. In the first paragraph, some factors in the mapping that can affect the system performance and can be considered in our framework are investigated. In the second paragraph, the method of implementing mapping in our framework is presented. In the third paragraph, simulation work flow for certain mapping is generally described.

In multiprocessor system, provided the same application and the same architecture, performance can still be different due to different mappings. Just as section 3.1.1 has described, mapping can be divided into two types: computation mapping and communication mapping. For a functionality specified in the process, it can be faster if it is implemented on DSP rather than on RISC. Therefore, given a target architecture, we may try to map this functionality on the DSP processor in the first place. For the communication between different processes, it seems to be advantageous to implement it in one processor, without the need to visit external bus or memory. The problem is that we have to map all the processes between which there are communications onto one processor, which may dramatically increase the load of the processor and reduce the overall performance. In addition, sometimes we need to map a software channel to a communication path in order to exchange data between two process mapped onto two different processors. In this circumstance, the location of the software buffer between communicating processes can also have impact on the system performance. If the buffer is put in the local memory of origin processor, it means that if some process on the processor tries to write to the software buffer, it does not need to take any external bus. By contrast, if the software channel buffer is located in some shared memory linked to the bus, both reads and writes will have to be scheduled by the arbiter in order to access external bus, leading to extra load on the bus. In a word, there are a number of interleaved factors in the process of mapping and they may result in contradictory performance outcome. Therefore, to obtain optimized overall system performance, we need to make balance among different choices of computation mapping and communication mapping.

In our framework, the configuration of the mapping is stored in a file on a parameter basis. It specifies binding of process and computation resource, binding of software channel and communication path, buffer of software channel and the corresponding scheduling policy for each resource. For each mapping, the framework automatically extracts the mapping parameters from the mapping file, updates the corresponding information in the application and architecture, and then simulates the system. To change the mapping, we only need to change the corresponding parameters in the mapping file. For instance, if we intend to change the scheduling policy of a resource from TDMA to FIFO, we only need to modify scheduling part for this resource. This mapping specification mechanism will facilitate the automation of design space exploration since to investigate a different mapping only requires reconfiguring the mapping file.

After the mapping information is established in the simulation framework, performance estimation about computation trace event as to how much time is needed to accomplish the event on the target processor is attached. Then the processes begin to dispatch trace events to the target architecture. The computation event is directly delivered to the target processor. The latter will simulate the performance impact on the system of the computation by waiting for some time delays related to both the computation behaviour and target processor. Communication event first requests to the software channel the right to write or read, to see whether the space or the data is available, thus modeling blocking write and blocking read in the application. Afterwards if the I/O requests can be satisfied, the communication will be delivered to the target resource. The architecture then simulates the procedure of data transfer either on a processor internally or via a path consisting of a series of communication resource. In the following the detailed steps about how the application and architecture interact with each other to deal with computation mapping and communication mapping are described.

3.4.1 Computation Event Simulation

The work flow for dispatching computation events can be depicted in Figure 3.12. The sequence of the execution of all trace events conform to partial order, we have explained. A computation event is only directly dependent on its precursor in the same process. Consequently, if the trace events before it have all been finished by the target architecture, it is time for the process to deliver the event to the target processor, which is the first step in the figure. In the second step, the scheduler of the processor will schedule the coming event according to a give arbitration mechanism, e.g. TDMA, FIFO or fixed priority, and model its performance effects. After the computation event is finished by the processor, the process the trace event belongs to will be notified the completion in the third step, which will trigger the handling of the next trace event. In this way, the timing effects of the computation behaviours can be modeled.

3.4.2 Communication Event Simulation

In the process of simulating the communication event, two important issues are taken into account by our framework. For one thing, the dependencies between trace events are

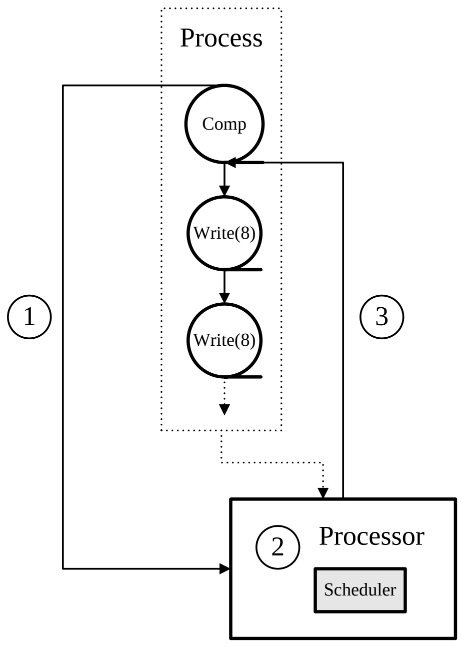


Figure 3.12: Computation event dispatching.

modeled. For another,the implementation of the software buffer can affect the performance of the system. Thus our framework should be able to handle the various scenarios of the locations of buffer.

In the application model, the dependency between different processes is established through software channels connecting them together. On the one hand, the read process ought to wait until the data it requests have been written by the write process. On the other hand, the write process has to ensure that the space in the software channel is sufficient for writing certain number of data. To model this kind of blocking write and blocking read, we introduce virtual software channel into our framework, in which only the abstract state of the software channel is recorded, e.g. the number of data or space available right now, without store actual data being transferred. This is compatible with the expression of communication trace event, in which only how many data items and which software channel to transfer are contained.

For a write event, it first sends requests to the virtual software channel for free space. If the space can not be satisfied for current writing, the write process then blocks. Otherwise it means the writing can take place, the space available of the software channel is changed by substracting the number of data in the write request. Yet the data available will not be updated immediately. Instead, it is changed after the write event is finished by the target architecture, which is more reasonable with regard to actual situation. The read event has similar behaviour, except it will have to request and change data available first and refresh space available after it is accomplished by the architecture.

Another important element that can exert influence on the performance of communication is the implementation of the software buffer if the software channel is mapped to a communication path. Assume the buffer is implemented in the original processor who starts the data transfer. It will not need to be granted to access the external bus in order to write to the local memory, resulting in much faster accessing speed. On the other hand, if the software buffer is located in the local memory of the target processor, accessing speed for read process will be faster. But since sometimes in the processor the local memory is so limited that the software buffer with certain size can not be allocated, it is necessary to place it in some shared memory connected to an external bus. This solution will make it essential for both reads and writes to compete for the bus and have access to the shared memory. Additionally, in multiprocessor system, two different processors may be connected through several buses. In this occasion, the communication between the processes mapped onto these two processors will be more costly. The model of the different scenarios in communication mapping and the relative work flow are described as follows.

Buffer on Original Processor

Work flow for dispatching write event when the buffer it writes to is implemented in the local memory of the origin processor can be described by Figure 3.13.

1. The write event of *Process1* sends request to *C1* to check if space is available. If not, *Process1* will block.
2. If in the first step the request is satisfied, the write event is dispatched to *Processor1*.
3. The scheduler of *Processor1* simulates the execution of the write event and finishes it by updating *C1*.

Work flow for dispatching read event when the buffer it reads from is implemented in the local memory of the origin processor can be described by Figure 3.14.

1. The read event of *process2* sends request to *C1* to check if data is available. If not, *process2* will block.
2. If in the first step the request is satisfied, the read event is dispatched to *bus*.

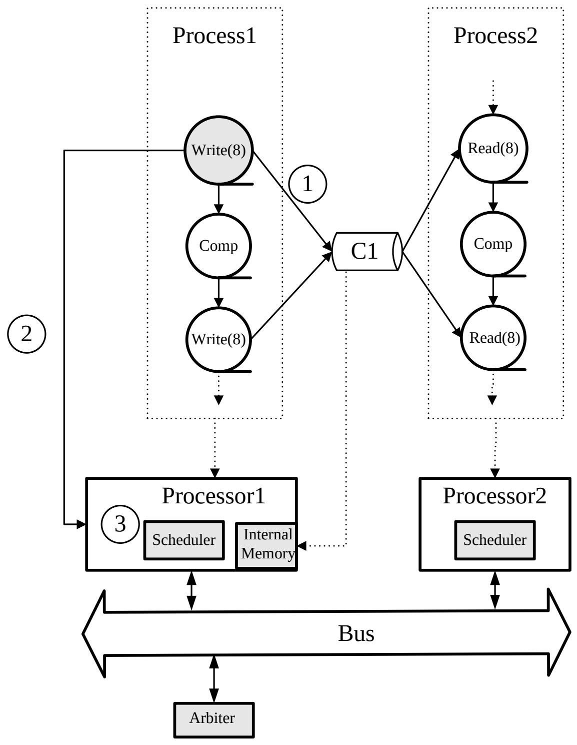


Figure 3.13: Write event dispatching when buffer of the software channel is on the origin processor.

3. The *bus* simulates the data transfer from *processor1* to *processor2* and updates the space available of *C1*.
4. The scheduler of *processor2* simulates the execution of the read event and finishes it.

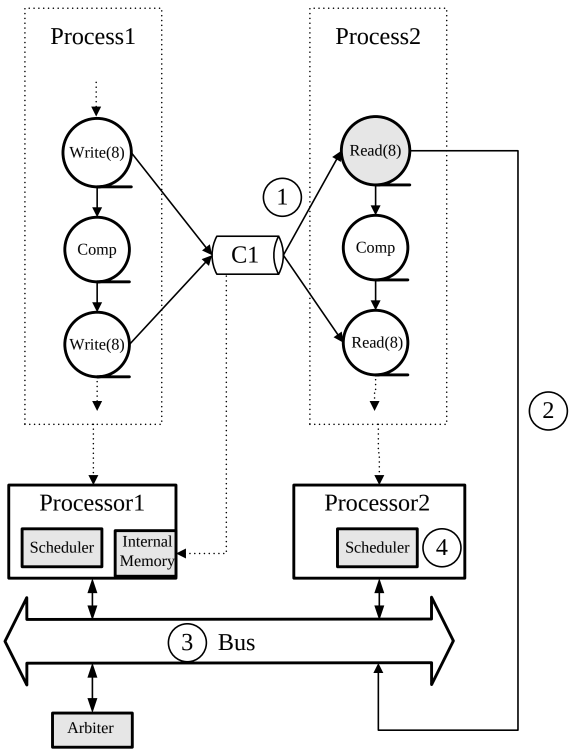


Figure 3.14: Read event dispatching when buffer of the software channel is on the origin processor.

Buffer on Target Processor

Work flow for dispatching write event when the buffer it writes to is implemented in the local memory of the target processor can be described by Figure 3.15.

1. The write event of *process1* sends request to *C1* to check if space is available. If not, *process1* will block.
2. If in the first step the request is satisfied, the write event is dispatched to *processor1.*
3. The scheduler of *processor1* simulates the execution of the write event and then dispatches it to the bus.
4. The *bus* simulates the data transfer from *processor1* to the internal memory of *processor2* and finishes it by updating *C1.*

Work flow for dispatching read event when the buffer it reads from is implemented in the local memory of the target processor can be described by Figure 3.16.

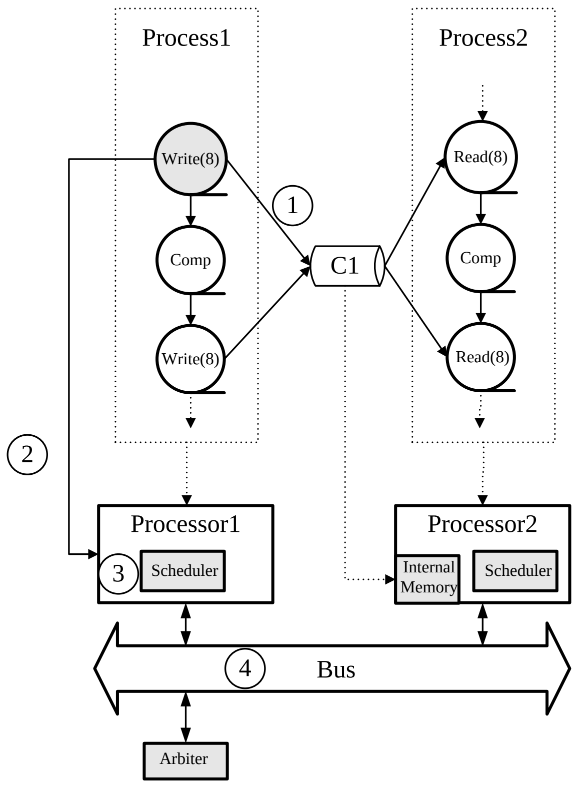


Figure 3.15: Write event dispatching when buffer of the software channel is on the target processor.

1. The read event of *process2* sends request to *C1* to check if data is available. If not, *process2* will block.
2. If in the first step the request is satisfied, the read event is dispatched to *processor2*.
3. The scheduler of *processor2* simulates the execution of the read event, and finishes it by updating space available of *C1*.

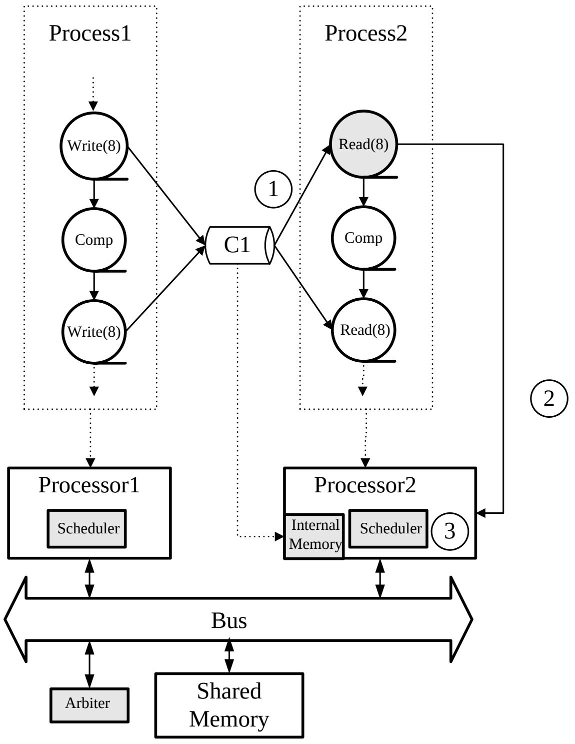


Figure 3.16: Read event dispatching when buffer of the software channel is on the target processor.

Buffer in Shared Memory

Work flow for dispatching write event when the buffer it writes to is implemented in the shared memory linked to an external bus can be described by Figure 3.17.

1. The write event of *process1* sends request to *C1* to check if space is available. If not, *process1* will block.
2. If in the first step the request is satisfied, the write event is dispatched to *processor1*.
3. The scheduler of *processor1* simulates the execution of the write event and then dispatches it to the *bus*.
4. The *bus* simulates the data transfer from *processor1* to the shared memory linked to the bus and finishes it by updating data available of C1.

Work flow for dispatching read event when the buffer it reads from is implemented in the shared memory linked to an external bus can be described by Figure 3.18.

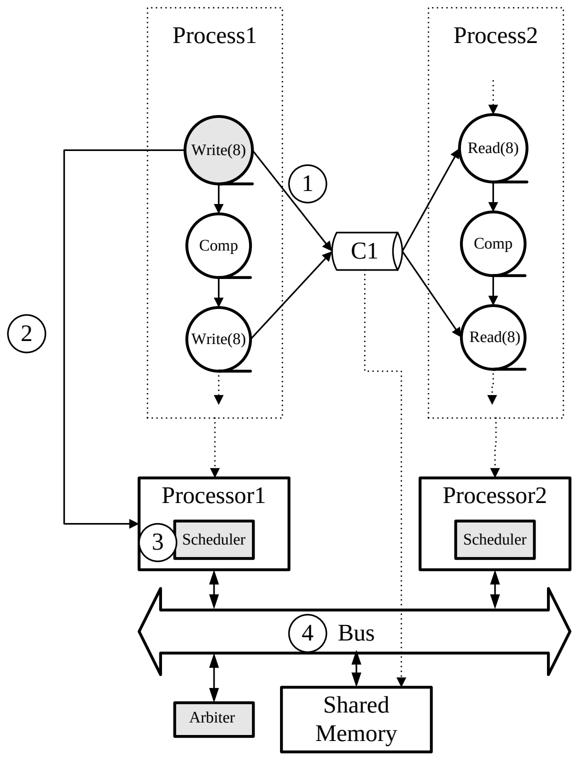


Figure 3.17: Write event dispatching when buffer of the software channel is in the shared memory.

1. The read event of *process2* sends request to *C1* to check if data is available. If not, *process2* will block.
2. If in the first step the request is satisfied, the read event is dispatched to the *bus*.
3. The *bus* simulates the data transfer from shared memory to *processor2* and updates the space available of *C1*.
4. The scheduler of *processor2* simulates the execution of the read event and finishes it.

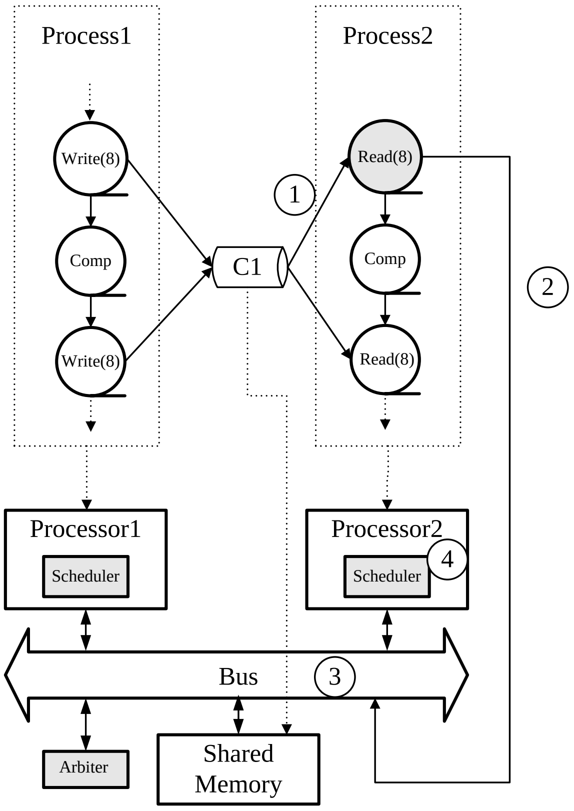


Figure 3.18: Read event dispatching when buffer of the software channel is in the shared memory.

Communication via Two Buses

Even though here we assume that the buffer of the software channel is placed in the *shared memory* linked to *bus1*, it can be easily extended to cover those general cases we have presented above. As is depicted in the Figure 3.19 and Figure 3.20, while the completion a write event requires the processing on both *processor1* and *bus1*, the accomplishment of a read event on *processor2* needs to access *bus1*, *bus2* and *processor2* sequentially. Besides, the number of resources that can appears in the communication path is not limited, which indicates that our framework is capable of handling complex communication architecture.

Work flow for dispatching write event when it involves two buses in the communication path can be described by Figure 3.19.

1. The write event of *process1* sends request to *C1* to check if space is available. If not, *process1* will block.
2. If in the first step the request is satisfied, the write event is dispatched to *processor1*.

3. The scheduler of *processor1* simulates the execution of the write event and then dispatches it to *bus1*.
4. *Bus1* simulates the data transfer from *processor1* to the shared memory linked to *bus1*, and then finishes the write event by updating data available of *C1*.

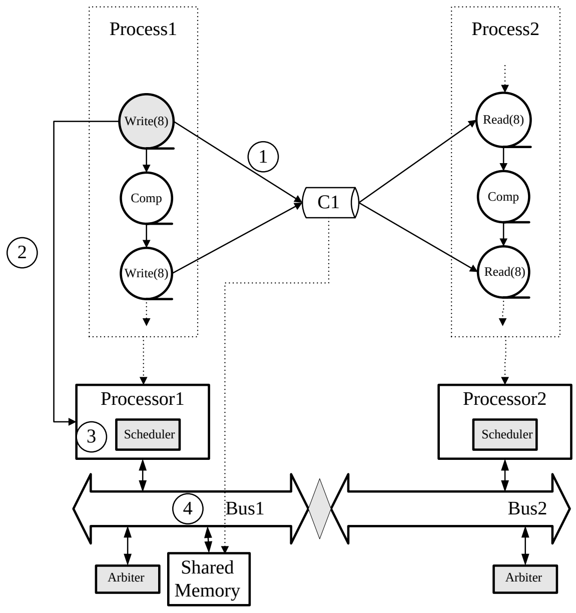


Figure 3.19: Write event dispatching when it involves two buses in the communication path.

Work flow for dispatching read event when it involves two buses in the communication path can be described by Figure 3.20.

1. The read event of *process2* sends request to *C1* to check if data is available. If not, *process2* will block.
2. If in the first step the request is satisfied, the read event is dispatched to *bus1*.
3. *Bus1* simulates the data transfer from shared memory to the bridge connecting *bus1* and *bus2*, and updates the space available of *C1*.
4. *Bus2* simulates the data transfer from the bridge to *processor2*.
5. The scheduler of *processor2* simulates the execution of the read event and finishes it.

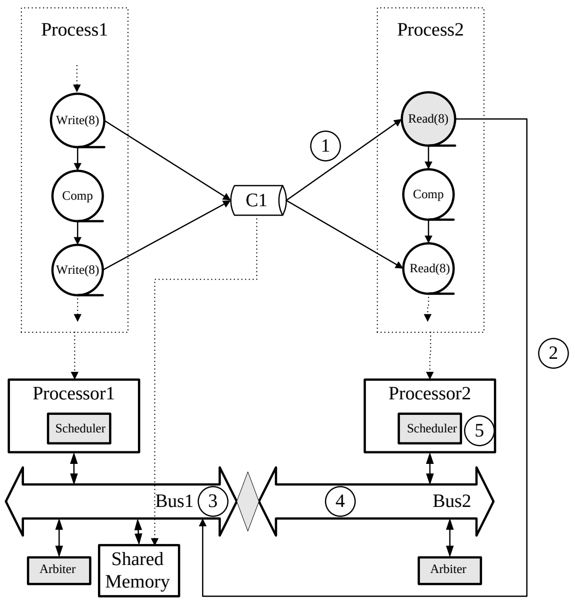


Figure 3.20: Read event dispatching when it involves two buses in the communication path.

Communication Path

As has been explained before, the software channel can be mapped onto a communication path consisting of an ordered set of resources. Here we redefine the communication path according to the track of data transfer on all the resources by a resource sequence $P_{\mathrm{comm}}\,(r_1, r_2, r_3, ..., r_{\mathrm{i}}, r_{\mathrm{mem}}, r_{\mathrm{i+1}}, ..., r_{\mathrm{n-1}}, r_{\mathrm{n}})$, where r_{mem} stands for the memory resource implementing software buffer and both r_1 and r_{n} are computation resources. The r_{mem} can be either a shared memory linked to an external bus or a local memory lying in a processor.

The write path can be derived from the communication path defined above and is defined as a resource sequence $P_{\mathrm{write}}\,(r_1, r_2, r_3, ..., r_{\mathrm{i}})$. Work flow for dispatching write event is to traverse all the resources, namely from r_1 to r_{i}, in the same order as they are specified in P_{write}. The performance consequence of a write event on $r_{\mathrm{k}} \in P_{\mathrm{write}}$ will be modeled by the scheduler of r_k after the trace event is delivered from $r_{\mathrm{k-1}}$ to r_k. Additionally, the software buffer will be updated after r_i accomplishes a write event.

In a similar way, the read path is defined as a resource sequence $P_{\mathrm{read}}\,(r_{\mathrm{i+1}}, r_{\mathrm{i+2}}, ..., i_{\mathrm{n}})$. Thus in order to finish a read event mapped onto P_{read}, each resource $r_{\mathrm{k}} \in P_{\mathrm{read}}$ will have to simulate the effect of the read event one after another according to their respective position in P_{read}. In contrast to the moment of updating software buffer in P_{write}, the first resource in P_{read}, namely $r_{\mathrm{i}+1}$, updates the buffer immediately after it finishes a read event.

All the scenarios investigated in previous sections conform to this scheme. And it is capable of dealing with all kinds of communication architectures considered in our framework.

4 Implementation

The framework of our trace-based simulation has been discussed in the previous chapter. In this chapter, we mainly concentrate on explaining how to implement the framework in DOL by means of SystemC. The first section gives an overview of the DOL, including its underlying mechanism and basic components. The method of expressing application, architecture and mapping in the xml files is then presented with concrete samples. The Section 4.2 describes the functional simulation of DOL and how to modify the original DOL APIs in order to generate traces. After that, in Section 4.3, the class diagram of the trace-based simulation is given and the major components are described in details.

4.1 Distributed Operation Layer

We implemented our framework as a component of Distributed Operation Layer (DOL), which is part of the framework of SHAPES European project [10][11]. The SHAPES project is proposed to overcome challenges that emerge in the design of multiprocessor embedded systems.

The DOL has as main inputs an application specification, an architecture specification and a mapping specification of the application onto the target architecture. The DOL aims at reducing significantly the effort of mapping applications onto multiprocessor hardware platforms under certain mapping constraints.

The primary components of DOL are the application model, the architecture model, the mapping model, and a design space exploration mechanism. As depicted in 3.2.1, the application is specified in the form of a restrict version of Kahn Process Network (KPN). The architecture specification in the DOL is an abstract description of a hardware platform, omitting implementation details in order to raise the design level. It consists of computation resources, communication resources and the interconnections among resources. The mapping specification in the DOL decides how the processes of the application are mapped onto computation resources, as well as how the software channels are mapped onto communication resources. The definitions for application, architecture and mapping are all written using XML schema [12].

4.1.1 Application Specification

In DOL, the process network computation model is used to model an application, where the structure of the process network is specified by the XML file, while the application behavior is specified by the associated C source code. Listing 4.1 is an example of a process network XML file. It contains three parts, the process description, the sw_channel description, and the point-to-point connections between process and the corresponding software channel. The detailed description of these parts is shown as follows:

```
<process name="generator" basename="generator">
  <port name="out" type="output" basename="out"/>
  <source location="generator.c" type="c"/>
</process>

...

<sw_channel name="C1" type="fifo" size="10" basename="C1">
  <port name="in" type="input" basename="in"/>
  <port name="out" type="output" basename="out"/>
</sw_channel>

...

<connection name="g-c">
  <origin name="generator">
    <port name="out" basename="out"/>
  </origin>
  <target name="C1">
    <port name="in" basename="in"/>
  </target>
</connection>
```

Listing 4.1: Process network XML file.

- process: The node *process* in the XML file corresponds to a process in the process network. The attribute *name* is required to serve as a unique identifier for a process. The child node *port* is used to connect the process with a software channel. A *process* node may contain one or more *port* nodes, which are divided into two types, namely *in* and *out*. The process can write data to port of type *out* and can read data from port of type *in*. The *source* node indicates the file containing the functional behaviour of the process. The attribute *type* specifies which type of programming language is used while the attribute *location* denotes where the source file is stored. For example, in Listing 4.1, the functionality of process *generator* is specified using the C programming language and the corresponding source file is *generator.c*.
- sw_channel: The node *sw_channel* describes a software channel in the process network. A software channel is the only way for two different processes to communicate with each other. The attribute *name* denotes the name of the software channel. The attribute *type* specifies the type of the software channel. Currently all the software channels have a classic FIFO behavior in DOL. Since in DOL's process network the software channel is bounded, an additional attribute *size* is used to specify the size of the FIFO. Two *port* nodes are defined to denote the input port and output port of this FIFO channel, respectively.
- connection: The node *connection* is used to explicitly specify all the connections between processes and software channels. The node *origin* and *target* can be either process or software channel. For example, in Listing 4.1, the connection named *g-c* connects process *generator* with the software channel *C1*. Besides, their respective port types determines that *generator* can write via the output port *out* to *C1*. *C1* can read via the input port *in*.

4.1.2 Architecture Specification

The architecture XML file is an abstract representation of the underlying HW platform. It includes all the HW characteristics useful for the performance evaluation. There are mainly two kinds of resources in the architecture: computation resources and communication resources. The way the resources are specified and how they connect with each other are presented in Listing 4.2.

- processor: The node *processor* describes a computation resource. The attribute *type* specifies the type of the processor, e.g. RISC or DSP. The child node *node* is a interface through which the computation resource can be connected to other

```
  <processor name="processor1" type="DSP">
    <node name="node">
      <duplexport name="processor_port"/>
    </node>
    <configuration name="clock" value="200 MHz"/>
  </processor>

...

  <hw_channel name="in_tile_link" type="BUS">
    <node name="node">
      <duplexport name="port1"/>
      <duplexport name="port2"/>
      <duplexport name="port3"/>
    </node>
    <configuration name="buswidth" value="32 bit"/>
    <configuration name="clock" value="200 MHz"/>
  </hw_channel>

...

  <connection name="processor1link">
    <origin name="processor1">
      <node name="node">
        <port name="processor_port"/>
      </node>
    </origin>
    <target name="in_tile_link">
      <node name="node">
        <port name="port1"/>
      </node>
    </target>
  </connection>

...
```

Listing 4.2: Architecture XML file.

components in the architecture. Another child node *configuration* can be used to specify performance characteristics of the resource, e.g. the clock frequency.

- hw_channel: The node *hw_channel* describes a communication resource, such as a bus. It is used to exchange data between different processors. Similarly, the child node *node* enables it to be connected with other resources. The parameters, e.g. bus width and clock frequency, are stored in the *configuration* node.
- connection: The node *connection* plays the role of interconnecting all the resources in the architecture together. The child node *origin* and *target* contain the two resources, which are supposed to connect to each other. They can be either computation or communication resources. The resources are connected via *node*.

4.1.3 Mapping Specification

The mapping specification links the application and the architecture. The mapping information can be split into two parts: computation mapping and communication mapping. The computation mapping specifies which process will execute on which computation resource, while the communication mapping denotes how to implement the communication at the architecture level. The mapping file also explicitly denotes all the utilized communication paths. Besides, various scheduling policies are as well included in the mapping file, for computation or communication resource sharing. Listing 4.3 illustrates the structure of the mapping file.

- path: A *path* specifies a possible communication path between two architecture resources. It consists of a series of resources, in which both *origin* and *target* should be computation resources. The resources between *origin* and *target* can be buses or shared memories. In addition, the child node *buffer* specifies where the communication buffer is located. The direction of the data flow accords with the order the resources in the path. For example, in Listing 4.3, the communication path *p1p2path* uses the shared memory *DXM* as its buffer, and it is composed of the following resources: *processor1*, *in_tile_link*, *DXM*, *processor2*.
- binding: The node *binding* can have two types: computation or communication binding. In the case of the computation binding, *origin* stands for a process in the process network and *target* describes a processor in the architecture. In the case of the communication binding, *origin* is a software channel while *target* is a communication path or a processor. If a software channel is mapped to a processor, it means internal communication.

```
  <path name="p1p2path">
    <origin name="processor1"/>
    <resource name="in_tile_link"/>
    <resource name="DXM"/>
    <resource name="in_tile_link"/>
    <target name="processor2"/>
    <buffer name="DXM"/>
  </path>

...

  <binding name="generator_binding" type="computation">
    <origin name="generator"/>
    <target name="processor2"/>
  </binding>

...

  <binding name="C2_binding" type="communication">
    <origin name="C2"/>
    <target name="p1p2path"/>
  </binding>

...

  <schedule name="processor1_schedule" type="fifo">
    <resource name="processor1"/>
    <origin name="square"/>
  </schedule>

  <schedule name="processor2_schedule" type="tdma">
    <resource name="processor2"/>
    <origin name="generator">
      <configuration name="startslot" value="1"/>
      <configuration name="numberofslots" value="1"/>
    </origin>
    <origin name="consumer">
      <configuration name="startslot" value="2"/>
      <configuration name="numberofslots" value="1"/>
    </origin>
    <configuration name="slotsonecycle" value="2"/>
    <configuration name="slotlength" value="30"/>
  </schedule>

  <schedule name="in_tile_link_schedule" type="fixedpriority">
    <resource name="in_tile_link"/>
    <origin name="processor1">
     <configuration name="priority" value="2"/>
    </origin>
    <origin name="processor2">
     <configuration name="priority" value="1"/>
    </origin>
  </schedule>
...
```

Listing 4.3: Mapping XML file.

- schedule: The node *schedule* is used to specify the scheduling policy and the configuration for a corresponding resource. Currently, three types of scheduling policies are implemented in our trace-based simulation framework, i.e. TDMA, FIFO and FP. The node *resource* indicates which resource this scheduling configuration is set for. The nodes *origin* are the processes or resources to be scheduled. For the scheduling on a processor, all processes that are bound to be correspondent processor will be scheduled. For the arbitration on a bus, all the resources connected to it can be scheduled. For instance, in the scheduling *processor1_schedule*, the process *square* will be scheduled by the processor *processor1* under the scheduling policy FIFO. In particular, if the scheduling type is FP, the configuration *priority* specifies the priority of the process and the larger value stands for higher priority. For instance, in the scheduling *in_tile_link*, the processor *processor1* is assigned a higher priority. If the scheduling type is TDMA, the configuration *slotsonecycle* stands for how many quantum slots within one TDMA cycle. The configuration *slotlength* stands for how many nanoseconds are assigned for each quantum slot. The configuration *startslot* stands for the start slot that is assigned to the process. The configuration *numberofslots* stands for how many quantum slots are allocated to the process. For instance, in the scheduling *processor2_schedule*, there are two quantum slots in one TDMA cycle. The length of each quantum is 30 ns, and the second slot is assigned to the process *consumer*.

4.1.4 Static Characterization

The static characterization file contains the performance estimation of each computation behaviour on each possible computation resource, as well as the time needed for writing or reading a word (4 bytes) on each processor. This file can be generated from the low-level model of architecture, using performance estimation tools, available documentation, or the experience of the designer. In the Listing 4.4, we present a segment of the static characterization file. It contains two parts: the time estimation for the computation behaviors of the processes and the time estimation for the communication behaviors.

- process: The node *process* stands for a process in the process network. The child node *computation* specifies a possible computation behavior in the process, identified by the start line and end line of a corresponding basic block in the source file. For instance, the pair $< 26, 28 >$ represents the computation behavior which starts at line *26* and ends at line *28* in the source file. In particular, the start line number *-1* stands for the start of the `fire()` function while the end line number *-1* stands for the end of the `fire()` function. Each computation behavior is characterized for all the possible resources where the process can run. The child node *processor* of node *computation* denotes the time needed to finish the computation event on this type of processor. The basic unit of the time is cycle. For example, it will take 60 cycles for a DSP processor to finish the computation behavior $< 26, 28 >$ of the process *generator*.

- communication: The node *communication* specifies the time estimation for internal communication on different processors. It has two types, namely *read* and *write*. The child node *processor* of node *communication* denotes the time needed to finish a writing or reading on this type of processor. The basic unit of the time is cycles per word. For example, it will take 2 cycles to read a word from the internal memory of the *DSP* processor.

```
  <process name="generator">
    <computation start="-1" end="26">
      <processor type="DSP" time="100"/>
      <processor type="RISC" time="150"/>
    </computation>
    <computation start="26" end="28">
      <processor type="DSP" time="60"/>
      <processor type="RISC" time="160"/>
    </computation>
    <computation start="28" end="-1">
      <processor type="DSP" time="80"/>
      <processor type="RISC" time="90"/>
    </computation>
  </process>
  <communication name="read">
    <processor type="DSP" time="2"/>
    <processor type="RISC" time="2"/>
  </communication>
  <communication name="write">
    <processor type="DSP" time="2"/>
    <processor type="RISC" time="2"/>
  </communication>
...
```

Listing 4.4: Static characterization XML file.

4.2 Trace Generation

In the DOL context, an application is modeled using the process network computation model stemming from the KPN. One of the most prominent characteristics of this computation model is its deterministic nature, indicating that given a same input the outputs will be the same regardless of the timing of the processes of the network. This characteristic builds the basic rationale of our framework where the timing of processes is abstracted as high-level trace events and the data dependency among processes is captured as a partial-order set of these trace events. This partial-order set is determinate for a given application and independent of the hardware resources as well as scheduling policies of any target architectures that this application is mapped to.

In the process network of the DOL, there is no global memory that can be used by the processes to communicate. Two processes can exchange data only by a bounded FIFO channel that connects these two processes. Other processes which are not connected to the corresponding FIFO channel cannot access it. The DOL provides two API primitives, i.e. `DOL_write()` and `DOL_read()`, for the writing to or reading from the software channel, respectively. The software channel in DOL process network is limited in size. As a consequence, both `DOL_write()` and `DOL_read()` are blocking primitives. The `DOL_write()` will block when the data in the buffer exceeds the upper limit of the software channel. The `DOL_read()` will block when the FIFO buffer is empty. The two communication APIs `DOL_write` and `DOL_read` are described in Listing 4.5:

```
API: int DOL_write(void *port, void *buf, int len, DOLProcess *p)
Description: DOL_write() writes len bytes from buffer pointed by buf to port.

API: int DOL_read(void *port,void *buf, int len, DOLProcess *p)
Description: DOL_read() reads len bytes from {port} into the buffer
             pointed by buf.
```

Listing 4.5: DOL APIs definition.

Functional Simulation

To verify the application specifications at the very beginning of the whole design cycle, a functional simulator is designed in the DOL context. Taking only the specifications of an application as inputs, the DOL generates a SystemC-based functional simulator which can be executed in a LINUX host machine. The simulator executes the application code and simulates the functionality of the application, neglecting the effects caused by the target architecture. In this way, functional bugs can be exposed and debugged at an early stage without interferences of low level details, e.g. OS and scheduling policies.

The functional simulation in DOL is implemented in SystemC. The first step in the DOL functional simulation is to generate the SystemC package automatically, from the process network specification and the source codes. To achieve this, DOL creates a wrapper file for each process. The wrapper adaption does the SystemC simulation. The DOL APIs, i.e. `DOL_write` and `DOL_read`, are defined in the wrapper file and are controllable at the simulation level. Thus it is possible to modify the DOL APIs in order to track the communication behavior in the functional simulation, and implicitly the computation behavior. In the wrapper file, we redefine the `DOL_write` and `DOL_read`, as depicted in Listing 4.6:

According to the API redefinition in Listing 4.6, `DOL_write()` called in the C source file will be replaced by a series of instructions and function calls. Take `DOL_write()` as an example. After the redefinition, the original `DOL_write()` call is maintained, thus the previous functionality of the process is not affected. The additional instructions and functional calls serve to trace the execution of the process. The computation behavior between two successive communication calls can be identified by `start_line` and `end_line`. The method `create_computation_event` will create a computation event for this process after each computation. The method `create_write_event` will create a write event each time `DOL_write` is called. Similarly, `DOL_read` is redefined so that the read behavior can be tracked.

Trace Format

After running once the instrumented functional simulation of the application, traces for each process are generated and stored in a trace file. Three types of trace events are generated, namely computation events, write events and read events. The format of the trace file is illustrated in Listing 4.7.

In the trace file, the events of a certain process are arranged consecutively in the order they are executed. For each process, the trace has the following elements. It starts with a line specifying the name of the process, identified by a special character *$*. After that, each line stands for a trace event until the trace of another process is reached. And the first element of a trace event is the identifier. For example, the line of computation event has *c* as its first character. The next two integer numbers are the *start_line* and the *end_line* which identify a computation segment in the source code. The line of a write event has *w* as its identifier. The next number records the number of data to be write, while the

```
static inline int DOL_write(void *port,
                            void *buf,
                            int len,
                            DOLProcess *process)
{
    sc_port<write_if> *write_port
        = static_cast<sc_port<write_if> *>(port);
    char *str = static_cast<char*>(buf);
    while (len -- > 0)
        (*write_port)->write(*str++);
#ifdef INCLUDE_TRACE
    strcpy(channel_name,
           dynamic_cast<fifo *>(write_port->get_interface())->basename());
#endif
}

#ifdef INCLUDE_TRACE
#define DOL_write(port, buf, len, process) end_line = __LINE__; \
dol_functional_trace.create_computation_event( \
              (static_cast<square_wrapper *>(p->wptr))->basename(), \
              start_line, end_line); \
DOL_write(port, buf, len, process); \
dol_functional_trace.create_write_event( \
              (static_cast<square_wrapper *>(p->wptr))->basename(), \
              len, channel_name); \
start_line = __LINE__;
#endif

static inline int DOL_read(void *port,
                           void *buf, int len,
                           DOLProcess *process)
{
    sc_port<read_if> *read_port
        = static_cast<sc_port<read_if> *>(port);
    char *str = static_cast<char*>(buf);
    while (len -- > 0)
        (*read_port)->read(*str++);
#ifdef INCLUDE_TRACE
    strcpy(channel_name,
          dynamic_cast<fifo *>(read_port->get_interface())->basename());
#endif
}

#ifdef INCLUDE_TRACE
#define DOL_read(port, buf, len, process) end_line = __LINE__;\
dol_functional_trace.create_computation_event( \
              (static_cast<square_wrapper *>(p->wptr))->basename(), \
              start_line, end_line);\
DOL_read(port, buf, len, process);\
dol_functional_trace.create_read_event( \
              (static_cast<square_wrapper *>(p->wptr))->basename(), \
              len, channel_name); \
start_line = __LINE__;
#endif
```

Listing 4.6: DOL APIs redefinition.

```
\$ generator
c −1 32
w 8 C1
c 32 −1
c −1 32
w 8 C1
c 32 −1
c −1 32
w 8 C1
c 32 −1
c −1 32
w 8 C1
c 32 −1

....
```

Listing 4.7: Trace format in the trace file.

string indicates the name of the channel to write to. The line of a read event has r as its first character. The next number records the number of data to be read from the channel, while the string tracks the name of channel to read from.

4.3 Trace-based Simulation

We implement the trace-based simulation in SystemC. The event-trigger mechanism is used to realize synchronization between differnt components. The `wait()` function call is used to model the execution delays of the trace events.

4.3.1 Class Diagram

The class diagram of the system is depicted in Figure 4.1. It consists of three parts: application layer, architecture layer and mapping layer. The application layer is automatically contructed from the trace file and the process network specification. The architecture layer is automatically constructed from the architecture specification. The mapping layer is used to bind the application and architecture together. They are explained in detail in the next sections.

4.3.2 Application Layer

The application layer has 3 major classes, i.e. the *Application* class, the *Process* class and the *SW_channel* class. Their respective descriptions are shown as follows.

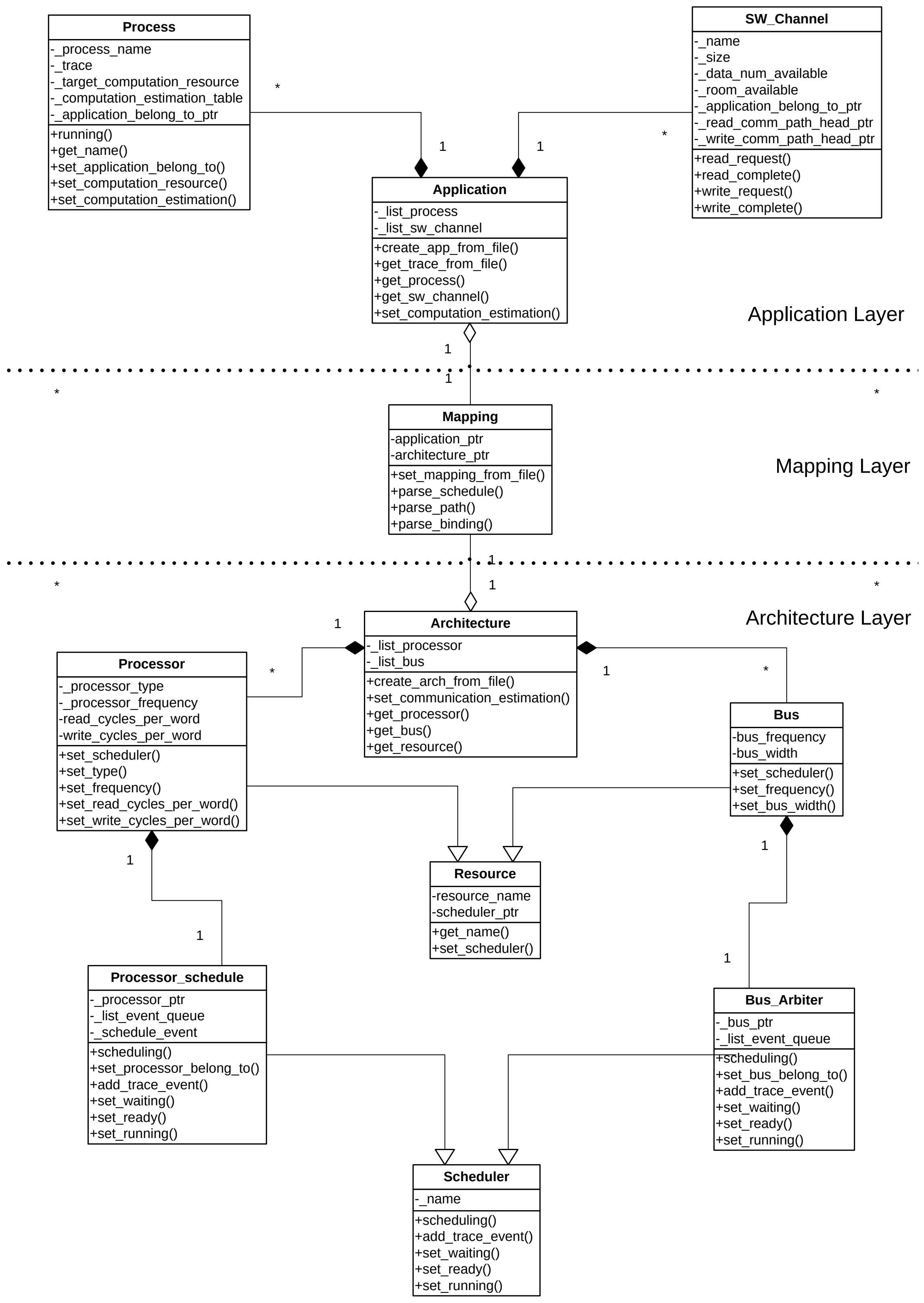


Figure 4.1: Class diagram of the trace-based simulation.

Application Component

As is depicted in Figure 4.1, the class `Application` is used to create processes and software channels, as well as to manage them.

To create processes and software channels, the method `create_app_from_file()` takes the process network xml file as input. By automatically parsing the xml file, the names and the configurations of processes and software channels are retrieved. The list of processes is then created and stored in the member `_list_process`. The list of software channels is created and stored in the member `_list_sw_channel`. The method `get_trace_from_file()` reads traces from the trace file and attaches them to the corresponding processes.

Another functionality of the class `Application` class is to manage the processes and software channels. For example, a call to the `get_process()` method will return the pointer to a certain process, given the name of the process. This is an important service for those components that need to access certain processes.

In addition, the method `set_computation_estimation()` updates the performance estimation of the computation behavior for each new mapping. Since the time needed to finish a computation event is dependent on the target computation resource, it will change when the mapping is changed, e.g. when the process is mapped to a different processor or when the processor's characteristic change. The method `set_computation_estimation()` takes a characterization file as input, in which the performance estimations for each computation event on each type of processor are specified. The run time information is updated for each process.

Process Component

The process is modeled by its execution trace generated during the functional simulation. The process is represented in the diagram by the *Process* class. During the trace-based simulation, the *Process* class dispatches the trace events to the architecture dynamically.

The trace of a process is stored in the member `_trace` of the `Process` class. It is implemented as a linked list, in which each node stands for a trace event. Since the trace can be large in the memory, the benefit of storing the trace as a linked list is that large continuous memory is not required. Besides, because there is no random access to the trace event of the trace, the speed of traversing the list will not be a drawback.

As is illustrated in Figure 4.2, each process has a performance table, which maintains the performance estimation of the computation behaviors for current mapping. In the class `Process`, the performance table is the member `_computation_estimation_table`. Each computation behavior has an entry in the table. The right column of the entry specifies how many cycles it will take the target processor to complete the computation behavior. Each computation trace event holds a pointer pointing to the entry it belongs. In this way, for a different mapping, only the performance table needs to be updated. This will increase the efficiency of the simulation because the timing of the computation event can be directly accessed without time-consuming queries.

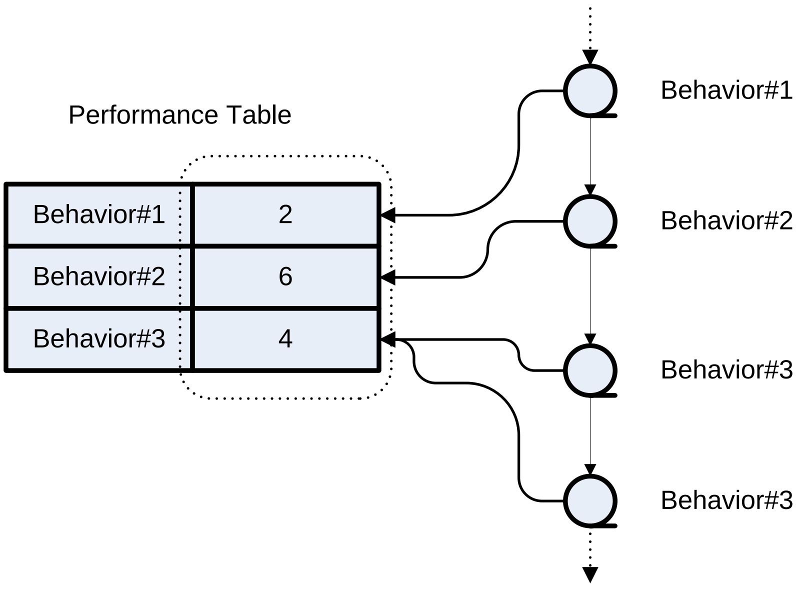


Figure 4.2: Performance table.

The `Running()` method of class `Process` delivers trace events to the target processor to trigger the simulation of the system. The method is registered as a SystemC thread, which means that it will be automatically called after the simulation starts. The computation events of a process will be directly dispatched to the processor onto which the process is mapped. For the communication events, instead of directly dispatching it to the processor, the process will first consult to the software channel to check whether or not the communication is safe (i.e. for read event, data are available, for write event, free rooms are available). Only if it is safe, it will be dispatched. Otherwise, the process will block.

SW_Channel Component

The software channel class is only an abstract description of the application communication channels, without modeling the actual data transfer. It is used to express the dependencies between two trace events who communicate with each other through the software channel.

In the class diagram, the software channel is represented by the *SW_Channel* class.

The class `SW_channel` has two important members, i.e. `_data_num_available` and `_room_available`, which are used to represent the current state of the software channel. The member `_data_num_available` stands for how many data items are available for reading while `_room_available` records the number of free space can be used for writing. The values of these two members can be changed by the following four methods:

- The method `read_request()` is called by the method `Running()` of the class `Process`. It checks if the available data can satisfy the read request. If not, the process who calls this `read_request` will block. Otherwise, it will subtract the number of data requested from the member `data_num_available` and return immediately.
- The method `read_complete()` is called by a certain resource of the architecture when a read event is finished. In this case, the member `room_available` is increased by the number of data in the read event. In addition, if there is process blocking on writing to this software channel, it will be waked up. If the available room still can not meet the write request, the process will remain blocking. Otherwise, the process will resume and dispatch the write event to the architecture.
- The method `write_request()` is also called by the method `Running()` of the class `Process`. It checks if the available room can meet the write request. If not, the process who calls this `write_request` will block. Otherwise, it will subtract the number of room requested from the member `room_available` and return immediately.
- The method `write_complete()` is called by some resource of the architecture when a write event is finished. The member `data_num_available` is increased by the number of data in the write event. If there is process blocking on reading from the software channel, it will be waked up. If the data available still can not meet the read request, the process will block again. Otherwise, the process will resume and dispatch the read event to the architecture.

4.3.3 Architecture Layer

As is depicted in Figure 4.1, the architecture mainly consists of three classes, i.e. the *Architecture* class, the *Resource* class and the *Scheduler* class. They are described as follows.

Architecture Component

The `Architecture` class is used to create and manage the resources, including processors and buses. The method `create_arch_from_file()` reads the configuration of the hardware platform from the architecture XML file. It then parses the parameters of the resources and creates them. The processors will be stored in the member `_list_processor` while the buses will be put in the member `_list_bus`. There are a series of methods which help managing the resources. For example, the call to the method `get_processor` with the name of a certain processor will return the pointer pointing to that processor.

Resource Component

The abstract base class `Resource` defines the common members and behaviors for all the resources in the architecture. One important member is `scheduler_ptr` that points to the scheduler attached to this resource. Here, we separate the implementations of resources and schedulers for the sake of flexibility. By means of this separation, the framework is flexible to adopt new scheduling policies. Both the class `Processor` and class `Bus` derive from this base class. Besides, in order to extend the architecture and introduce new resource, we only need to define a new class that inherits from the abstract base class `Resource`. Here we explain the two resources implemented.

- The `Processor` class models the processor, e.g. DSP or RISC, in the architecture. The type of the processor is specified in the member `_processor_type`. The frequency of the processor is stored in the member `_processor_frequency`. Besides, a processor scheduler is attached to the class `Processor` so that the access requests to the processor from different processes can be scheduled according to a given scheduling policy.
- The `Bus` class models the communication bus. The clock frequency of the bus is specified by the member `bus_frequency`. The width of the bus is stored in the member `bus_width`. A bus arbiter is attached to the bus to provide exclusively mutual access.

Scheduler Component

The abstract base class `Scheduler` defines the common interfaces for the schedulers. Both processor scheduler and bus arbiter inherit from this base class. Currently, three scheduling policies have been implemented, i.e. TDMA, FIFO and FP.

The data structure EVENT_QUEUE in Listing 4.8 is used to stored the trace events that need to be processed by the resource. The member _list_event_queue of the scheduler stores all the event queues, which can be scheduled according to the specified scheduling policy. In the data structure EVENT_QUEUE, some members are used to specify the parameters for a special scheduling policy. For example, the member priority denotes the priority of the event queue in the fixed-priority scheduler. The member time_slice specify the time slice for this event queue in the TDMA scheduler. As discussed in Section 3.3.4, the event queue has three states in the state machine. In the data structure, the state of the event queue is stored in the member state. The trace events for this event queue is stored in the member list_trace_event.

```
typedef struct EVENT_QUEUE
{
    char name[NAME_LENGTH];

    /*For fixed priority.*/
    int priority;

    /*For TDMA.*/
    double time_slice;

    /*For state transformation.*/
    int state;
    sc_event schedulable_event;

    /*trace event to be scheduled*/
    list<TRACE_EVENT *> list_trace_event;
    list<TRACE_EVENT *>::iterator iter_trace_event;
}EVENT_QUEUE;
```

Listing 4.8: The EVENT_QUEUE data structure.

In the method Scheduling(), the event queues are scheduled by the processor scheduler or the bus arbiter. The implementation of the Scheduling() methode is different for two reasons. On the one hand, the bus arbiter does not handle computation events as processor scheduler does. On the other hand, the method of measuring timing of the trace event is different. For example, the time needed to finish a read event on a bus is dependent on some parameters of the bus, e.g. clock frequency and bus width, while on a processor it is determined by the clock frequency and the processor types. For different scheduling policies, the implementation of the Scheduling() method is also different, because different scheduling policies have different algorithms.

The state transitions of an event queue in the scheduler are carried out by the following methods: set_ready(), set_running() and set_waiting(). For example, the method set_ready will change the current state of the event queue to *ready* while the method `set_waiting()` will change the current state of the event queue to *waiting*. They are implemented differently for various scheduling policies.

4.3.4 Mapping Layer

The mapping layer is used to associate the application with the architecture. This is realized in the *Mapping* class. The method `set_mapping_from_file()` reads mapping information from the mapping file. It then calls the following methods to set the mapping configuration:

- The method `parse_schedule()` creates schedulers for the corresponding resources. It also creates event queues that need to be scheduled by the scheduling. The relative parameters, such as the priority of an event queue in a fixed priority scheduler, are attached to each event queue.
- The method `parse_path()` creates the communication paths specified in the mapping file. The paths are used to connect different communication resources together in order to conduct a data transfer.
- The method `parse_binding()` connects the processes of the application and the processors of the architecture. It also links the software channels and the communication paths.

4.3.5 Conclusion

The trace-based simulation framework implemented consists of two phases. In the first phase, we modify the previous DOL functional simulation model to make it able to automatically generate traces. The traces can then be reused in the design space exploration. In the second phase, we implement the trace-based simulation in SystemC. To model the architecture, we adopt the uniform resource definition. Both processor and bus are defined by the abstract basic class *Resource*.Thus the architecture in the framework can be modeled in a modular and flexible way. In addition, all the inputs of the trace-based simulation, e.g. the process network, the architecture and the mapping, are specified in the form of XML file under certain scheme. In this way, the inputs are configurable and easy to modify.

5 Case Study

We have already presented framework of our trace-based simulation in Chapter 3 and its implementation aspects in Chapter 4. In this chapter, we focus on using two compelling case studies to verify the framework and to prove its efficiency: the producer-consumer application and the MPEG-2 decoder [13][14][15]. In Section 5.1, by investigating different mappings for the producer-consumer application, several important functionalities of the framework are testified, such as various scheduling policies, the ability to handle different locations of the software buffer and the capability to handle communication via a path consisting a sequence of resources. In Section 5.2, a more complex application, namely the MPEG-2 decoder, is used to test the efficiency and other characters of the framework.

5.1 The Producer-consumer Case Study

In this case study, we use a simple example, i.e. producer-consumer application, to show how our trace-based simulation framework capture the performance impacts caused by different mappings of this application, e.g. scheduling policies, locations of communication buffer, communication paths and resource bindings.

5.1.1 Application and Architecture Description

The producer-consumer application is written using the process network specification scheme of DOL. It consists of three processes, i.e. *generator*, *square* and *consumer*, and two software channels connecting them, i.e. *C1* and *C2*. The buffer sizes of *C1* and *C2* are both 10 bytes. The process *generator* generates data items, which are then written to *C1*. The process *square* reads data from *C1*, computes the square of the number and writes the result to *C2*. The process *consumer* reads data from *C2* and displays it on the terminal screen. Figure 5.1 depicts the structure of the producer-consumer process network.

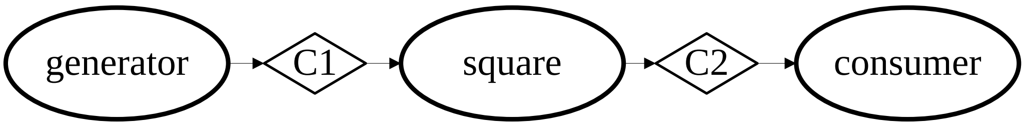


Figure 5.1: The producer-consumer application used in our experiment.

The default hardware platform used in our experiments is made of two processors, i.e. *processor1* and *processor2*, one external bus, i.e. *in_tile_link*, and one shared memory. The structure of the architecture is described in Figure 5.2. The configuration of the architecture is shown in Table 5.1. This is the basic structure of a tile architecture. During our experiments, we will use more complex architecture composed of several basic tiles interconnecting via the inter-tile communication backbone. In particular, the default atomic data transaction size is 4 bytes.

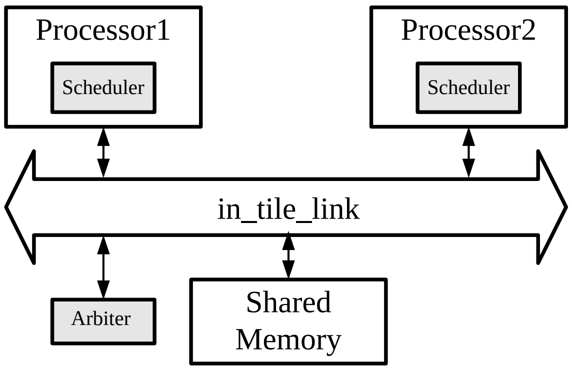


Figure 5.2: The one tile architecture used in our experiment.

Resource	Configuration
processor1	frequency: 200M.
processor2	frequency: 200M.
in_tile_link	frequency: 200M. width: 32bit.

Table 5.1: Configuration of the architecture.

5.1.2 Mapping Exploration Overview

We do the design space exploration for the producer-consumer application by changing various parameters, e.g. the different scheduling policies, the different locations of communication buffer, the communication path and the different resource bindings. The experimental settings and exploration results are listed and analyzed as follows.

5.1.3 Scheduling Policy

In this section, we will explore three different scheduling policies for the producer-consumer application, i.e. TDMA, FIFO and FP.

TDMA

Example 1. *The first test case is depicted in Figure 5.3, in which the producer-consumer application is mapped to the single tile architecture. As seen in Figure 5.3, both* generator *and* consumer *are mapped to* processor2*,* square *to* processor1*,* C1 *to the communication path* $(processor2, in\_tile\_link, processor1)$ *and C2 to the communication path* $(processor1, in\_tile\_link, processor2)$*. Both the buffer of* C1 *and that of* C2 *are located in the target processors. In particular, the scheduling policies adopted by all the resources are TDMA. The detailed scheduling configuration information is shown in Table 5.2. For example, in* processor2*, the second time slot is assigned to* consumer*.*

Resource	Scheduling	Configuration
processor1	TDMA	quantum_slot_length: 30ns, square←1
processor2	TDMA	quantum_slot_length: 30ns, generator←1,consumer←2
in_tile_link	TDMA	quantum_slot_length: 30ns, processor1←1, processor2←2

Table 5.2: Scheduling configuration for Example 1.

We utilize the VCD waveform tracing utility provided by SystemC to track the execution

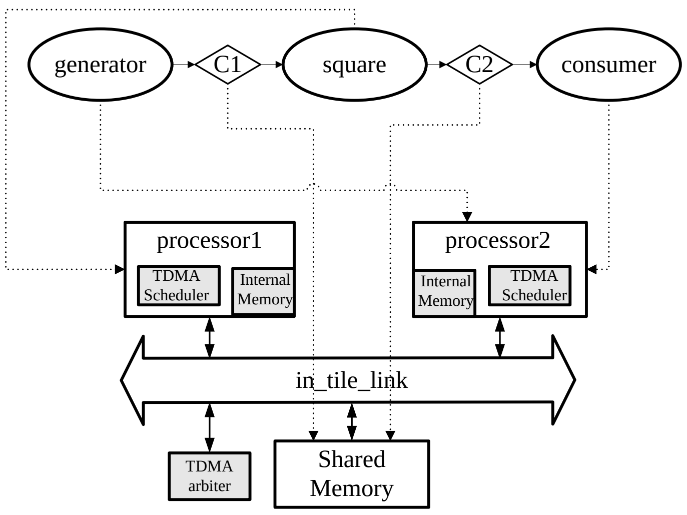


Figure 5.3: The mapping specification in Example 1 where the scheduling policy is TDMA for all resource.

procedure of the input traces on the target architecture. The generated waveform is listed in Figure 5.4, in which the waves stands for the execution states of the processes. The high period represents that the process is being processed by the target resource while the low period depicts the inactive state of the corresponding process. Take the variable *consumer_compute* as an example. What can be deduced from this wave is that the target processor *processor2* starts to execute the process *consumer* at the time stamp of 30 ns and halts its execution at time stamp of 50 ns. All the other waves can be interpreted in the same manner, such as *consumer_read_on_processor* and *consumer_read_through_buses*. The wave *consumer_read_on_processor* represents the execution of the reading of *consumer* on the target processor. The wave *consumer_read_through_buses* stands for the execution of the reading of *consumer* via the buses. The important phenomena of the waveform are indicated in the figure and the corresponding analysis is shown as follows:

1. At this moment, even though the process *generator* is still ready, it is stalled by the processor because its time slot is used up and the next time slot does not belong to it.
2. At this moment, the time slot for *generator* arrived and *generator* is selected by *processor2* to run again.
3. The series of arrows denote the data flow direction of the first four bytes which are transferred among the processes.
4. For the same write event with 4 bytes of *generator*, the time consumed on the

processor *processor2* is larger than that consumed on the bus *in_tile_link*. The reason is that accessing local memory is faster than accessing the external memory.

5. The processor *processor2* does not block when the bus *in_tile_link* is transferring data. The reason is that we adopt an asynchronous communication manner. Thus these two resources which can run concurrently.
6. Since the process *generator* and the process *square* are mapped to two different processors, they can run concurrently.
7. Since *generator* and *consumer* are both mapped to the same processor, they access *processor2* exclusively.

Another output of the trace-based simulation is the performance statistics as presented in Listing 5.1. It contains important performance numbers. The following performance data are derived from this experiment.

- The estimated runtime T_{system} for executing the entire application is 3815 ns. It denotes the total time of the application in order to finish its execution on the specific architecture, and under the specific mapping. It is an important indicator of the system performance.
- The load of the processor *processor1* $L_{\text{processor1}}$ is 2020 ns. It is defined as the total time consumed by all the processes mapped onto the corresponding processor. The processor load reflects the usage of the processor.
- The load of the processor *processor2* $L_{\text{processor2}}$ is 2420 ns.
- The load of the bus *in_tile_link* is $L_{\text{in\_tile\_link}}$ 2400 ns. It is defined as the total time when the bus is busy with transferring data. The bus load reflects the usage of the bus.
- The maximum processing time of all processors is 2420 ns.
- The time needed to execute the process *generator* on the processor *processor2* $T_{\text{generator,processor2}}$ is 1200 ns. It denotes how long it will take the target processor to finish this process. The process execution time can help the designers to decide which process demands more computation time.
- The time needed to execute the process *consumer* on the processor *processor2* $T_{\text{consumer,processor2}}$ is 1220 ns.
- The time needed to execute *square* on *processor1* $T_{\text{square,processor1}}$ is 2020 ns. It denotes the processor load of the processor which has the largest processing time

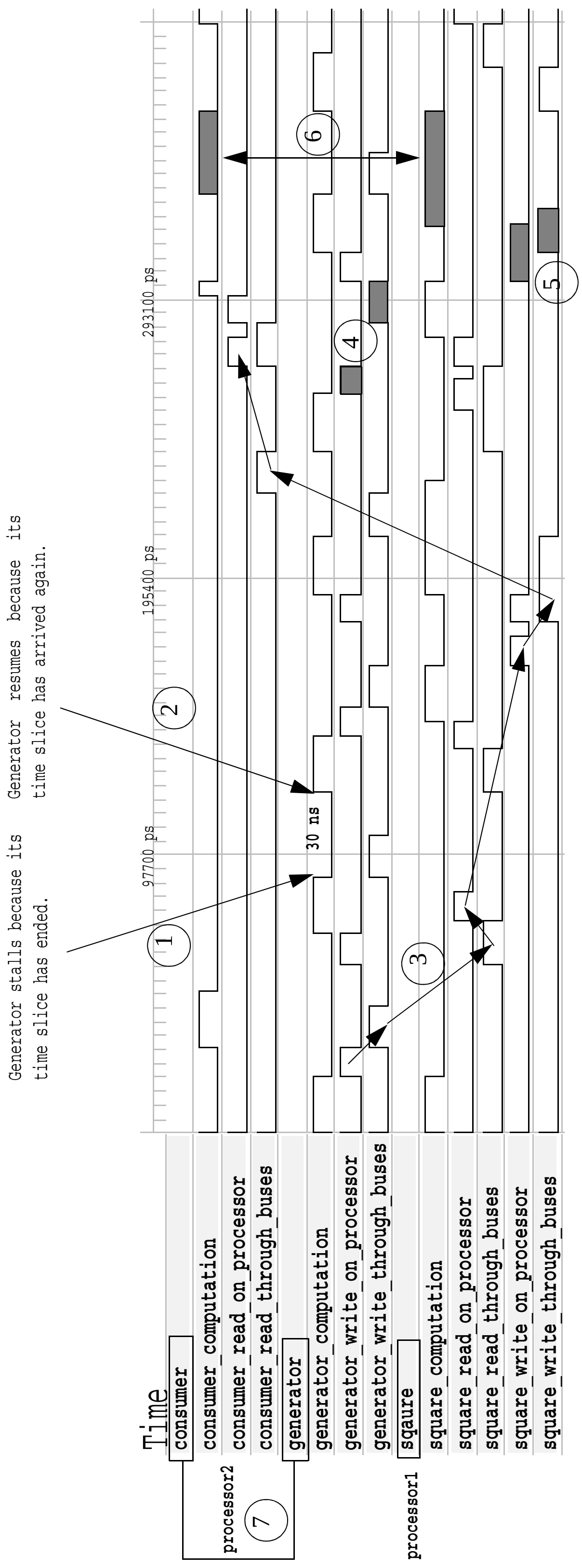


Figure 5.4: VCD waveform output for Example 1.

```
Performance statistics:
------------------------------------------------
process: generator
processing time on processor (NS): 1.200000e+03
processing time on buses (NS): 6.000000e+02
------------------------------------------------
process: consumer
processing time on processor (NS): 1.220000e+03
processing time on buses (NS): 6.000000e+02
------------------------------------------------
process: square
processing time on processor (NS): 2.020000e+03
processing time on buses (NS): 1.200000e+03
------------------------------------------------
processor: processor1
processing time (NS): 2.020000e+03
------------------------------------------------
processor: processor2
processing time (NS): 2.420000e+03
------------------------------------------------
bus: in_tile_link
processing time (NS): 2.400000e+03
------------------------------------------------
Estimated execution time (NS): 3.815000e+03
------------------------------------------------
Max processing time of processors (NS): 2.420000e+03
------------------------------------------------
Max processing time of buses (NS): 2.400000e+03
------------------------------------------------
```

Listing 5.1: Performance statistics for Example 1.

among all the computation resources in the architecture. Since the time needed to finish the application on the target architecture is closely related with the processing time on each hardware resource, in order to improve the system performance, we should increase the degree of load balance among the processors as much as possible.

- The max processing time of buses is 2400 ns. It denotes the bus load of the bus which has the largest bus load among all the communication resources in the architecture. If one bus is overloaded because of too many data items to transfer, the overall performance may degrade.
- The time needed for handling the communication on the bus *in_tile_link* from *generator* $T_{\text{generator,in\_tile\_link}}$ is 600 ns.
- The time needed for handling the communication from *consumer* on *in_tile_link* $T_{\text{consumer,in\_tile\_link}}$ is 600 ns.
- The time needed for handling the communication on *in_tile_link* from *square* $T_{\text{square,in\_tile\_link}}$ is 1200 ns.

It is self-evident that the following constraints hold for the above performance numbers:

$$\begin{aligned} L_{\text{processor1}} &= T_{\text{square,processor1}} \\ L_{\text{processor2}} &= T_{\text{generator,processor2}} + T_{\text{consumer,processor2}} \\ L_{\text{in\_tile\_link}} &= T_{\text{generator,in\_tile\_link}} + T_{\text{consumer,in\_tile\_link}} + T_{\text{square,in\_tile\_link}} \end{aligned} \tag{5.1a}$$

In general, for a processor R, if there are n processes mapped onto it, namely $p_1, p_2, ..., p_n$, the following constraint holds for the load of R, i.e. L_{R}, the timed needed for p_i to execute on R, i.e $T_{p_i,R}$.

$$L_{\text{R}} = \sum_{i=0}^{n} T_{\text{p}_\text{i},\text{R}}$$

The performance numbers can be used to guide the designers in the design space exploration. In our experiments, by comparing the performance numbers of different mappings, we check the functionality of the application and verify certain constraints.

FIFO

Example 2. *To testify the FIFO scheduling policy, we map the producer-consumer application described in Figure 5.1 onto the architecture of a single tile presented in Figure 5.2. The mapping is depicted in Figure 5.5. The scheduling configuration is described in Table 5.3: all the architecture resources, i.e.* processor1, processor2 *and* in_tile_link, *employ FIFO scheduling policy.*

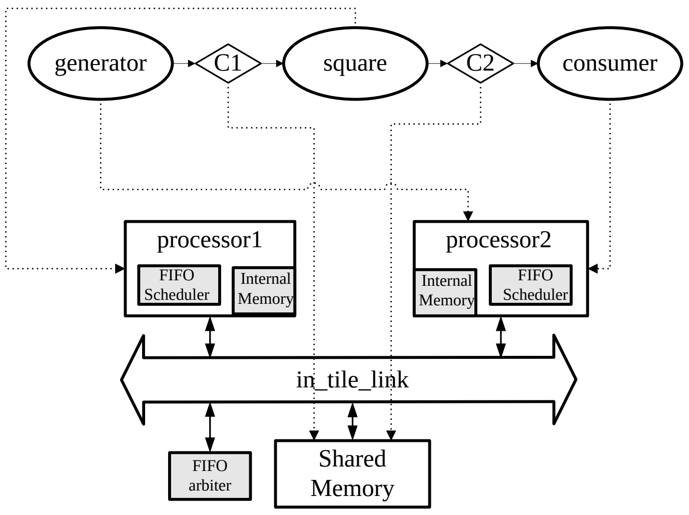


Figure 5.5: The mapping specification in Example 2 where the scheduling policy is FIFO for all resource.

Resource	Scheduling	Configuration
processor1	FIFO	square
processor2	FIFO	generator, consumer
in_tile_link	FIFO	processor1, processor2

Table 5.3: Scheduling configuration for Example 2.

The output waveform for Example 2 is depicted in Figure 5.6. The important phenomena of the waveform are indicated in the figure and the corresponding analysis is shown as follows:

1. The process *generator* and the process *consumer* are scheduled by the processor *processor* according to FIFO scheduling policy. Therefore, *generator* and *consumer* never preempt each other. Once one of them gets *processor2*, it will continue its execution, until there is no data in its input channel.

```
Performance statistics:
--------------------------------------------
process: generator
processing time on processor (NS): 1200
communication time through buses (NS): 600
--------------------------------------------
process: consumer
processing time on processor (NS): 1220
communication time through buses (NS): 600
--------------------------------------------
process: square
processing time on processor (NS): 2020
communication time through buses (NS): 1200
--------------------------------------------
processor: processor1
processing time (NS): 2020
--------------------------------------------
processor: processor2
processing time (NS): 2420
--------------------------------------------
bus: in_tile_link
processing time (NS): 2400
--------------------------------------------
Estimated execution time (NS): 2540
--------------------------------------------
Max processing time of processors (NS): 2420
--------------------------------------------
Max processing time of buses (NS): 2400
--------------------------------------------
```

Listing 5.2: Performance statistics for Example 2.

2. The process *square* monopolies the processor *square*. In this circumstances, *square* can be selected by the scheduler to run whenever it is ready.

The output performance statistics table for Example 2 is depicted in Listing 5.2. By comparing with Example 1, we can draw the following conclusions:

- The loads of all resources keep the same in that the resource binding does not change. For a certain processor, the processes mapped onto it are the same.
- The total execution time decreases from 3815 ns to 2540 ns. The reason is that in Example 1, the scheduling policies for all resources are TDMA. In TDMA, each time slot is fixed to a certain process. If the process is not ready to run, the time slot can not be used by any other ready process and will be wasted. Therefore, the resources are not fully utilized. By contrast, the resources in this example are better utilized.

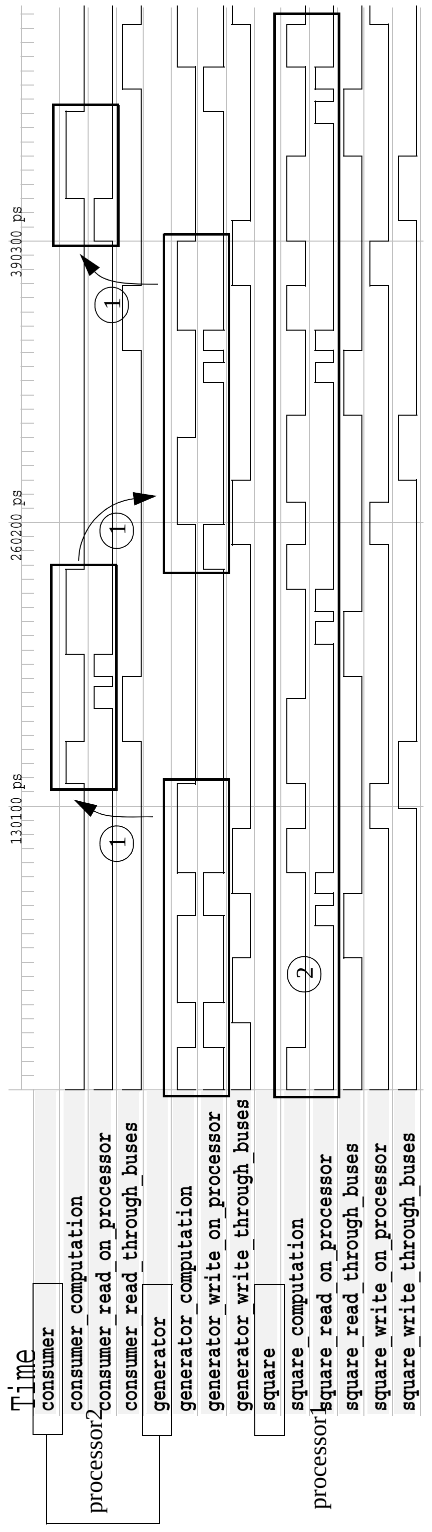


Figure 5.6: VCD waveform output for Example 2.

Fixed Priority

Example 3. *The third example is depicted in Figure 5.7 to testify the FP scheduling policy. The resource binding is the same as the previous two examples. The scheduling configuration is described in Table 5.4. The scheduling policies for all resources are FP.*

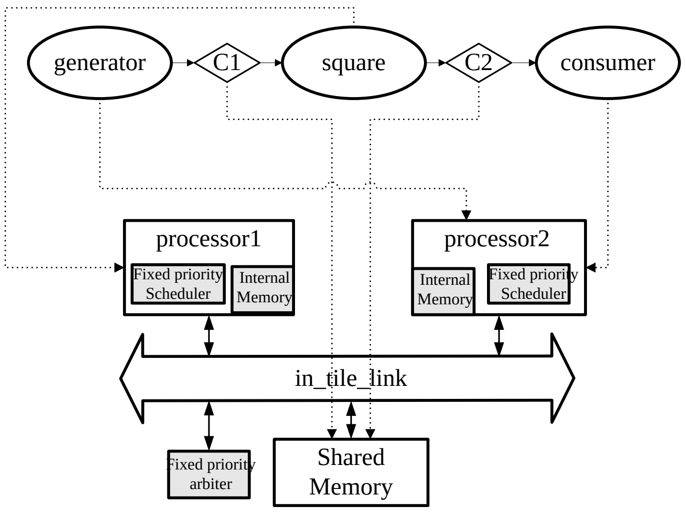


Figure 5.7: The mapping specification in Example 3 where the scheduling policy is FP for all resource.

Resource	Scheduling	Configuration
processor1	FP	square ← 1
processor2	FP	generator ← 1,consumer ← 2
in_tile_link	FP	processor1 ← 2, processor2 ← 1

Table 5.4: Scheduling configuration for Example 3.

The output waveform for Example 3 is depicted in Figure 5.8. The important phenomena of the waveform are indicated in the figure and the corresponding analysis is shown as follows:

1. At the beginning, even though both consumer and generator are ready, consumer is selected to run by processor2 because it has a higher priority.
2. At this moment, the data is available for consumer to read. Since it has higher priority, the consumer preempts the execution of generator. The arrow denotes the context switch.

```
Performance statistics:
--------------------------------------------------
process: generator
processing time on processor (NS): 1.200000e+03
processing time on buses (NS): 6.000000e+02
--------------------------------------------------
process: consumer
processing time on processor (NS): 1.220000e+03
processing time on buses (NS): 6.000000e+02
--------------------------------------------------
process: square
processing time on processor (NS): 2.020000e+03
processing time on buses (NS): 1.200000e+03
--------------------------------------------------
processor: processor1
processing time (NS): 2.020000e+03
--------------------------------------------------
processor: processor2
processing time (NS): 2.420000e+03
--------------------------------------------------
bus: in_tile_link
processing time (NS): 2.400000e+03
--------------------------------------------------
Estimated execution time (NS): 2.560000e+03
--------------------------------------------------
Max processing time of processors (NS): 2.420000e+03
--------------------------------------------------
Max processing time of buses (NS): 2.400000e+03
--------------------------------------------------
```

Listing 5.3: Performance statistics for Example 3.

3. This is another context switch caused by preemption for the same reason.

The output performance statistics table for Example 3 is depicted in Listing 5.3. Since the resource binding does not change, the loads of all resources keep the same. The total execution time is similar to Example 2, but is smaller than Example 1. The reason is that the resources are better utilized.

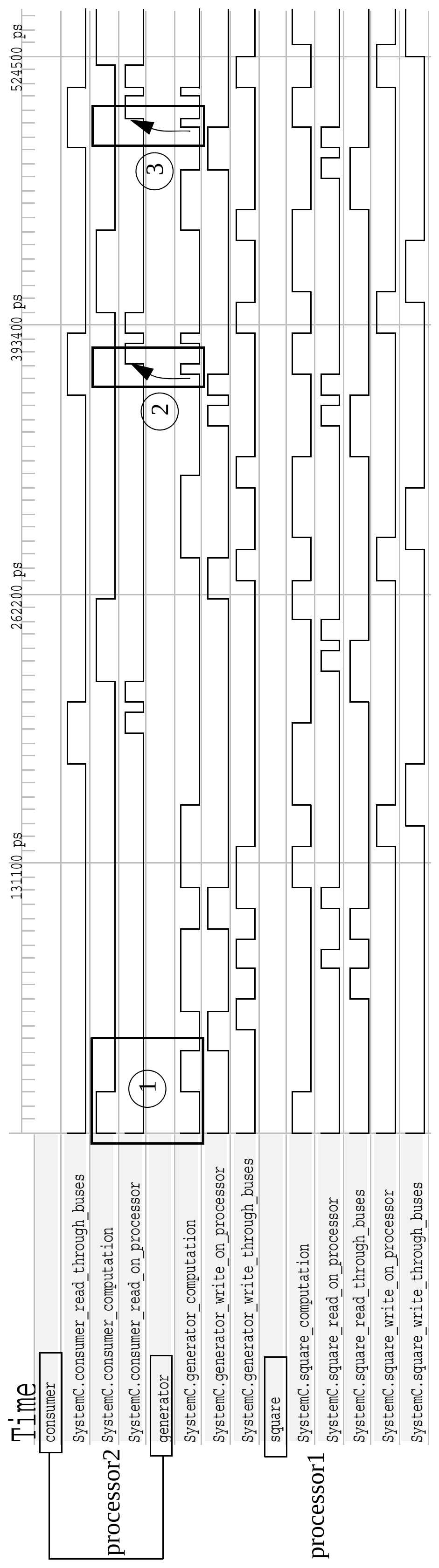


Figure 5.8: VCD waveform output for Example 3.

5.1.4 Communication Buffer Location

In this section, we will explore different communication buffer locations for the producer-consumer application, namely origin processor, target processor and shared memory.

Origin Processor

Example 4. *First of all, in this mapping, we investigate the scenario when the software buffer is located in the origin processor. The mapping is depicted in Figure 5.9. The scheduling configuration of the mapping is presented in Table 5.5.*

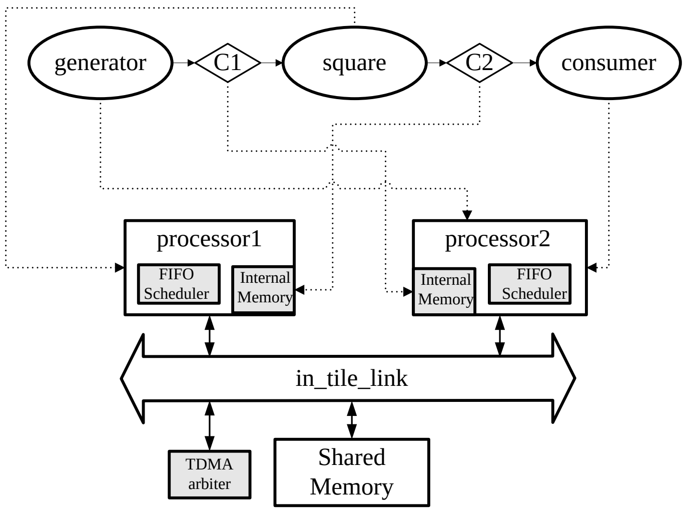


Figure 5.9: The mapping specification in Example 4 where the communication buffer is located in the origin processor.

Resource	Scheduling	Configuration
processor1	FIFO	square
processor2	FIFO	generator, consumer
in_tile_link	TDMA	quantum_slot_length: 30ns, processor1←1, processor2←2

Table 5.5: Scheduling configuration for Example 4.

The output waveform for Example 4 is depicted in Figure 5.10. The important phenomena of the waveform are indicated in the figure and the corresponding analysis is shown as follows:

```
Performance statistics:
--------------------------------------------------
process: generator
processing time on processor (NS): 1.200000e+03
processing time on buses (NS): 0.000000e+00
--------------------------------------------------
process: consumer
processing time on processor (NS): 1.220000e+03
processing time on buses (NS): 6.000000e+02
--------------------------------------------------
process: square
processing time on processor (NS): 2.020000e+03
processing time on buses (NS): 6.000000e+02
--------------------------------------------------
processor: processor1
processing time (NS): 2.020000e+03
--------------------------------------------------
processor: processor2
processing time (NS): 2.420000e+03
--------------------------------------------------
bus: in_tile_link
processing time (NS): 1.200000e+03
--------------------------------------------------
Estimated execution time (NS): 2.570000e+03
--------------------------------------------------
Max processing time of processors (NS): 2.420000e+03
--------------------------------------------------
Max processing time of buses (NS): 1.200000e+03
--------------------------------------------------
```

Listing 5.4: Performance statistics for Example 4.

1. Since the buffer of the software channel *C1* is on the origin processor *processor2*, the process *generator* does not need to take the bus *in_tile_link* in order to write to the buffer. So in the waveform, *generator* never uses the bus *in_tile_link* to write.
2. Since the buffer of the software channel *C2* is on the origin processor *processor1*, the process *square* does not need to take the bus *in_tile_link* to write to the buffer.

The output performance statistics table for Example 4 is depicted in Listing 5.4. Since the process *generator* never uses the bus to write to the software channel *C1*, the load of *generator* on the bus is 0.

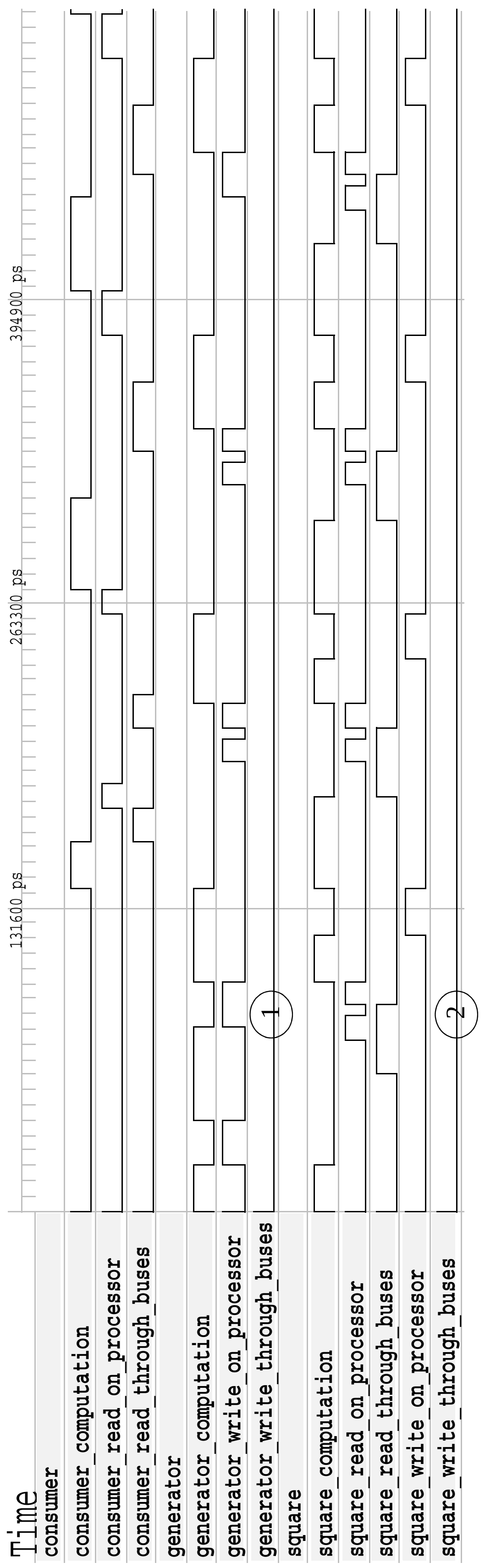


Figure 5.10: VCD waveform output for Example 4.

Target Processor

Example 5. *In this example, we investigate the scenario when the buffer is located in the target processor. The mapping is depicted in Figure 5.11. The scheduling configuration of the mapping is presented in Table 5.6.*

The performance statistics is listed in Listing 5.5. The output waveform is shown in Figure 5.12. Since the buffer of the software channel *C2* is implemented on the target processor *processor2*, the process *consumer* does not need to take the bus in order to read from *C2*. Consequently, the load of *consumer* on the bus is 0 and in the waveform there is no variable for the reading of *consumer* on the bus.

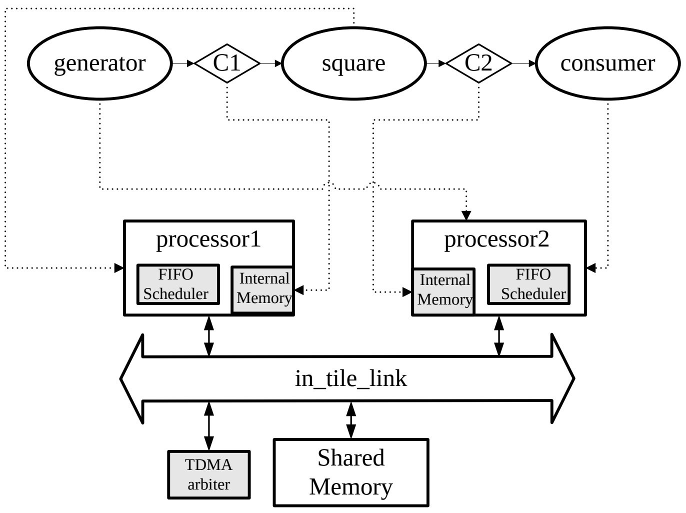


Figure 5.11: The mapping specification in Example 5 where the communication buffer is located in the target processor.

Resource	Scheduling	Configuration
processor1	FIFO	square
processor2	FIFO	generator, consumer
in_tile_link	TDMA	quantum_slot_length: 30ns, processor1←1, processor2←2

Table 5.6: Scheduling configuration for Example 5.

The output waveform for Example 5 is depicted in Figure 5.12. The important phenomena of the waveform are indicated in the figure and the corresponding analysis is shown as follows:

```
Performance statistics:
------------------------------------------------
process: generator
processing time on processor (NS): 1.200000e+03
processing time on buses (NS): 6.000000e+02
------------------------------------------------
process: consumer
processing time on processor (NS): 1.220000e+03
processing time on buses (NS): 0.000000e+00
------------------------------------------------
process: square
processing time on processor (NS): 2.020000e+03
processing time on buses (NS): 6.000000e+02
------------------------------------------------
processor: processor1
processing time (NS): 2.020000e+03
------------------------------------------------
processor: processor2
processing time (NS): 2.420000e+03
------------------------------------------------
bus: in_tile_link
processing time (NS): 1.200000e+03
------------------------------------------------
Estimated execution time (NS): 2.480000e+03
------------------------------------------------
Max processing time of processors (NS): 2.420000e+03
------------------------------------------------
Max processing time of buses (NS): 1.200000e+03
------------------------------------------------
```

Listing 5.5: Performance statistics for Example 5.

1. Since the buffer of the software channel *C2* is on the target processor *processor2*, the process *consumer* does not need to take the bus *in_tile_link* to read from the buffer.
2. Since the buffer of the software channel *C1* is on the target processor *processor1*, the process *square* does not need to take the bus in order to read from the buffer. So in the waveform, *square* never uses the bus to read.

The output performance statistics table for Example 5 is depicted in Listing 5.5. Since the process *consumer* never uses the bus to read from the software channel *C2*, the load of *consumer* on the bus is 0.

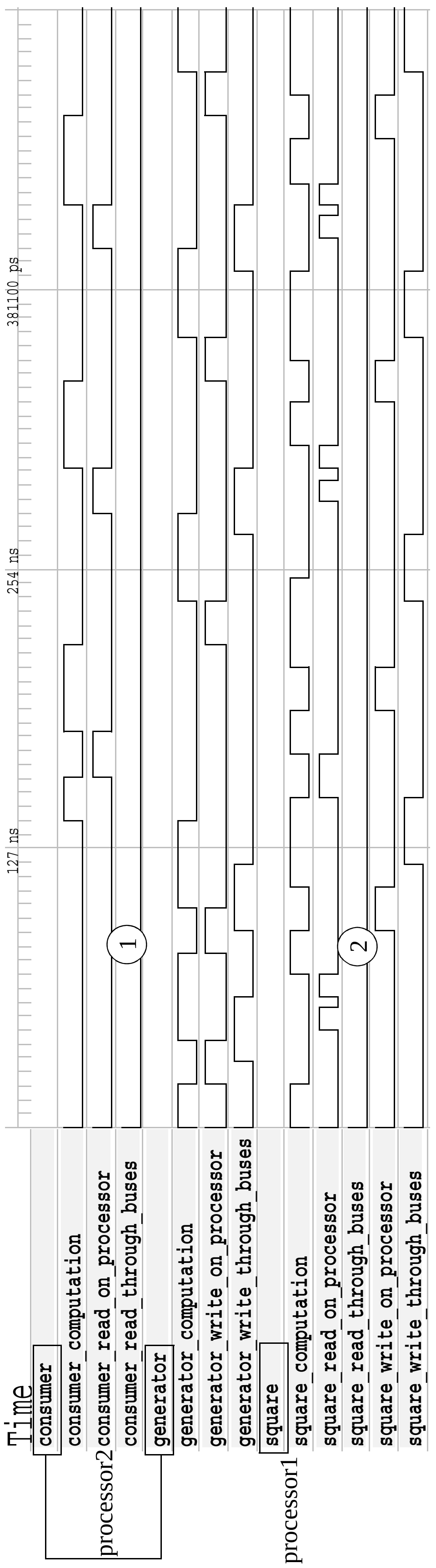


Figure 5.12: VCD waveform output for Example 5.

Shared Memory

Example 6. *In this mapping, we investigate the scenario when the buffer is located in the shared memory. The mapping is depicted in Figure 5.13. The scheduling configuration of the mapping is presented in Table 5.7.*

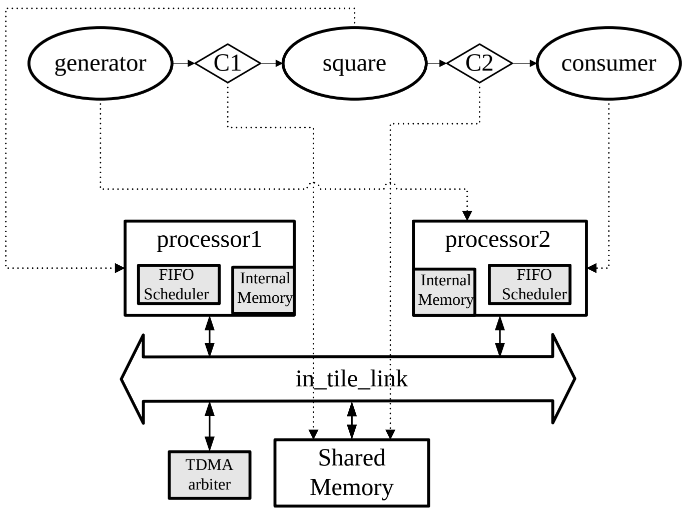


Figure 5.13: The mapping specification in Example 6 where the communication buffer is located in the shared memory.

Resource	Scheduling	Configuration
processor1	FIFO	square
processor2	FIFO	generator, consumer
in_tile_link	TDMA	quantum_slot_length: 30ns, processor1←1, processor2←2

Table 5.7: Scheduling configuration for Example 6.

The output waveform for Example 6 is depicted in Figure 5.14. The important phenomena of the waveform are indicated in the figure and the corresponding analysis is shown as follows:

1. Both writing to the software channel *C1* and writing to *C2* have to take the bus *in_tile_link* in order to access the shared memory.
2. Both reading from the software channel *C1* and reading from *C2* have to take the bus *in_tile_link* in order to access the shared memory.

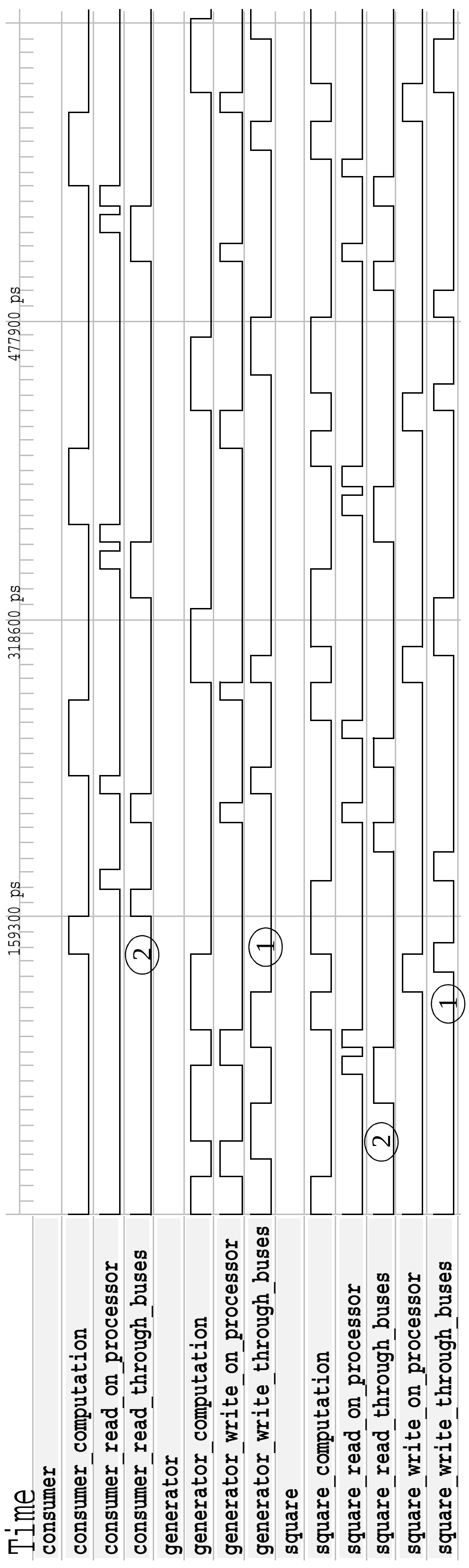


Figure 5.14: VCD waveform output for Example 6.

```
Performance statistics:
--------------------------------------------------------
process: generator
processing time on processor (NS): 1.200000e+03
processing time on buses (NS): 6.000000e+02
--------------------------------------------------------
process: consumer
processing time on processor (NS): 1.220000e+03
processing time on buses (NS): 6.000000e+02
--------------------------------------------------------
process: square
processing time on processor (NS): 2.020000e+03
processing time on buses (NS): 1.200000e+03
--------------------------------------------------------
processor: processor1
processing time (NS): 2.020000e+03
--------------------------------------------------------
processor: processor2
processing time (NS): 2.420000e+03
--------------------------------------------------------
bus: in_tile_link
processing time (NS): 2.400000e+03
--------------------------------------------------------
Estimated execution time (NS): 3.110000e+03
--------------------------------------------------------
Max processing time of processors (NS): 2.420000e+03
--------------------------------------------------------
Max processing time of buses (NS): 2.400000e+03
--------------------------------------------------------
```

Listing 5.6: Performance statistics for Example 6.

The output performance statistics table for Example 6 is depicted in Listing 5.6. By comparing with the previous two examples, we can draw the following conclusions:

- Both the load of *generator* on the bus and the load of *consumer* on the bus are not 0. The reason is that *generator* have to access the bus in order to write to the shared memory. The process *consumer* has to access the bus in order to read from the shared memory.
- The total execution time is 3110 ns, which is larger than the previous two examples. This is reasonable because when the buffer is implemented in the shared memory, it is more costly to access it than when it is implemented in the local memory of the processor. Both read and write will have to take the bus in order to access the shared memory.

5.1.5 Communication Path

Example 7. *To prove that our framework is able to handle communication path consisting of a series of resources in the architecture, we mapped the producer-consumer application to a more complex architecture. The mapping is depicted in Figure 5.15. The configuration of the mapping is similar to the one in Example 2, except that in this example an additional bus* in_tile_link2 *is added.*

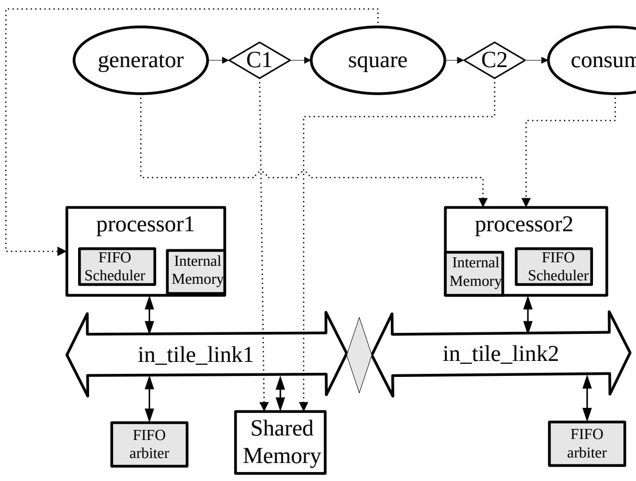


Figure 5.15: The mapping specification in Example 7 where the communication path consists of two buses.

Resource	Scheduling	Configuration
processor1	FIFO	square
processor2	FIFO	generator, consumer
in_tile_link1	FIFO	processor1, in_tile_link2
in_tile_link2	FIFO	processor2, in_tile_link1

Table 5.8: Scheduling configuration for Example 7.

The output waveform is shown in Figure 5.16. As can be seen in the figure, the time consumed by writing of *generator* on the buses is larger than that in Example 2. The reason is that *generator* will have to take two buses successively in order to access the buffer of *C1*.

The performance statistics are listed in Listing 5.7. As is shown in Figure 5.15, the buffer of *C1* and the buffer of *C2* are both implemented in the *shared memory* linked

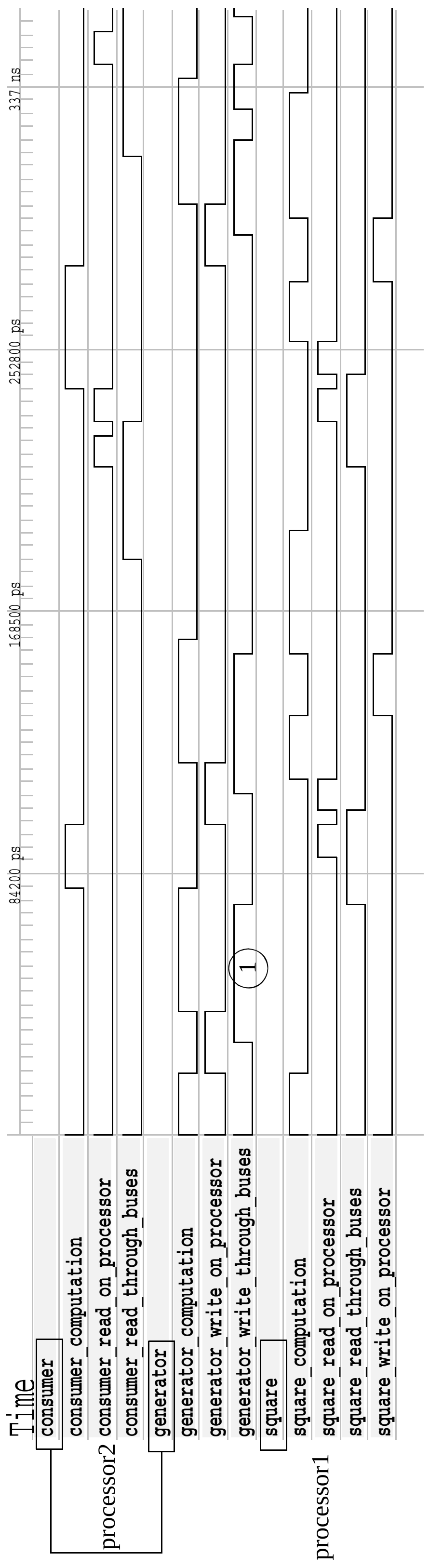


Figure 5.16: VCD waveform output for Example 7.

```
Performance statistics:
------------------------------------------------
process: generator
processing time on processor (NS): 1200
communication time through buses (NS): 1200
------------------------------------------------
process: consumer
processing time on processor (NS): 1220
communication time through buses (NS): 1200
------------------------------------------------
process: square
processing time on processor (NS): 2020
communication time through buses (NS): 1200
------------------------------------------------
processor: processor1
processing time (NS): 2020
------------------------------------------------
processor: processor2
processing time (NS): 2420
------------------------------------------------
bus: in_tile_link1
processing time (NS): 2400
------------------------------------------------
bus: in_tile_link2
processing time (NS): 1200
------------------------------------------------
Estimated execution time (NS): 3290
------------------------------------------------
Max processing time of processors (NS): 2420
------------------------------------------------
Max processing time of buses (NS): 2400
------------------------------------------------
```

Listing 5.7: Performance statistics for Example 7.

to *in_tile_link1*. As a result, *generator* needs to access both *in_tile_link1* and *in_tile_link2* in order to write to the buffer of *C1*. This is why in Example 2 *generator* can write through the communication path in 600 ns, while in this example it is 1200 ns. Besides, the estimated execution time for this mapping is 3290 ns, much larger than the estimated execution time in Example 2, namely 2540 ns. The reason is the extra bus will create additional delays in the communication path.

5.1.6 Resource Binding

Example 8. *Different from all the experiments above, we map* generator *and* square *to the same processor* processor1*, while* consumer *monopolies* processor2*. The mapping is depicted in Figure 5.17.*

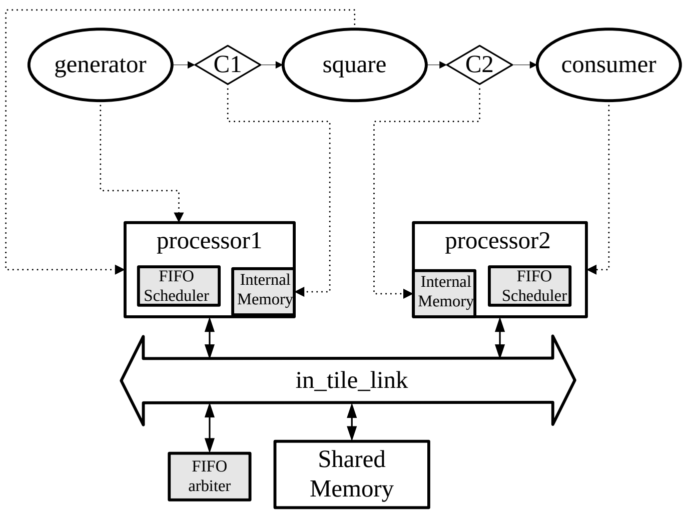


Figure 5.17: The mapping specification in Example 8 with a different resource binding.

Resource	Scheduling	Configuration
processor1	FIFO	generator, square
processor2	FIFO	consumer
in_tile_link	FIFO	processor1, processor2

Table 5.9: Scheduling configuration for Example 8.

The output waveform is shown in Figure 5.18.

1. Different form the previous examples, in this example, *generator* and *consumer* are mapped to different processors. Therefore, they can run concurrently.
2. The process *generator* and *square* are both mapped to the same processor *processor1*. Therefore, their executions are serialized.

The performance statistics are listed in Listing 5.8. The load of *processor1* in this mapping is 3220 ns and the load of *processor2* is 1220 ns. In Example 2, which has similar scheduling configuration, the load of *processor1* is 2020 ns and the load of *processor2* is

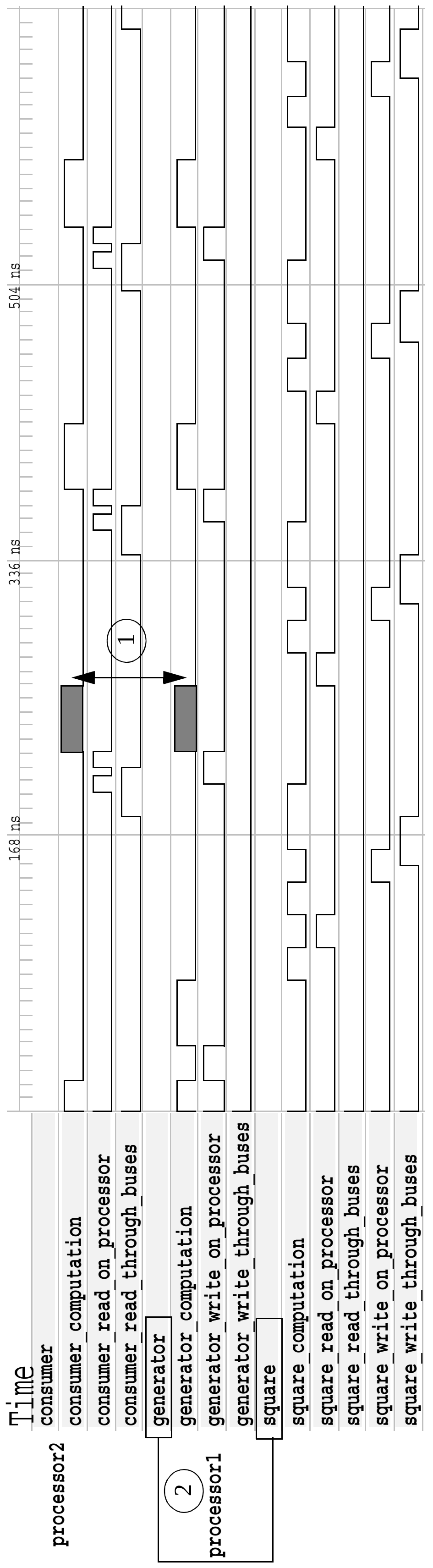


Figure 5.18: VCD waveform output for Example 8.

```
Performance statistics:
------------------------------------------------
process: generator
processing time on processor (NS): 1.200000e+03
processing time on buses (NS): 0.000000e+00
------------------------------------------------
process: consumer
processing time on processor (NS): 1.220000e+03
processing time on buses (NS): 6.000000e+02
------------------------------------------------
process: square
processing time on processor (NS): 2.020000e+03
processing time on buses (NS): 6.000000e+02
------------------------------------------------
processor: processor1
processing time (NS): 3.220000e+03
------------------------------------------------
processor: processor2
processing time (NS): 1.220000e+03
------------------------------------------------
bus: in_tile_link
processing time (NS): 1.200000e+03
------------------------------------------------
Estimated execution time (NS): 3.280000e+03
------------------------------------------------
Max processing time of processors (NS): 3.220000e+03
------------------------------------------------
Max processing time of buses (NS): 1.200000e+03
------------------------------------------------
```

Listing 5.8: Performance statistics for Example 8.

2420 ns. The difference results from the fact that in this mapping, two processes, i.e. *generator* and *square*, are both mapped onto *processor1* while only *consumer* is mapped onto processor2. By contrast, in Example 2, only *square* is mapped onto *processor1* while both *generator* and *consumer* are mapped onto *processor2*. Besides, the total execution time is 3250 ns, which is larger than 2440 ns in Example 2. The reason is that in this example the loads of the resources are not well balanced.

5.2 The MPEG-2 Case Study

In this case study, we use a more complex application, namely MPEG-2 decoder to test our framework for handling mapping of complex system and to prove its efficiency. The results of the trace-based simulation are analyzed and compared with pure functional simulation.

5.2.1 The MPEG-2 Application and Architecture Description

The MPEG-2 decoder used in this experiment is written using the application specification schema of DOL, where it has been implemented as a process network and uses blocking read and blocking write semantics. Two kinds of architectures are used as the implementation platforms in the experiments: the single tile architecture and the two tile architecture.

The Application

The MPEG-2 decoder is implemented in a reconfigurable manner where the granularity of parallelism is scalable. The parameters N_1, N_2 and N_3 described in Table 5.10 are the levels of parallel processed entities. By changing the parameters N_1, N_2 and N_3 in the process network XML file, different variants of the implementation can be obtained. If we set $N_1 = 1$, $N_2 = 1$ and $N_3 = 1$, the pipelined version is achieved, as shown in Figure 5.19. If we set $N_1 = 2$, $N_2 = 2$ and $N_3 = 2$, a parallel version is conducted, as shown in Figure 5.20.

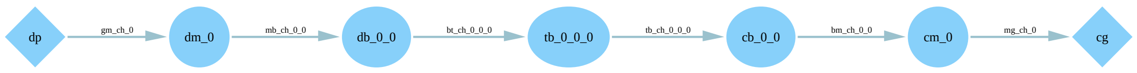


Figure 5.19: MPEG-2 process network with the configuration ($N_1 = 1$, $N_2 = 1$, $N_3 = 1$).

N_1	groups of pictures are processed in parallel
N_2	macroblocks are processed in parallel
N_3	blocks are processed in parallel

Table 5.10: The decompositions parameters in MPEG-2.

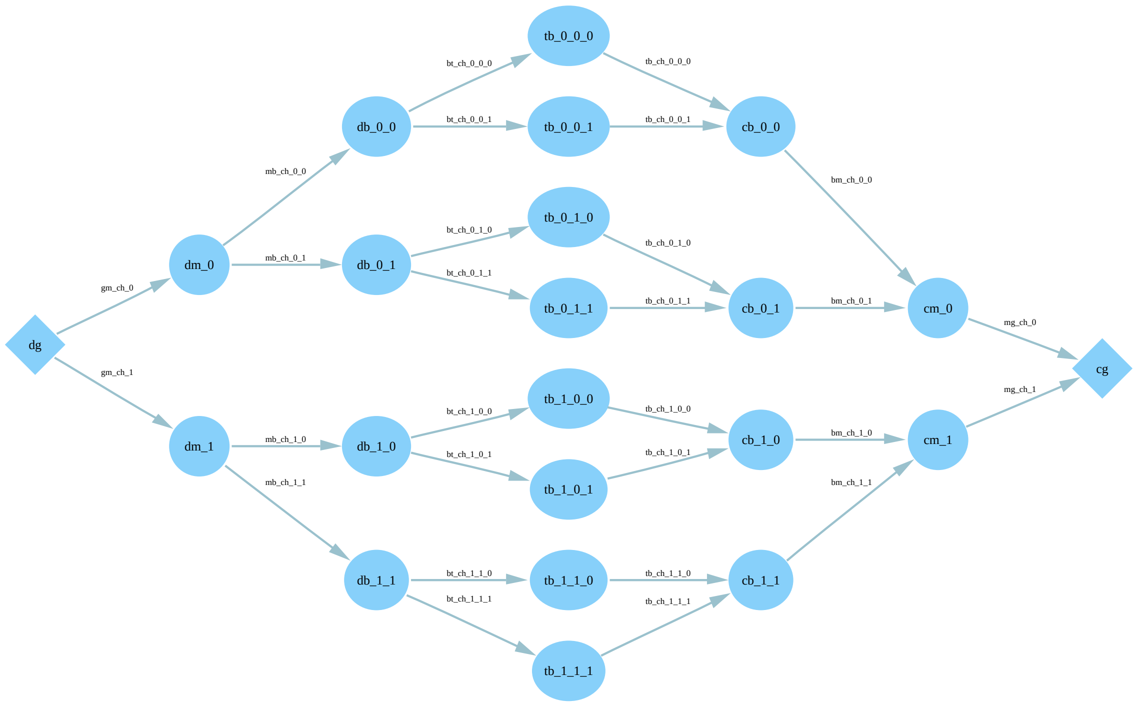


Figure 5.20: MPEG-2 process network with the configuration ($N_1 = 2$, $N_2 = 2$, $N_3 = 2$).

The Architecture

In our experiments, two different architectures are used as implementation platforms for the MPEG-2 decoder. They are described as follows:

- The first architecture shown in Figure 5.21 is a simple one, with two processors, i.e. *processor1* and *processor2*, and a bus *in_tile_link* connecting them. The configuration for the resources in the architecture is illustrated in Table 5.11. We have already denoted this configuration in the beginning of this section as a tile.
- The second architecture shown in Figure 5.22 is more complex compared with the first one. It has four computation resources, namely *DSP_0*, *DSP_1*, *RISC_0* and *RISC_1*. Besides, it has three communication resources, i.e. *intra_tile_bus_0*, *intra_tile_bus_1* and *inter_tile_bus*. The bus *inter_tile_bus* is used to connect the tiles.

5.2.2 Experimental Results

In order to demonstrate the efficiency and other characteristics of our trace-based simulation framework, we conduct several experiments on the MPEG-2 application, including

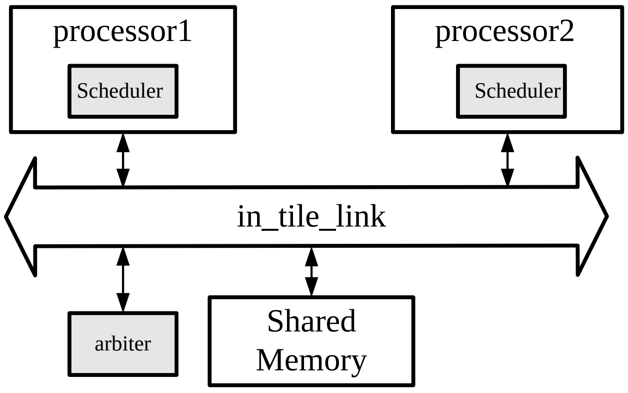


Figure 5.21: Architecture with two processors and one bus.

Resource	Configuration
Processor1	dsp, 200MHz
Processor2	risc, 200MHz
in_tile_link	32bit, 200MHz

Table 5.11: Configuration of the resources in Figure 5.21.

several mapping solutions, and that are compared with the pure functional simulation of this application. The input test files for all the MPEG-2 process networks in the following experiments are the same in order to make the results comparable.

Functional Simulation

First we conduct the functional simulation for the MPEG-2 process network depicted in Figure 5.19. The extra functionalities that we have added into the functional simulation, e.g. trace generation, are removed in order to obtain the actual performance. Without instrumentation perturbation, the functional simulation takes about 167 seconds for the MPEG-2 application to process the input file, i.e. a `.m2v` video file of about 2.7 MB. The overhead lies in both running the computation code and executing data transfer through software channels.

Trace Generation

As is discussed before, the deterministic character of the process network enables the traces be generated only once and used for different mappings. Thus the time needed for running

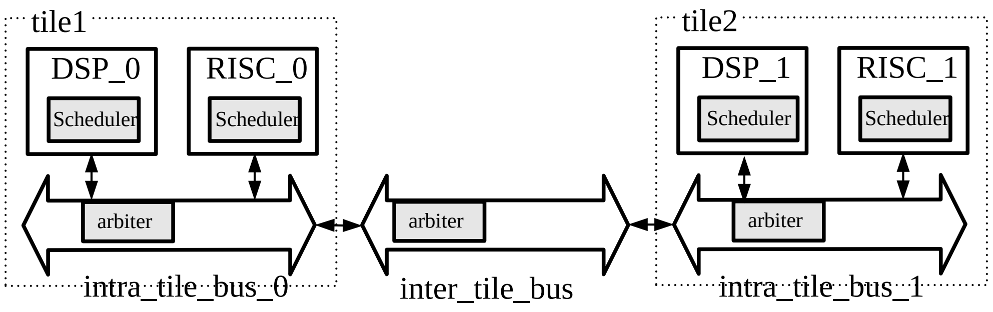


Figure 5.22: Architecture with four processors and three buses.

Resource	Configuration
RISC_0	risc, 200MHz
RISC_1	risc, 200MHz
DSP_0	dsp, 200MHz
DSP_0	dsp, 200MHz
intra_tile_bus_0	32bit, 200MHz
intra_tile_bus_1	32bit, 200MHz
inter_tile_bus	32bit, 200MHz

Table 5.12: Configuration of the resources in Figure 5.22.

the actual computation code is saved and the efficiency of the trace-based simulation can be increased. This will be demonstrated later.

To generate the traces used for the trace-based simulation, again we run the functional model of the MPEG-2 application in DOL. This time we include the trace generation instrumentation in the functional simulation. Consequently, the functional simulation time is larger for the reason that extra overhead is needed for dynamically generating and storing the traces. It takes 263 seconds to finish the simulation with respect to 167 seconds without instrumentation. The traces stored in the trace file are used to trigger the trace-based simulation which will re-execute these traces for different mappings and implementation platforms.

The size of the trace generated is about 500 MB. When the traces get larger the simulation time will increase as well. Besides, since the trace-based simulation reads into the whole traces once into the memory, the input traces can not exceed 700 MB for the

MPEG-2 decoder.

Mapping the pipelined version to the one tile architecture

The architecture of a single tile used in this mapping is depicted in Figure 5.21. The traces of the application are generated from the functional simulation of the MPEG-2 configuration depicted in Figure 5.19. The configuration of the mapping is illustrated in Table 5.13. Particularly, the scheduling policy adopted by all the resources is FIFO. The performance statistics of the mapping are generated as an output of the trace-based simulation, and they are shown in Listing 5.9.

Process or Software Channel.	Binding Configuration
dispatch_gops	processor2
collect_gops	processor1
dispatch_mb_0	processor2
collect_mb_0	processor1
dispatch_block_0_0	processor1
collect_block_0_0	processor1
transform_block_0_0_0	processor1
mb_channel_0_0	processor2→in_tile_link→shared memory→ in_tile_link→processor1, buffer:processor1
gm_channel_0	processor2
mg_channel_0	processor1
bm_channel_0_0	processor1
bt_channel_0_0_0	processor1
tb_channel_0_0_0	processor1

Table 5.13: Configuration of Mapping MPEG-2 in Figure 5.19 to architecture in Figure 5.21.

The efficiency of the trace-based simulation is high. There are two reasons:

- It takes about 33 seconds to finish the trace-based simulation while it takes about 167 seconds to finish the functional simulation. The speed-up is more than 5 times.
- It is more accurate because the trace-based simulation has already taken the implementation architecture into account.

The reason is that in our trace-based simulation framework, it does need to run the actual computation code and the traces can be reused in the design space exploration.

```
Performance statistics:
---------------------------------------------
process: dispatch_gops
processing time on processor (NS): 6.059948e+08
processing time on buses (NS): 3.530310e+06
---------------------------------------------
process: collect_gops
processing time on processor (NS): 1.637740e+10
---------------------------------------------
process: dispatch_mb_0
processing time on processor (NS): 4.560947e+09
---------------------------------------------
process: collect_mb_0
processing time on processor (NS): 7.060805e+09
---------------------------------------------
process: dispatch_blocks_0_0
processing time on processor (NS): 2.500740e+09
---------------------------------------------
process: collect_blocks_0_0
processing time on processor (NS): 2.280960e+09
---------------------------------------------
process: transform_block_0_0_0
processing time on processor (NS): 4.000590e+09
---------------------------------------------
processor: processor1
processing time (NS): 3.678144e+10
---------------------------------------------
processor: processor2
processing time (NS): 6.059948e+08
---------------------------------------------
bus: in_tile_link
processing time (NS): 3.530310e+06
---------------------------------------------
Estimated execution time (NS): 3.678614e+10
---------------------------------------------
Max processing time of processors (NS): 3.678144e+10
---------------------------------------------
Max processing time of buses (NS): 3.530310e+06
---------------------------------------------
```

Listing 5.9: Performance statistics for mapping the pipelined version to the one tile architecture.

Mapping the pipelined version to the two tile architecture

The two tile architecture used in this mapping is shown in Figure 5.22. The application is the same as in the previous experiment. The configuration of this mapping is described in Table 5.14. Besides, the scheduling policy for this mapping is FIFO. The performance statistics generated from the trace-based simulation are shown in Listing 5.10.

Process or Software Channel	Binding Configuration
dispatch_gops	RISC_0
collect_gops	DSP_1
dispatch_mb_0	DSP_0
collect_mb_0	RISC_1
dispatch_block_0_0	DSP_1
collect_block_0_0	RISC_1
transform_block_0_0_0	RISC_1
mb_channel_0_0	DSP_0→intra_tile_bus_0→inter_tile_bus →intra_tile_bus_1→DSP_1, buffer:DSP_1
gm_channel_0	RISC_0→intra_tile_bus_0→DSP_0, buffer:DSP_0
mg_channel_0	RISC_1→intra_tile_bus_1→DSP_1, buffer:DSP_1
bm_channel_0_0	RISC_1
bt_channel_0_0_0	DSP_1→intra_tile_bus_1 →RISC_1, buffer: RISC_1
tb_channel_0_0_0	RISC_1

Table 5.14: Configuration of Mapping MPEG-2 in Figure 5.19 to architecture in Figure 5.22.

The time to finish the simulation for this mapping is about 41 seconds, which is larger compared with the previous mapping, i.e. 33 seconds. The major difference between these two experiments is that the communication architecture in this mapping is more complex. Therefore, it is likely that a communication trace event needs to be processed by more resources in this experiment, thus increasing the simulation time. Nevertheless, as the architecture becomes more complex, the speed does not decrease exponentially.

Mapping the fully parallel version to the one tile architecture

The single tile architecture used in this mapping is shown in Figure 5.21. The traces of the application are generated from the functional simulation on the MPEG-2 in Figure 5.20. In the configuration of the mapping, all the processes are mapped onto *processor1* except

```
Performance statistics:
---------------------------------------------
process: dispatch_gops
processing time on processor (NS): 6.059948e+08
processing time on buses (NS): 3.530310e+06
---------------------------------------------
process: collect_gops
processing time on processor (NS): 1.637740e+10
---------------------------------------------

...

processor: RISC_0
processing time (NS): 6.059948e+08
---------------------------------------------
processor: RISC_1
processing time (NS): 1.334236e+10
---------------------------------------------
processor: DSP_0
processing time (NS): 4.560947e+09
---------------------------------------------
processor: DSP_1
processing time (NS): 1.887814e+10
---------------------------------------------
bus: intra_tile_bus_0
processing time (NS): 3.346853e+08
---------------------------------------------
bus: intra_tile_bus_1
processing time (NS): 9.683438e+08
---------------------------------------------
bus: inter_tile_bus
processing time (NS): 3.311550e+08
---------------------------------------------
Estimated execution time (NS): 2.496425e+10
---------------------------------------------
Max processing time of processors (NS): 1.887814e+10
---------------------------------------------
Max processing time of buses (NS): 9.683438e+08
---------------------------------------------
```

Listing 5.10: Performance statistics for mapping the pipelined version to the two tile architecture.

dispatch_gops, which is mapped onto *processor2*. By changing the scheduling policy of *processor1*, the following performance results are retrieved from the trace-based simulation.

- FIFO scheduling: The performance statistics are shown in Listing 5.11. The time to finish the simulation is about 34 seconds, which is similar to the first mapping of the MPEG-2 in Figure 5.19 to the single tile architecture in Figure 5.21. The architectures for these two mappings are the same, while the process network for this last mapping is more complex. Yet it seems that the complexity of the application does not lead to the sacrifice of the simulation speed. The reason is that both process networks have the same input and the size of the generated traces are similar. Hence the number of traces events that need to be processed by the architecture is similar. In a word, the speed of the simulation depends on the trace size and the complexity of the architecture.
- TDMA scheduling: The performance statistics are shown in Listing 5.12. The time to finish the simulation is about 43 seconds. The simulation speed is slower compared to FIFO scheduling policy for the reason that there are more transitions to model in the TDMA scheduler. In this mapping, the length of the time slot is set to 1 ms. In particular, if the time slot length is too small, the simulation speed will degrade significantly. The reason is that the TDMA scheduler will need much more transitions in order to model the switches because of the completion of time slots. Yet in the actual situation, the time slot has a convenient size, e.g. 3 ms.
- FP scheduling: The performance statistics are shown in Listing 5.13. The time to finish the simulation is about 39 seconds.

In particular, the time used in the initialization phase of the trace-based simulation is 57 seconds, which includes the process of reading traces from the disk. Yet after that, the traces in the memory can be reused for each mapping. So the time for initialization can be saved for the following mappings.

```
Performance statistics:
--------------------------------------------
process: dispatch_gops
processing time on processor (NS): 5.752707e+08
processing time on buses (NS): 3.530325e+06
--------------------------------------------
process: collect_gops
processing time on processor (NS): 1.667483e+10
processing time on buses (NS): 7.140376e+07
--------------------------------------------
process: dispatch_mb_0
processing time on processor (NS): 4.475900e+09
--------------------------------------------
process: collect_mb_0
processing time on processor (NS): 7.099947e+09
processing time on buses (NS): 7.140376e+07
--------------------------------------------


...

processor: processor1
processing time (NS): 2.089579e+10
--------------------------------------------
processor: processor2
processing time (NS): 1.725010e+10
--------------------------------------------
bus: in_tile_link
processing time (NS): 1.463382e+08
--------------------------------------------
Estimated execution time (NS): 2.101954e+10
--------------------------------------------
Max processing time of processors (NS): 2.089579e+10
--------------------------------------------
Max processing time of buses (NS): 1.463382e+08
--------------------------------------------
```

Listing 5.11: Performance statistics for mapping the fully parallel version to the one tile architecture using FIFO.

```
Performance statistics:
---------------------------------------------
process: dispatch_gops
processing time on processor (NS): 5.752707e+08
processing time on buses (NS): 3.530325e+06
---------------------------------------------
process: collect_gops
processing time on processor (NS): 1.667483e+10
processing time on buses (NS): 7.140376e+07
---------------------------------------------
process: dispatch_mb_0
processing time on processor (NS): 4.475900e+09
processing time on buses (NS): 0.000000e+00
---------------------------------------------
process: collect_mb_0
processing time on processor (NS): 7.099947e+09
processing time on buses (NS): 7.140376e+07
---------------------------------------------

...

processor: processor1
processing time (NS): 2.089579e+10
---------------------------------------------
processor: processor2
processing time (NS): 1.725010e+10
---------------------------------------------
bus: in_tile_link
processing time (NS): 1.463382e+08
---------------------------------------------
Estimated execution time (NS): 1.426309e+11
---------------------------------------------
Max processing time of processors (NS): 2.089579e+10
---------------------------------------------
Max processing time of buses (NS): 1.463382e+08
---------------------------------------------
```

Listing 5.12: Performance statistics for mapping the fully parallel version to the one tile architecture using TDMA.

```
Performance statistics:
--------------------------------------------------
process: dispatch_gops
processing time on processor (NS): 5.752707e+08
processing time on buses (NS): 3.530325e+06
--------------------------------------------------
process: collect_gops
processing time on processor (NS): 1.667483e+10
processing time on buses (NS): 7.140376e+07
--------------------------------------------------
process: dispatch_mb_0
processing time on processor (NS): 4.475900e+09
processing time on buses (NS): 0.000000e+00
--------------------------------------------------
process: collect_mb_0
processing time on processor (NS): 7.099947e+09
processing time on buses (NS): 7.140376e+07
--------------------------------------------------

...

processor: processor1
processing time (NS): 2.089579e+10
--------------------------------------------------
processor: processor2
processing time (NS): 1.725010e+10
--------------------------------------------------
bus: in_tile_link
processing time (NS): 1.463382e+08
--------------------------------------------------
Estimated execution time (NS): 2.101520e+10
--------------------------------------------------
Max processing time of processors (NS): 2.089579e+10
--------------------------------------------------
Max processing time of buses (NS): 1.463382e+08
--------------------------------------------------
```

Listing 5.13: Performance statistics for mapping the fully parallel version to the one tile architecture using FP.

5.3 Summary

We have investigated two case studies in this chapter. The producer-consumer case study proves that our framework is able to handle various scheduling policies, different locations of the software buffer and communications via a series of resources. In the second case study, we investigate mappings of a more complex application, i.e. the MPEG-2 decoder, onto complex multiprocessor system. The positive results testify the scalability and modularity of our framework. Besides, the speed of the trace-based simulation is faster than the functional simulation for the MPEG-2 case study, which proves the efficiency of our framework. The exploration results are summarized as follows for both the producer-consumer application and the MPEG-2 decoder.

The Producer-consumer Application

The results of exploring different communication buffer locations are shown in Table 5.15. We keep other parameters the same and only change the communication buffer location. When we mapped the communication buffer onto the shared memory, the total execution time is larger than that when it is put into the internal memory of the processor. The reason is that both read and write will need to compete and access the bus for shared memory. Thus extra overhead is required.

Communication buffer	Execution time (ns)
Origin processor	2510
Target processor	2480
Shared memory	3110

Table 5.15: Exploration of communication buffer locations.

The results of exploring communication path are shown in Table 5.16. When we increase the number of buses in the communication path from one to two, the total execution time will increase. The reason is that the extra bus will introduce additional delays on the communication path.

Number of buses between two processors	Execution time (ns)
One	2540
Two	3290

Table 5.16: Exploration of communication path.

The results of exploring different resource bindings are shown in Table 5.17. In the first binding, the loads of the resources are better balanced. Consequently, the total execution time of the first binding, i.e. 2440 ns, is smaller compared with that of the second binding, i.e. 3250.

Binding	Processor1 load (ns)	Processor2	Execution
	load (ns)	load (ns)	time (ns)
generator, consumer→processor2; square→processor1	2020	2420	2440
generator, square→processor1; consumer→processor2	3220	1220	3280

Table 5.17: Exploration of resource bindings.

The results of exploring different scheduling policies are shown in Table 5.18. The total execution time of TDMA is largest. The reason is that in TDMA, each time slot is fixed to a certain process. If the process is not ready to run, the time slot will be wasted. Therefore, the resources are not fully utilized.

Scheduling	Execution time (ns)
FIFO	2540
FP	2560
TDMA	3815

Table 5.18: Exploration of scheduling policies.

The MPEG-2 Decoder

The results of exploring different implementation platforms for the pipelined version is shown in Table 5.19. When it is mapped to a two-tile architecture rather than on one tile architecture, the total execution time will reduce by one third. The reason is that the two tile architecture has more computation ability and can explore more concurrency between the processes.

The time needed for functional simulation is 167 seconds. When the pipelined version is mapped onto the one tile architecture, the time needed for trace-based simulation is 33 seconds. The speed-up is about 5 times in this scenario.

The results of exploring different scheduling policies for the fully parallel version are shown in Table 5.20. It turns out that the total execution time of TDMA is the largest. The reason is that some time slots are wasted and the resources are not fully utilized.

Application	Architecture	Scheduling	Execution time (s)	Speed of functional simulation (s)	Speed of trace-based simulation (s)	Speed-up
Pipelined	One tile	FIFO	36.8	167	33	5.1
Pipelined	Two tile	FIFO	25	167	41	4.1

Table 5.19: Exploration of implementation platforms.

Application	Architecture	Scheduling	Execution time (s)	Speed of functional simulation (s)	Speed of trace-based simulation (s)	Speed-up
Parallel	One tile	FIFO	21	169	34	5
Parallel	One tile	FP	21	169	39	4.3
Parallel	One tile	TDMA	142.6	169	43	3.9

Table 5.20: Exploration of scheduling policies.

6 Conclusion and Future Work

This chapter gives a conclusion for the whole thesis, as well as the future work. In Section 6.1, the content of this thesis and the major characteristics of the framework are summarized. Section 6.2 gives an overview of the future work that can help to improve the current framework.

6.1 Conclusion

In this thesis we have presented a trace-based simulation framework which can guide designers at early stages of the design space exploration. We described the basic components and the underlying mechanisms of the framework, such as the application modeling, architecture modeling and mapping modeling. In the application modeling, an application is first specified using a restricted version of the Kahn Process Network. The speed-up of the simulation is achieved by abstracting the functionality of the application as high-level traces. In the architecture modeling, we used a uniform resource definition to represent both processors and buses. The architecture is modular and can be extended to have arbitrary resources. In the mapping modeling, the application and the architecture are associated together to form a system. Besides, important factors, e.g. sharing resources and communication architectures, that can exert influence on the system performance, are investigated in the framework. Subsequently, how the framework is implemented in the context of DOL is given. The method of expressing application, architecture and mapping in the XML files is presented. After that, we used two case studies, i.e. the producer-consumer application and the MPEG-2 decoder, to verify a variety of aspects of the framework and to prove the efficiency of our framework. The major characteristics of our trace-based simulation framework are summarized as follows:

- Modular: Both the application and the architecture are specified in a modular way. An application written within the process network specified scheme can have arbitrary number of processes and software channels. Besides, the architecture in the framework can be composed of arbitrary processors and buses.
- Automated: The process of trace generation is automated deriving from the modified functional simulation. In the trace-based simulation framework, the application and architecture are constructed automatically from the files. In addition, the procedure of parsing the mapping configuration is automatic.
- Efficient: The efficiency of the simulation is guaranteed by abstracting computation behaviors as traces. Thus in the procedure of simulation, there is no need to execute the actual computation programming code, but only to consider the timing effects. The traces are generated once and can be reused for different mappings for the same application.
- Capable to explore a variety of parameters: Many factors that can affect the system performance are considered. The framework is able to explore arbitrary communication structures, e.g. internal communication, external communication, communication via a path. The framework is capable of investigating the various scenarios of implementing the software channel buffer on origin processor, target processor or shared memory. In addition, important scheduling policies, including TDMA, FIFO and Fixed Priority, have been implemented to provide exclusively mutual access to the resources.

6.2 Future Work

In future work, we plan to:

- Couple our framework with a low-level simulation to increase the accuracy of the performance evaluation results. Since execution delays of the computation trace events are used to model the timing effect of the computation behaviors, the precision of the delays can profoundly affect the final simulation results. A low-level simulation, e.g. an instruction set simulator (ISS), can be used to achieve this goal. For each type of processor, its corresponding ISS can be utilized to obtain the performance estimation for the computation behavior. Normally, the speed on the ISS is very slow, with 3 orders of magnitude slower than the functional simulation. Yet because the performance estimation for the basic computation is retrieved once and for all, the overall efficiency of the trace-based simulation framework will not be affected.

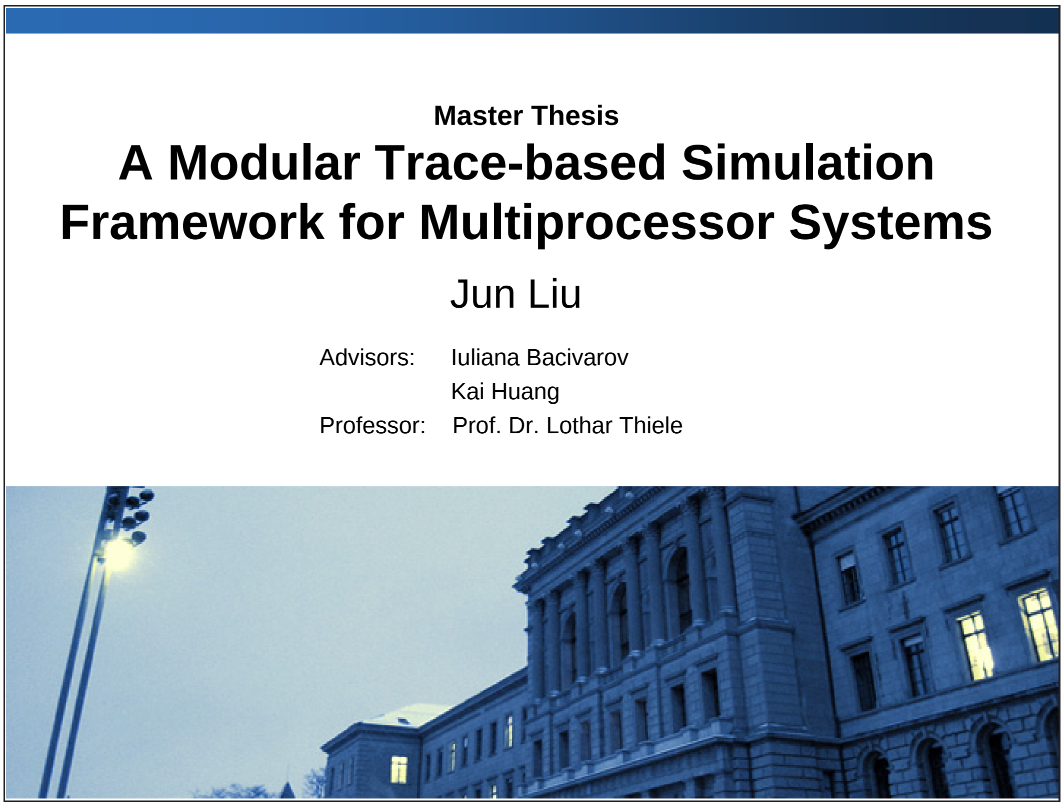
- Consider the size of the hardware communication transfer buffer in the process of simulation. Currently we assume that the transfer buffer between two hardware resources is unbounded. Thus no process will be blocked because of the overflow of the hardware buffer. In the actual simulation, however, the size of a hardware communication buffer is limited, which will block the processes when the buffer is full. This will cause indirect effect on the overall performance of the system. If we can take into account of this effect, the accuracy of the framework can be enhanced.

- Model other performance aspects besides timing, e.g. power consumption and buffer fill level. The current framework models only the timing effect of the system. In the process of designing embedded system, another significant performance indicator is the power consumption. We can take this into consideration by adding more information in the trace event. For example, the computation trace event now contains only the execution delays required to finish the computation on a target processor. Similarly, we can include the energy estimation in the computation event so that the energy consumption effect for a target processor can be simulated. In addition, the buffer fill level can help designers to determine the size of software buffer to implement. The buffer fill level can be retrieved by tracking the state change of the virtual software channel. Finally, even for timing aspect, we will include the ending time of each process, which reflects the response speed when the process is mapped to certain processor.

- Provide a graphic interface to facilitate the usage for the designers. There are several input files for the simulation tool, one of which is the mapping XML file. Each design solution needs a mapping file to specify the configuration. The process of writing a mapping file is tedious and error-prone, especially when the application and architecture get larger and more complex. Therefore, graphic interface can be used to solve this problem. The mapping XML file will be automatically derived from the graphic mapping specification.

- Model more types of resources in the architecture. Computation resources in the current framework are restricted to processors while the communication resources are restricted to buses. To increase the accuracy, the external shared memory may also be modeled as an independent resource. Besides, a communication resource may also be a dedicated link between two processors, or a NoC (Network-on-Chip). The uniform resource definition in our framework reduces the efforts of modeling these new types of resources.
- Implement more scheduling policies for shared resources. The implemented scheduling policies are TDMA, FIFO and FP. Practically, there may be more types of scheduling mechanisms, such as round-robin or priority-based round-robin.
- Investigate the error introduced by the atomic data transaction size. By introducing the atomic data transaction size, the data transfer procedure in our simulation framework can be closer to the actual situation. Yet a larger atomic data transaction size will sacrifice the accuracy of the simulation. The side effect of the atomic data transaction size should be exploited.
- Integrate the current framework with the existing DOL high level multi-objective DSE framework [16]. When the design space is large, it is difficult or even infeasible to investigate all the solutions one by one manually. To overcome this problem, the multi-objective design exploration framework can be utilized to make the process of design optimization automatic. At the beginning, the multi-objective optimization explores the overall design space. After a set of optimized candidates are available, the designers can filter better choices by comparing the performance numbers by means of our trace-based simulation.

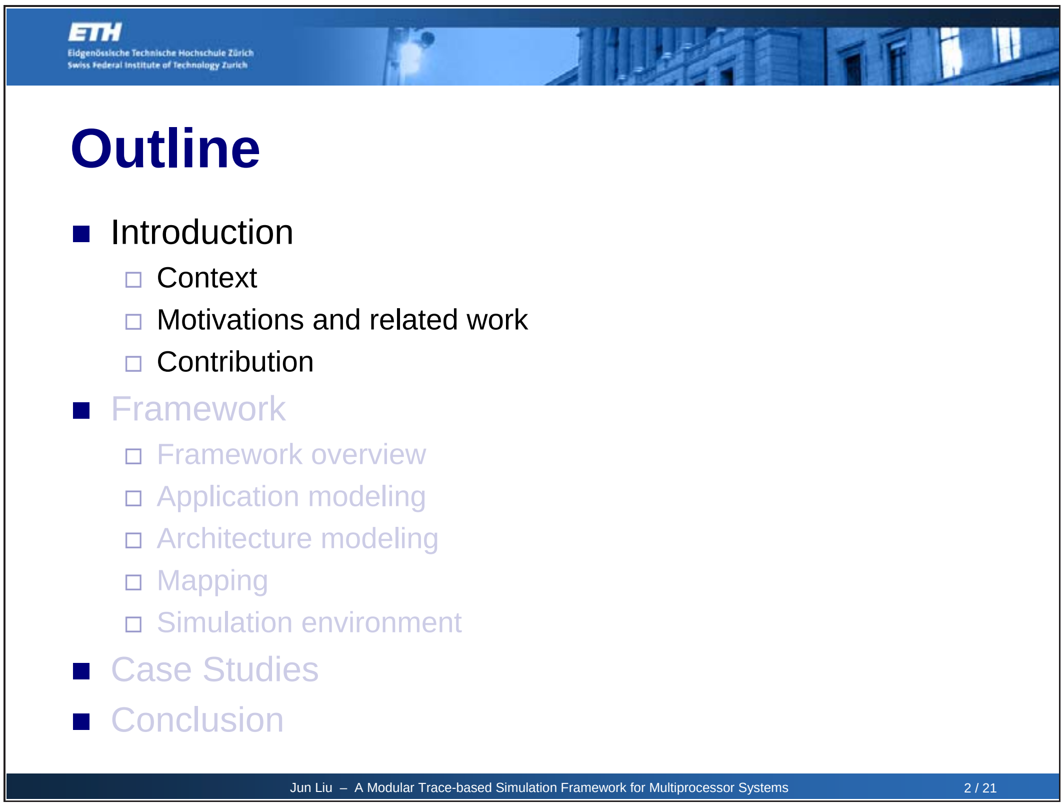
A
Presentation Slides


Master Thesis

A Modular Trace-based Simulation Framework for Multiprocessor Systems

Jun Liu

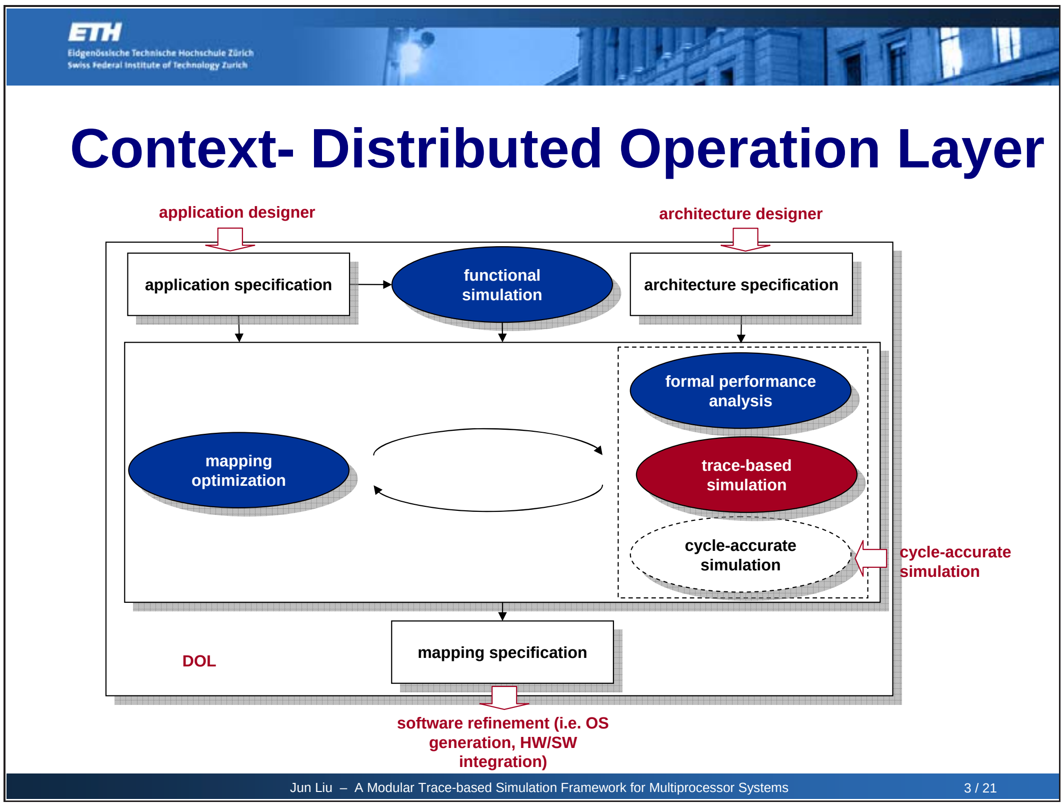
Advisors:	Iuliana Bacivarov
	Kai Huang
Professor:	Prof. Dr. Lothar Thiele



Outline

- Introduction
  - Context
  - Motivations and related work
  - Contribution
- Framework
  - Framework overview
  - Application modeling
  - Architecture modeling
  - Mapping
  - Simulation environment
- Case Studies
- Conclusion

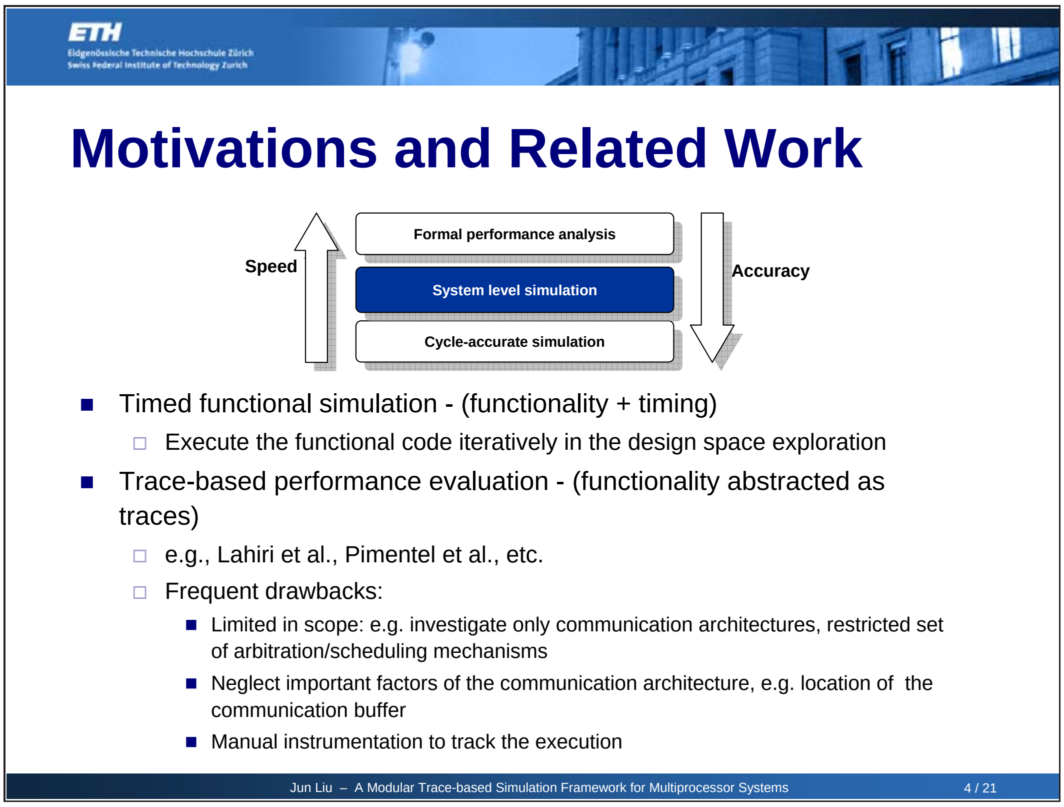
Jun Liu – A Modular Trace-based Simulation Framework for Multiprocessor Systems 2 / 21



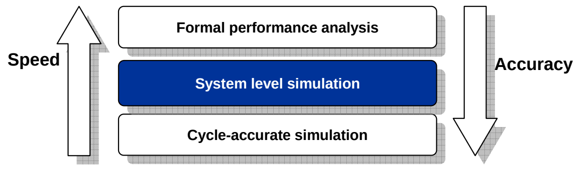
Context- Distributed Operation Layer

application designer
architecture designer
application specification
functional simulation
architecture specification
formal performance analysis
mapping optimization
trace-based simulation
cycle-accurate simulation
cycle-accurate simulation
DOL
mapping specification
software refinement (i.e. OS generation, HW/SW integration)

Jun Liu – A Modular Trace-based Simulation Framework for Multiprocessor Systems 3 / 21

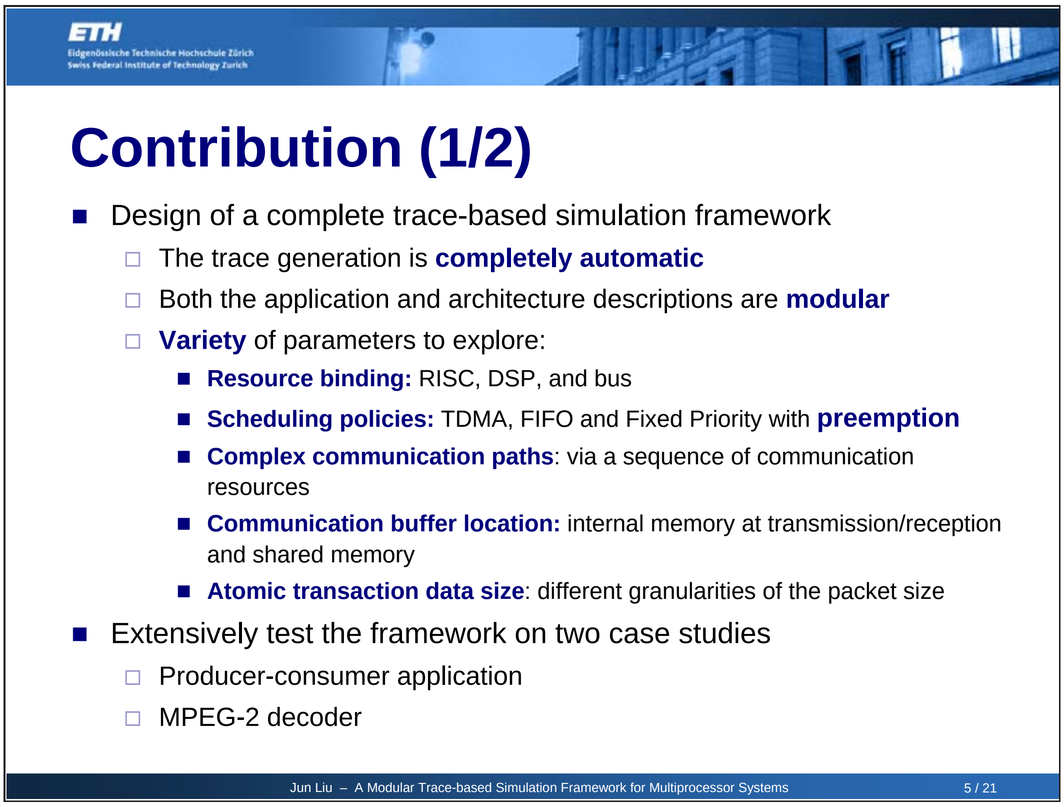


Motivations and Related Work



- Timed functional simulation - (functionality + timing)
  - Execute the functional code iteratively in the design space exploration
- Trace-based performance evaluation - (functionality abstracted as traces)
  - e.g., Lahiri et al., Pimentel et al., etc.
  - Frequent drawbacks:
    - Limited in scope: e.g. investigate only communication architectures, restricted set of arbitration/scheduling mechanisms
    - Neglect important factors of the communication architecture, e.g. location of the communication buffer
    - Manual instrumentation to track the execution

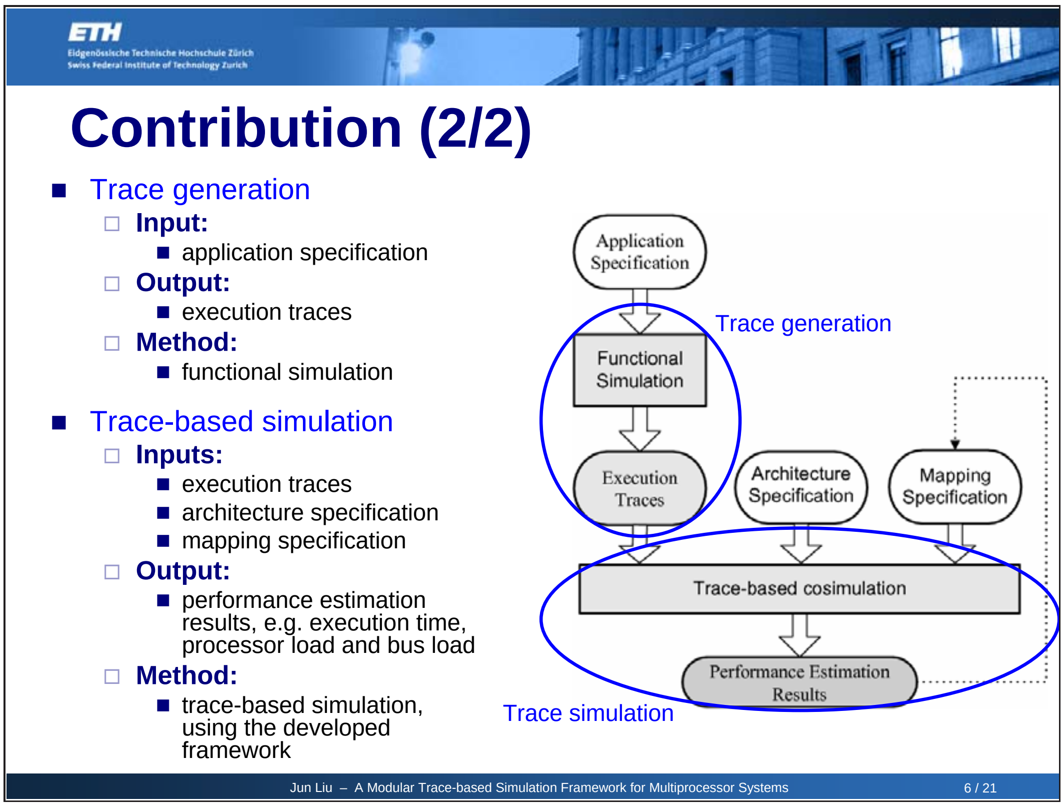
Jun Liu – A Modular Trace-based Simulation Framework for Multiprocessor Systems 4 / 21



Contribution (1/2)

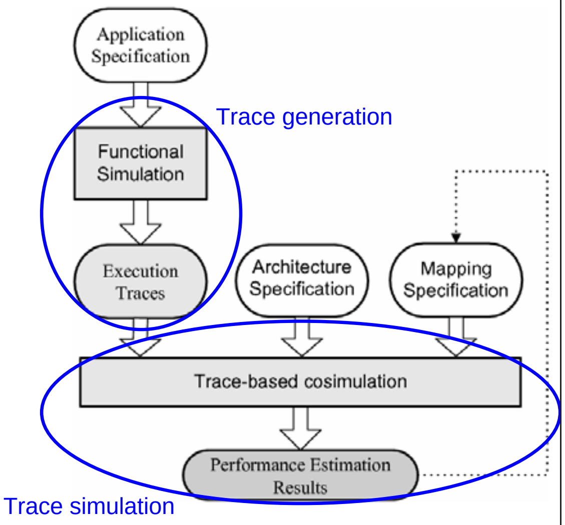
- Design of a complete trace-based simulation framework
  - The trace generation is **completely automatic**
  - Both the application and architecture descriptions are **modular**
  - **Variety** of parameters to explore:
    - **Resource binding:** RISC, DSP, and bus
    - **Scheduling policies:** TDMA, FIFO and Fixed Priority with **preemption**
    - **Complex communication paths**: via a sequence of communication resources
    - **Communication buffer location:** internal memory at transmission/reception and shared memory
    - **Atomic transaction data size**: different granularities of the packet size
- Extensively test the framework on two case studies
  - Producer-consumer application
  - MPEG-2 decoder

Jun Liu – A Modular Trace-based Simulation Framework for Multiprocessor Systems 5 / 21

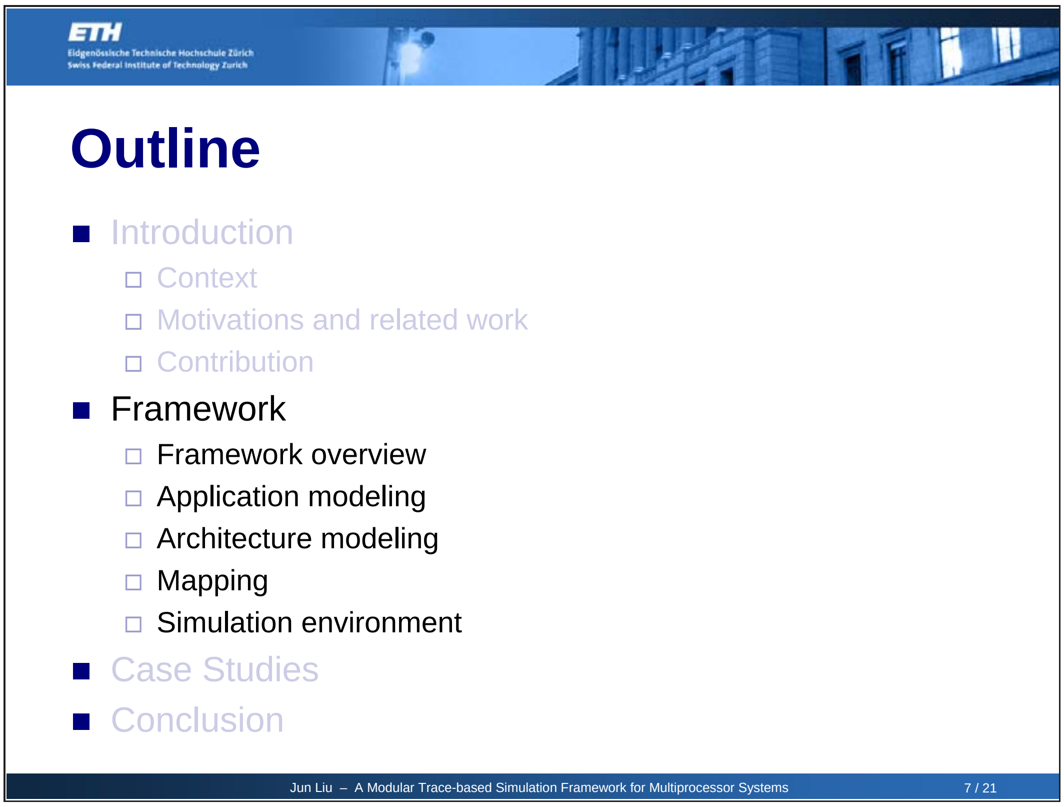


Contribution (2/2)

- Trace generation
  - Input:
    - application specification
  - Output:
    - execution traces
  - Method:
    - functional simulation
- Trace-based simulation
  - Inputs:
    - execution traces
    - architecture specification
    - mapping specification
  - Output:
    - performance estimation results, e.g. execution time, processor load and bus load
  - Method:
    - trace-based simulation, using the developed framework



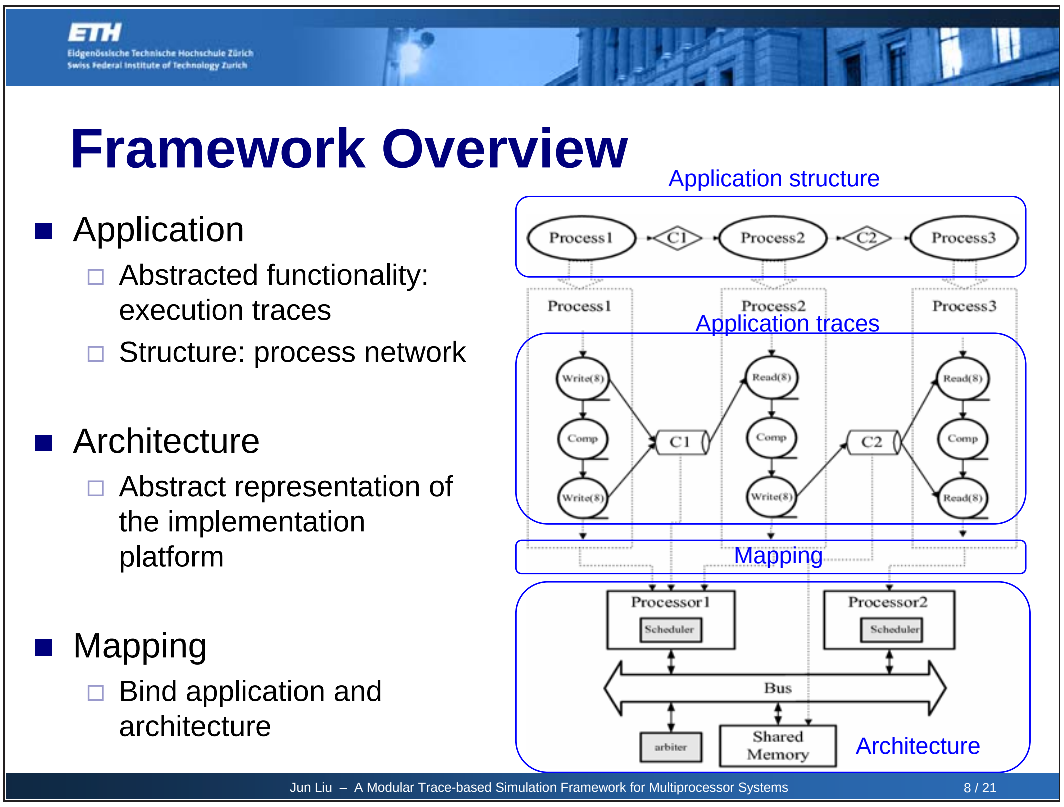
Jun Liu – A Modular Trace-based Simulation Framework for Multiprocessor Systems 6 / 21



Outline

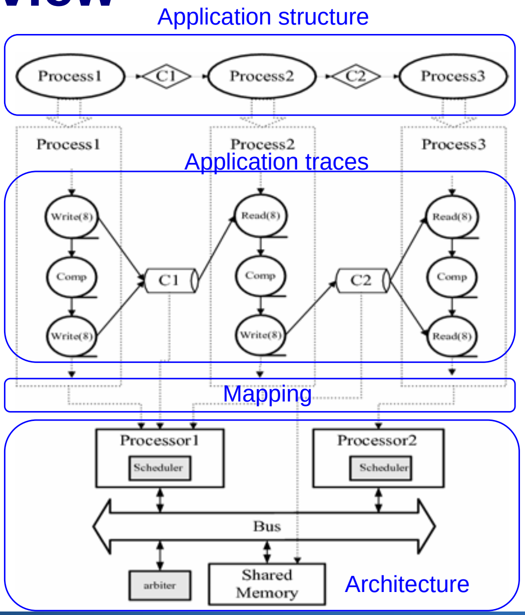
- Introduction
  - Context
  - Motivations and related work
  - Contribution
- Framework
  - Framework overview
  - Application modeling
  - Architecture modeling
  - Mapping
  - Simulation environment
- Case Studies
- Conclusion

Jun Liu – A Modular Trace-based Simulation Framework for Multiprocessor Systems 7 / 21

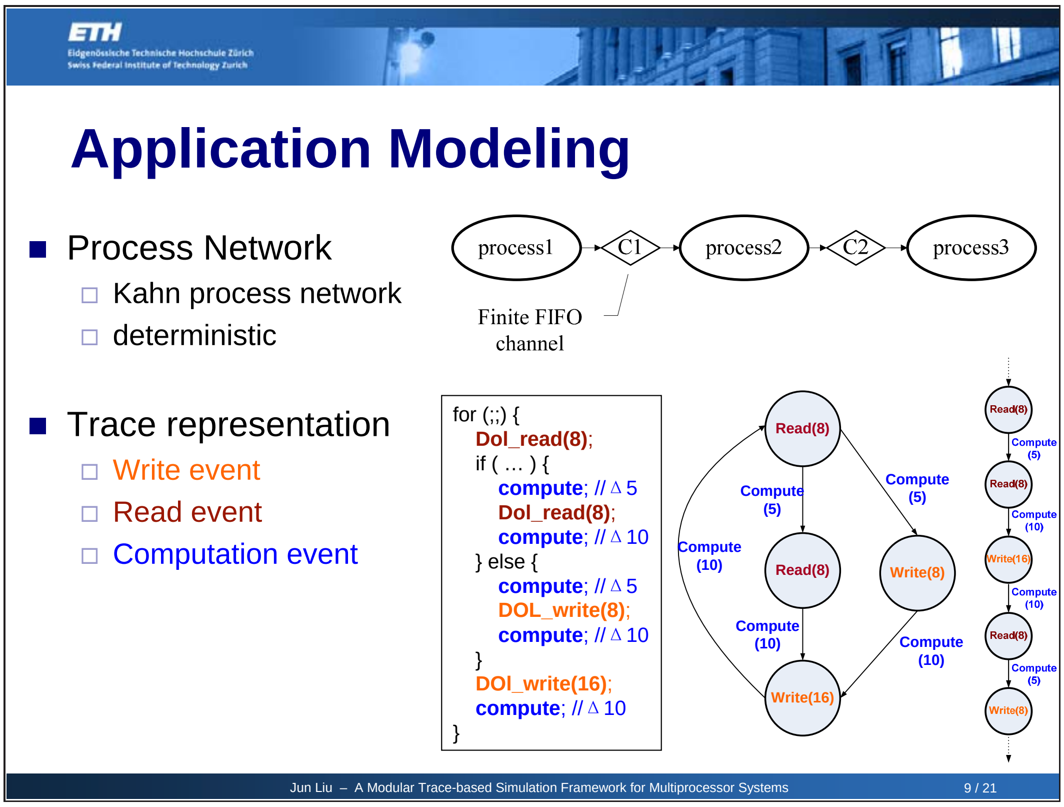


Framework Overview

- Application
  - Abstracted functionality: execution traces
  - Structure: process network
- Architecture
  - Abstract representation of the implementation platform
- Mapping
  - Bind application and architecture

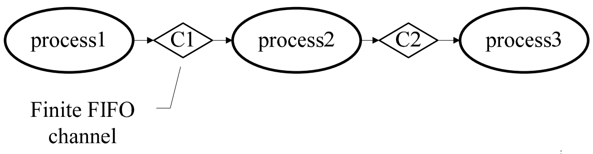


Jun Liu – A Modular Trace-based Simulation Framework for Multiprocessor Systems 8 / 21

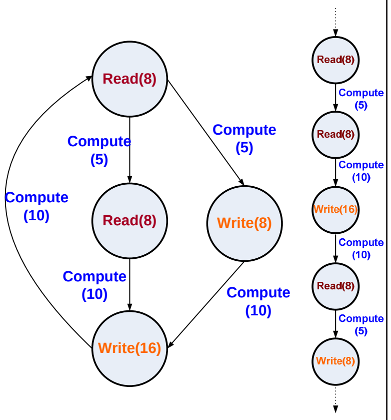


Application Modeling

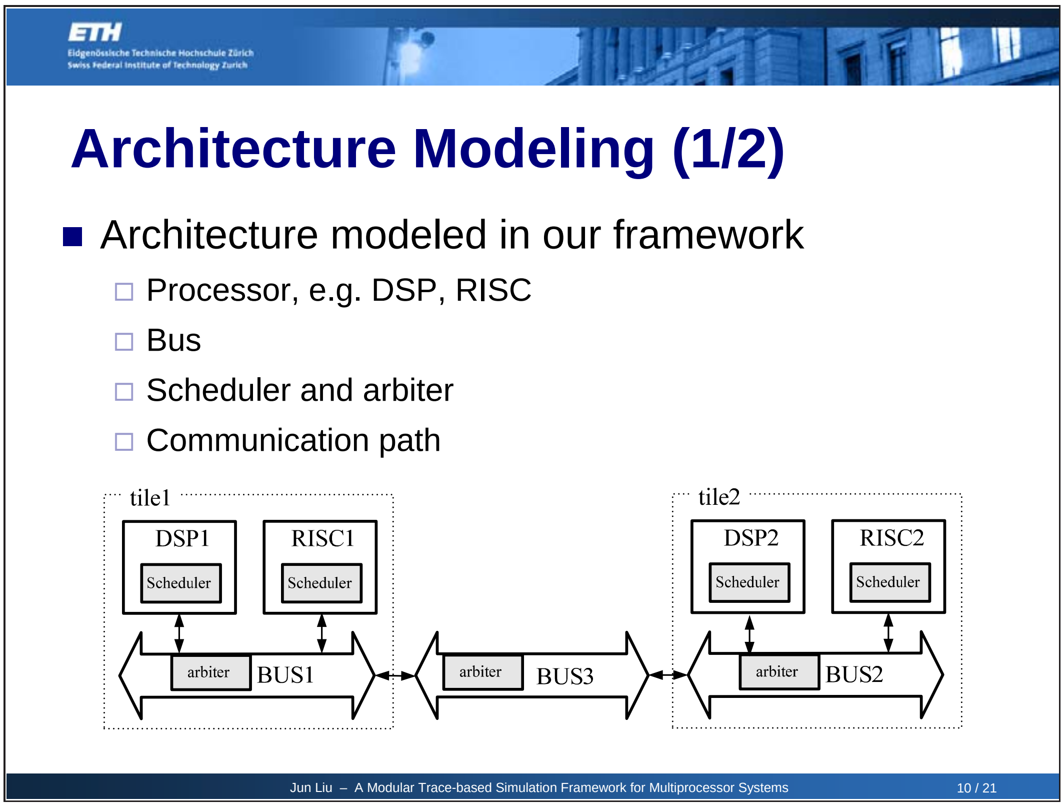
- Process Network
  - Kahn process network
  - deterministic
- Trace representation
  - Write event
  - Read event
  - Computation event



```
for (;;) {
   DoI_read(8);
   if ( … ) {
      compute; // Δ 5
      DoI_read(8);
      compute; // Δ 10
   } else {
      compute; // Δ 5
      DOL_write(8);
      compute; // Δ 10
   }
   DOI_write(16);
   compute; // Δ 10
}
```



Jun Liu – A Modular Trace-based Simulation Framework for Multiprocessor Systems 9 / 21

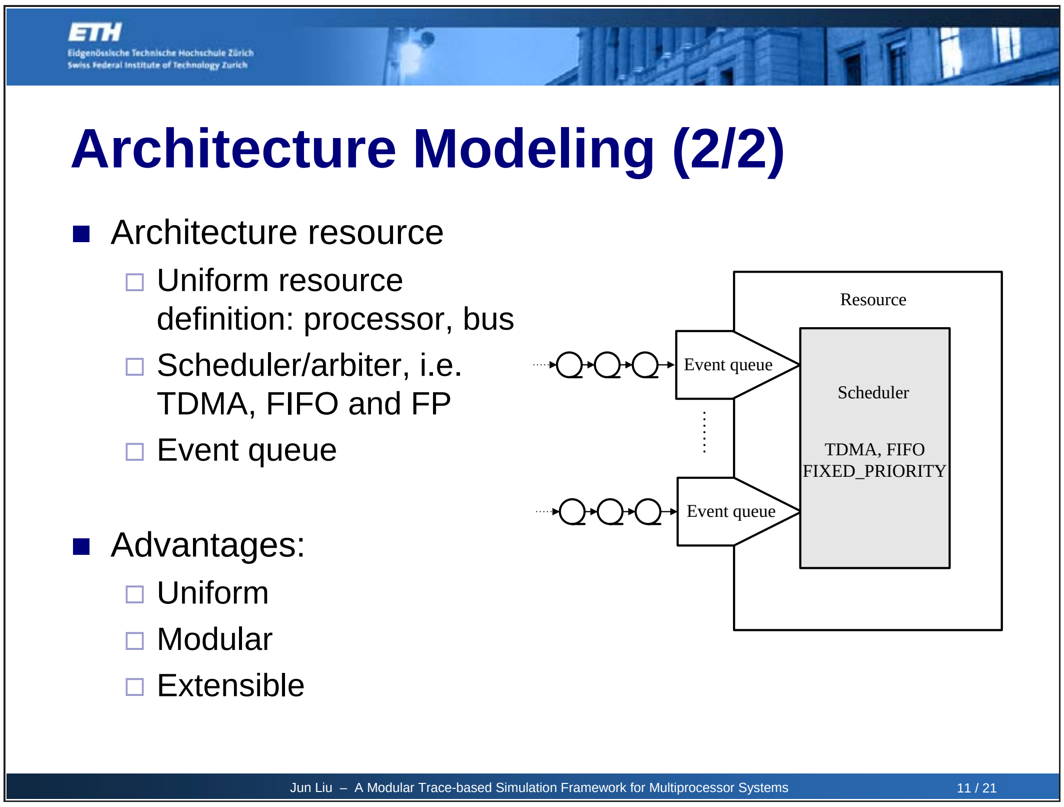


Architecture Modeling (1/2)

- Architecture modeled in our framework
  - Processor, e.g. DSP, RISC
  - Bus
  - Scheduler and arbiter
  - Communication path

tile1
DSP1
Scheduler
RISC1
Scheduler
arbiter
BUS1
arbiter
BUS3
tile2
DSP2
Scheduler
RISC2
Scheduler
arbiter
BUS2

Jun Liu – A Modular Trace-based Simulation Framework for Multiprocessor Systems 10 / 21



Architecture Modeling (2/2)

- Architecture resource
  - Uniform resource definition: processor, bus
  - Scheduler/arbiter, i.e. TDMA, FIFO and FP
  - Event queue
- Advantages:
  - Uniform
  - Modular
  - Extensible

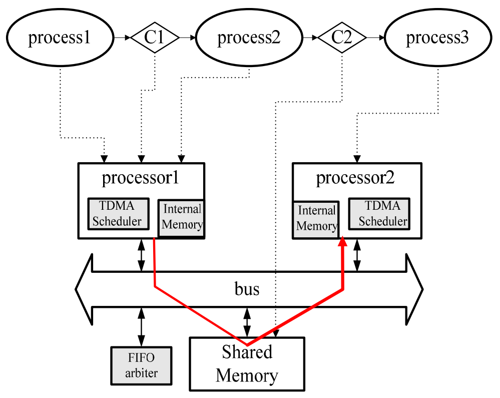
Resource
Event queue
Scheduler
TDMA, FIFO
FIXED_PRIORITY
Event queue

Jun Liu – A Modular Trace-based Simulation Framework for Multiprocessor Systems 11 / 21

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Mapping

- Binding
  - Process -> processor
  - Software channel ->communication path
  - Communication buffer -> memory (internal memory, shared memory)
- Scheduling
  - For shared resources, e.g. FIFO, TDMA and FP

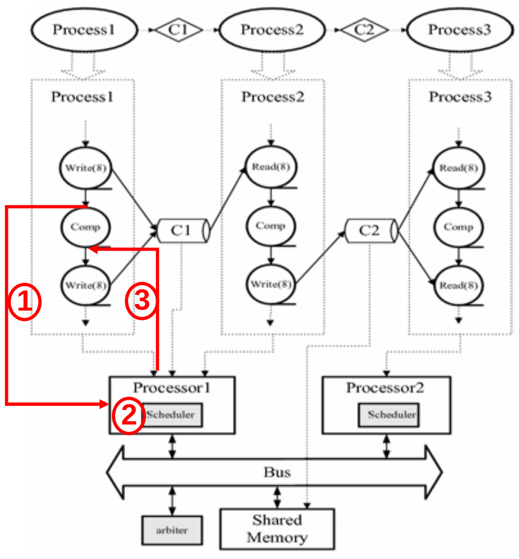


Jun Liu – A Modular Trace-based Simulation Framework for Multiprocessor Systems 12 / 21

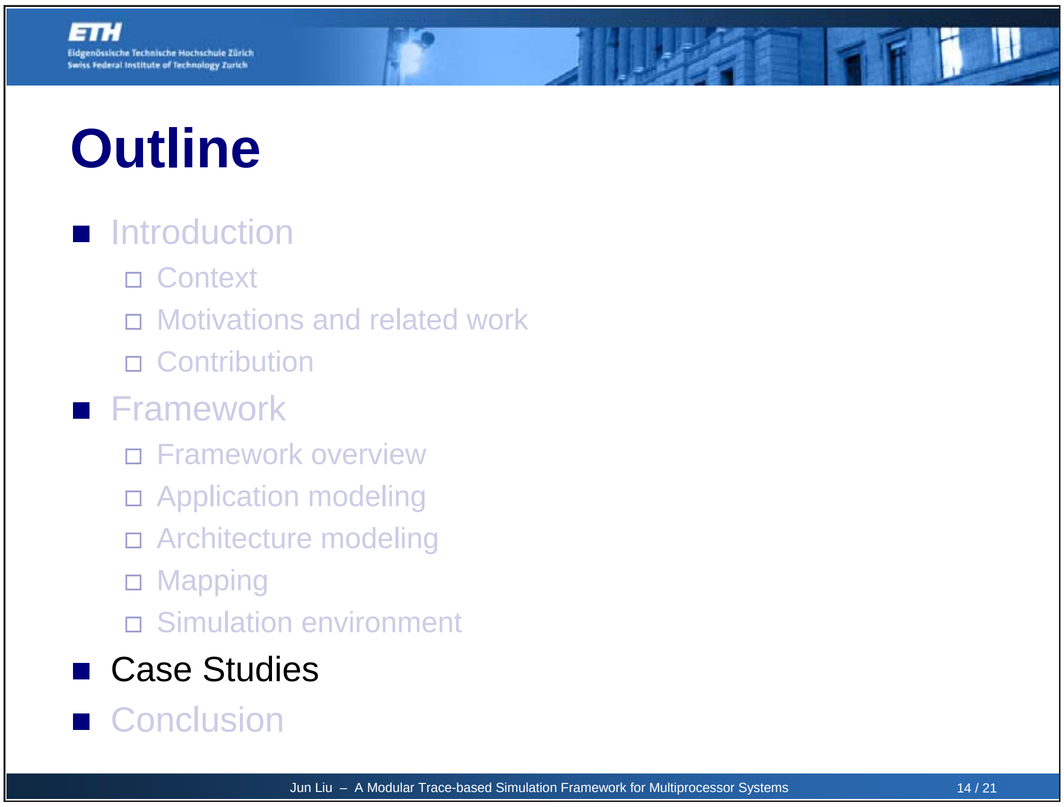
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Simulation Environment

- SystemC (www.systemc.org)
  - Event-trigger mechanism
  - wait() function to model execution delays
- Computation event simulation
  1. Deliver the computation event to the target processor
  2. The processor models the execution effect of the computation event
  3. Notify the process the completion of the trace event
- Communication event simulation
  - Similar



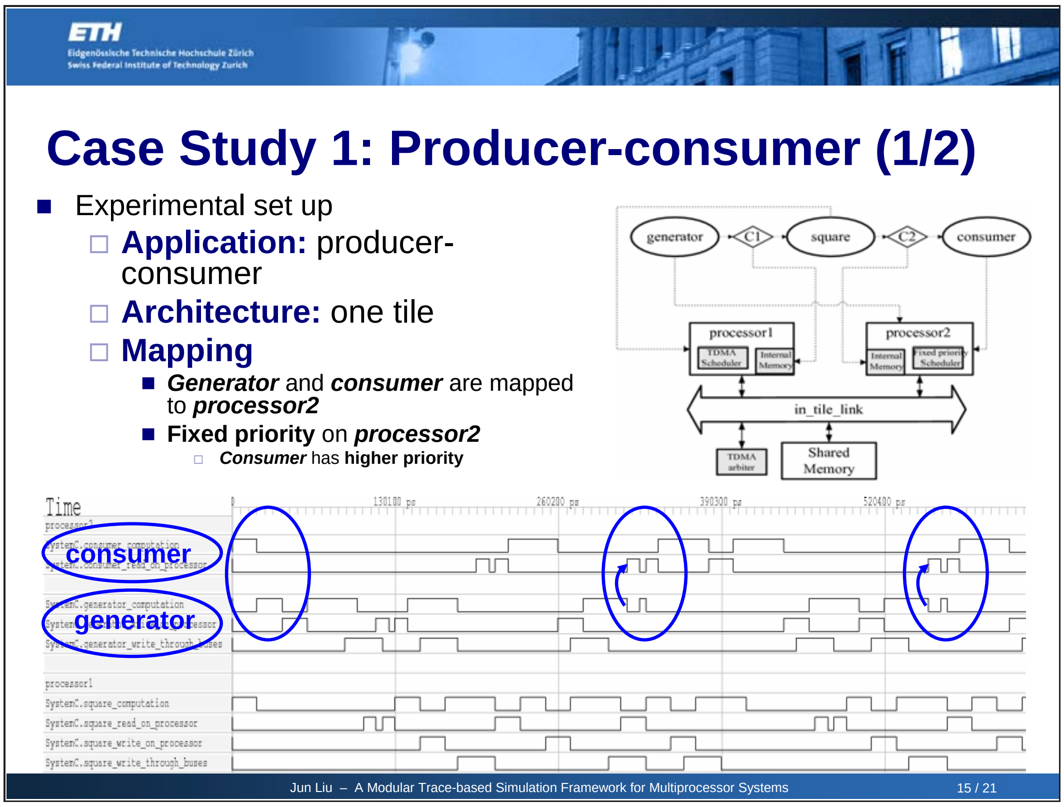
Jun Liu – A Modular Trace-based Simulation Framework for Multiprocessor Systems 13 / 21



Outline

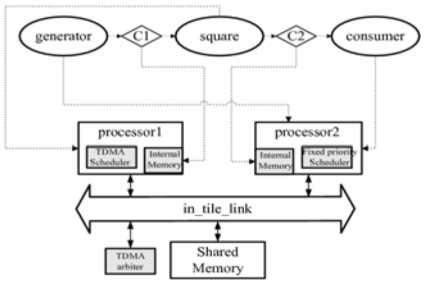
- Introduction
  - Context
  - Motivations and related work
  - Contribution
- Framework
  - Framework overview
  - Application modeling
  - Architecture modeling
  - Mapping
  - Simulation environment
- Case Studies
- Conclusion

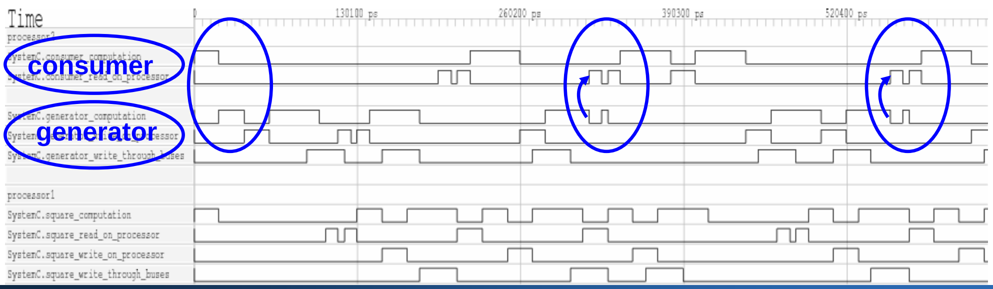
Jun Liu – A Modular Trace-based Simulation Framework for Multiprocessor Systems 14 / 21



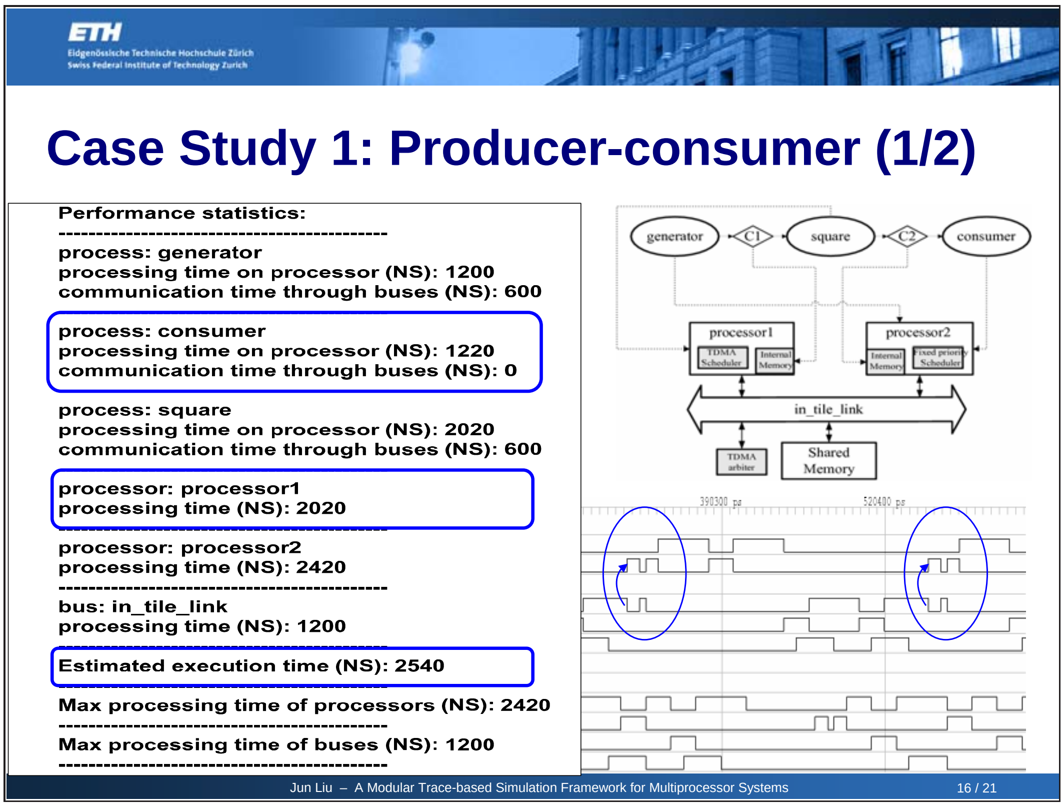
Case Study 1: Producer-consumer (1/2)

- Experimental set up
  - **Application:** producer-consumer
  - **Architecture:** one tile
  - **Mapping**
    - ***Generator*** and ***consumer*** are mapped to ***processor2***
    - **Fixed priority** on ***processor2***
      - ***Consumer*** has **higher priority**





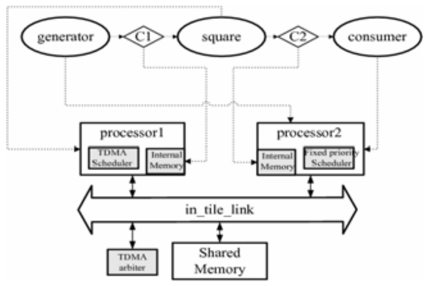
Jun Liu – A Modular Trace-based Simulation Framework for Multiprocessor Systems 15 / 21

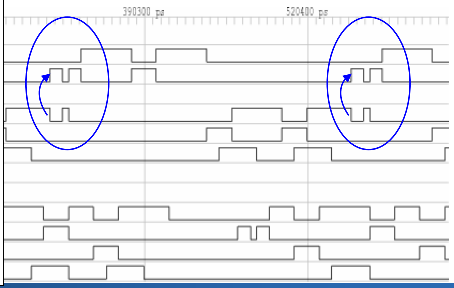


ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

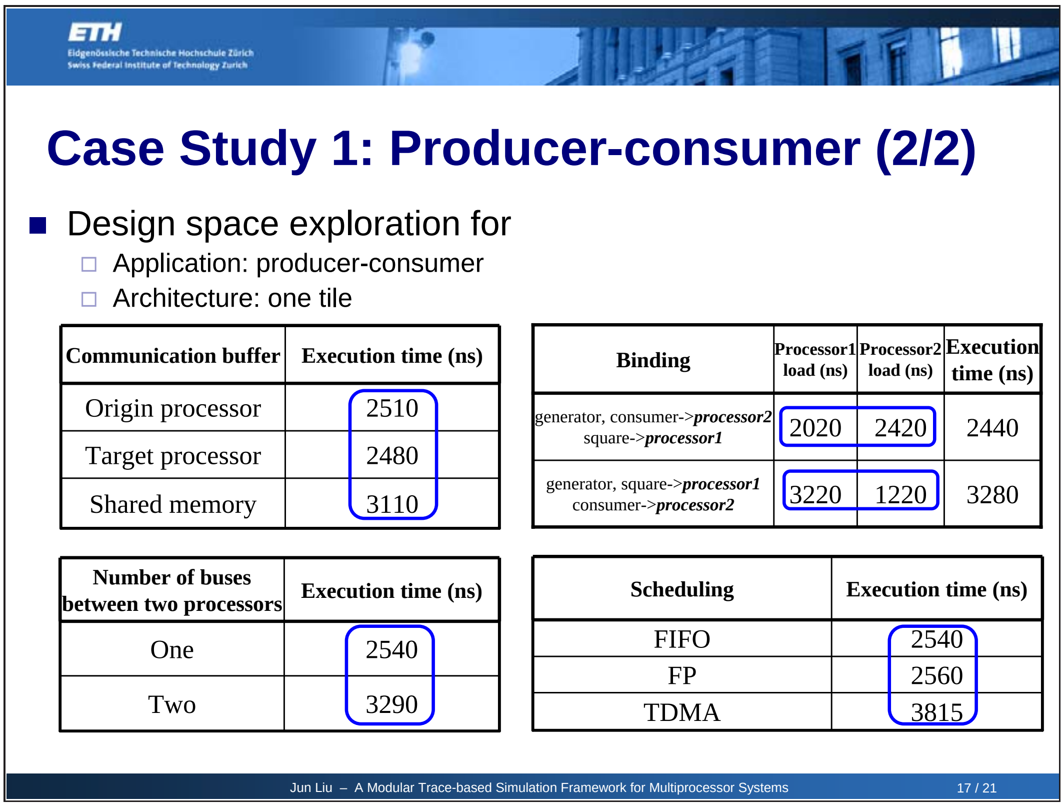
Case Study 1: Producer-consumer (1/2)

```
Performance statistics:
---------------------------------------------
process: generator
processing time on processor (NS): 1200
communication time through buses (NS): 600

process: consumer
processing time on processor (NS): 1220
communication time through buses (NS): 0

process: square
processing time on processor (NS): 2020
communication time through buses (NS): 600

processor: processor1
processing time (NS): 2020
---------------------------------------------
processor: processor2
processing time (NS): 2420
---------------------------------------------
bus: in_tile_link
processing time (NS): 1200
---------------------------------------------
Estimated execution time (NS): 2540
---------------------------------------------
Max processing time of processors (NS): 2420
---------------------------------------------
Max processing time of buses (NS): 1200
---------------------------------------------
```





Jun Liu – A Modular Trace-based Simulation Framework for Multiprocessor Systems 16 / 21



ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

Case Study 1: Producer-consumer (2/2)

- Design space exploration for
  - Application: producer-consumer
  - Architecture: one tile

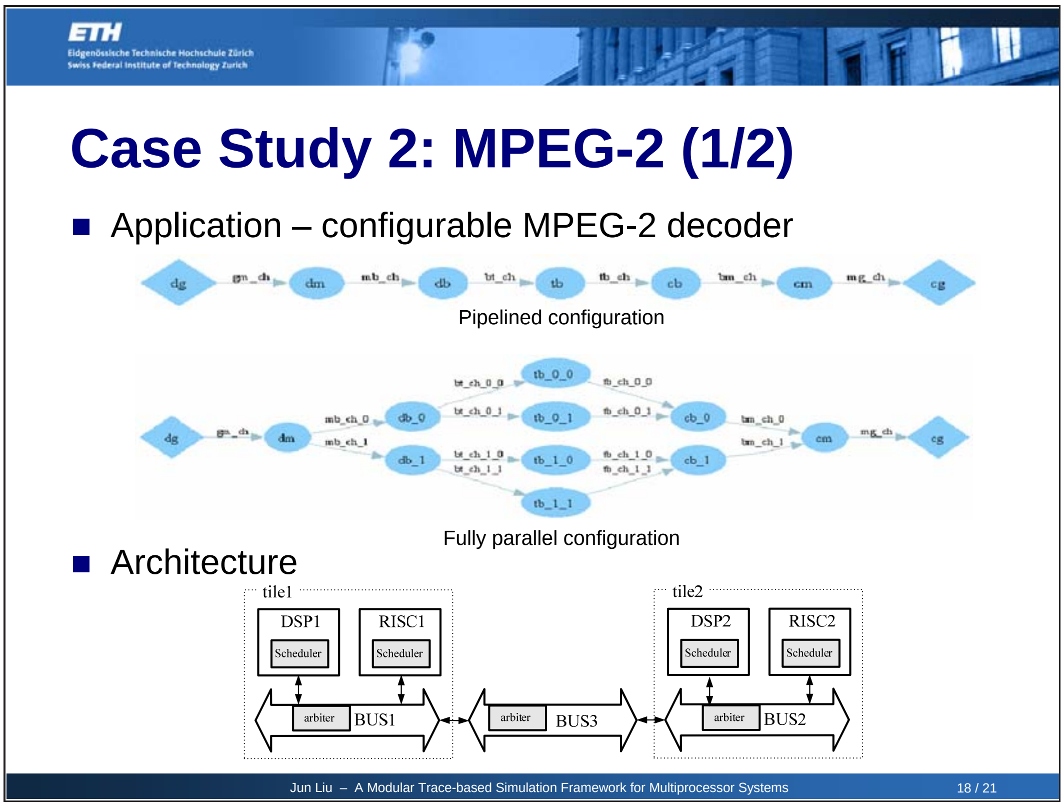
Communication buffer	Execution time (ns)
Origin processor	2510
Target processor	2480
Shared memory	3110

Binding	Processor1 load (ns)	Processor2 load (ns)	Execution time (ns)
generator, consumer->***processor2*** square->***processor1***	2020	2420	2440
generator, square->***processor1*** consumer->***processor2***	3220	1220	3280

Number of buses between two processors	Execution time (ns)
One	2540
Two	3290

Scheduling	Execution time (ns)
FIFO	2540
FP	2560
TDMA	3815

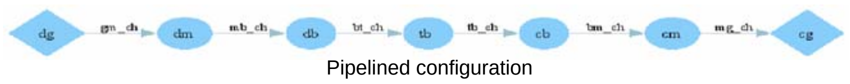
Jun Liu – A Modular Trace-based Simulation Framework for Multiprocessor Systems 17 / 21



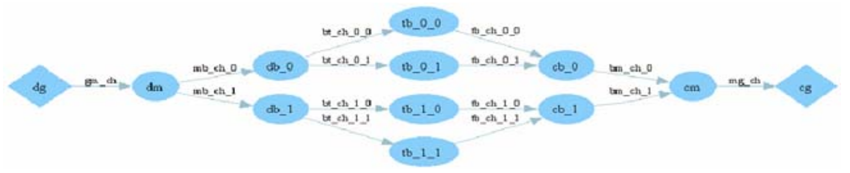
ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

Case Study 2: MPEG-2 (1/2)

- Application – configurable MPEG-2 decoder

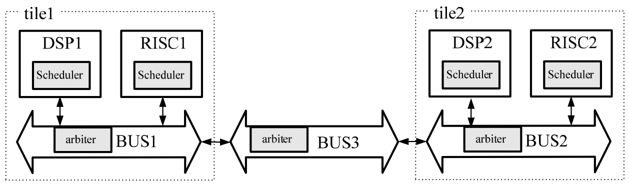


Pipelined configuration

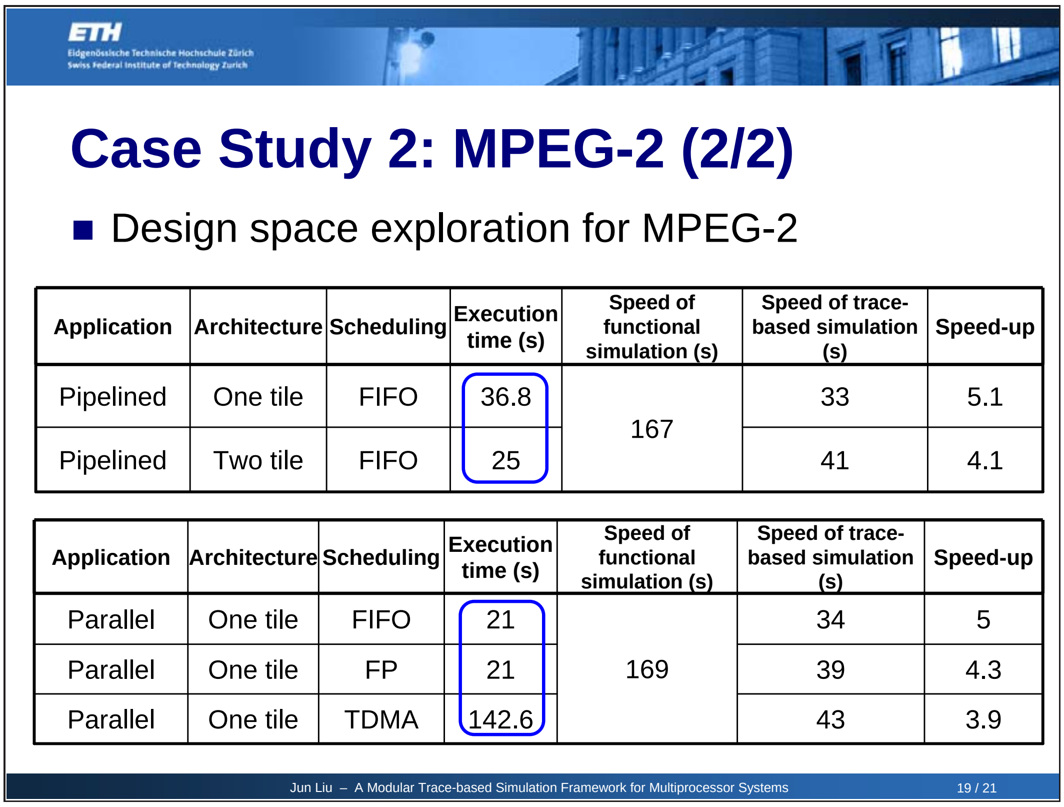


Fully parallel configuration

- Architecture



Jun Liu – A Modular Trace-based Simulation Framework for Multiprocessor Systems 18 / 21



ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

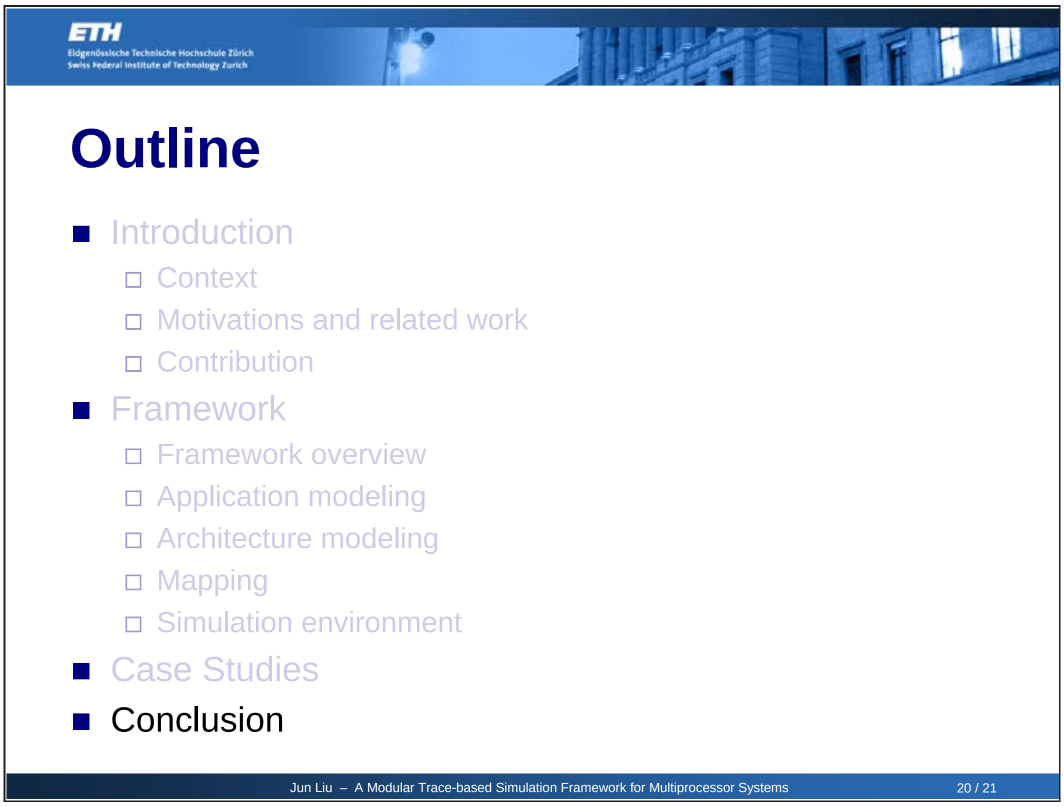
Case Study 2: MPEG-2 (2/2)

- Design space exploration for MPEG-2

Application	Architecture	Scheduling	Execution time (s)	Speed of functional simulation (s)	Speed of trace-based simulation (s)	Speed-up
Pipelined	One tile	FIFO	36.8	167	33	5.1
Pipelined	Two tile	FIFO	25		41	4.1

Application	Architecture	Scheduling	Execution time (s)	Speed of functional simulation (s)	Speed of trace-based simulation (s)	Speed-up
Parallel	One tile	FIFO	21	169	34	5
Parallel	One tile	FP	21		39	4.3
Parallel	One tile	TDMA	142.6		43	3.9

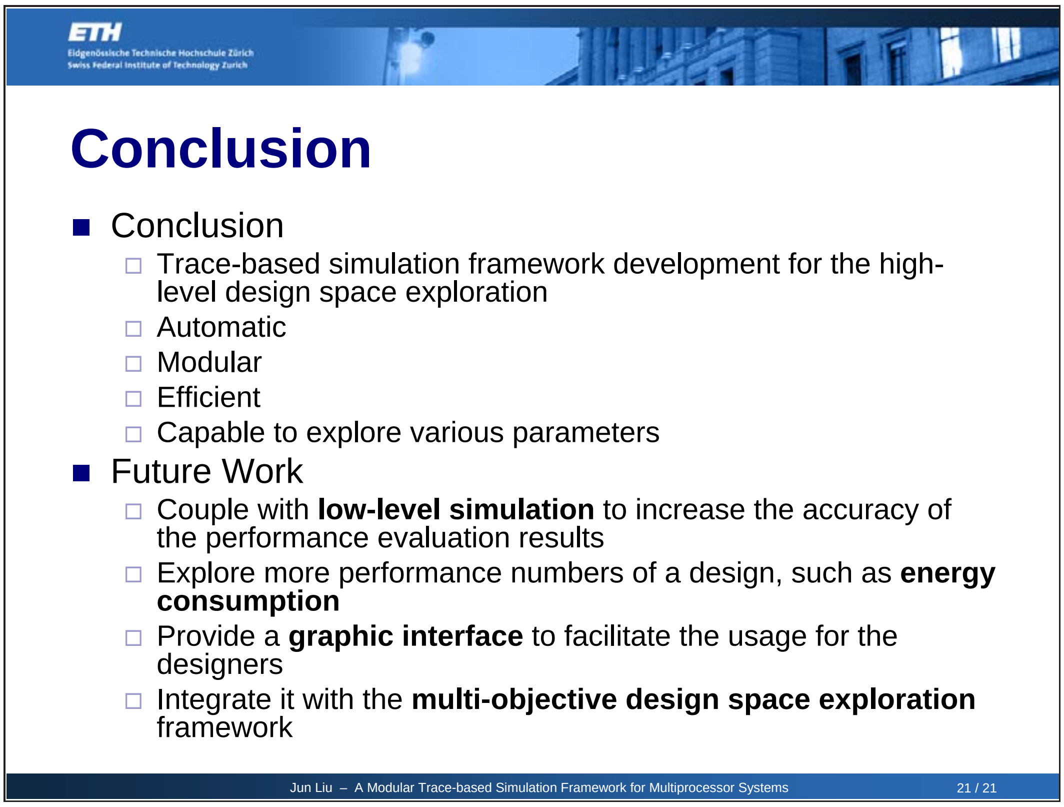
Jun Liu – A Modular Trace-based Simulation Framework for Multiprocessor Systems 19 / 21



Outline

- Introduction
  - Context
  - Motivations and related work
  - Contribution
- Framework
  - Framework overview
  - Application modeling
  - Architecture modeling
  - Mapping
  - Simulation environment
- Case Studies
- Conclusion

Jun Liu – A Modular Trace-based Simulation Framework for Multiprocessor Systems 20 / 21



Conclusion

- Conclusion
  - Trace-based simulation framework development for the high-level design space exploration
  - Automatic
  - Modular
  - Efficient
  - Capable to explore various parameters
- Future Work
  - Couple with **low-level simulation** to increase the accuracy of the performance evaluation results
  - Explore more performance numbers of a design, such as **energy consumption**
  - Provide a **graphic interface** to facilitate the usage for the designers
  - Integrate it with the **multi-objective design space exploration** framework

Jun Liu – A Modular Trace-based Simulation Framework for Multiprocessor Systems 21 / 21

Acknowledgments

I would like to take this opportunity to thank all the people who have dedicated to the accomplishment of my master thesis. First of all, I would like to thank **Prof. Dr. Lothar Thiele** for providing the opportunity to write this master thesis at the Computer Engineering and Networks Laboratory, as well as for the inspiring discussions and suggestions on this project.

I would like to thank my thesis advisors, **Iuliana Bacivarov** and **Kai Huang** for their constant dedication throughout the whole procedure of my work. I have benefited a lot from their enthusiastic and patient guidance, their solid foundation and professional insights.

I would also like to thank all the staff of the Services Group of the Computer Engineering and Networks Laboratory for providing a perfect working environment.

Finally, I would like to thank my family and friends for their understanding and support of my choice. Their encouragement always makes me confident whenever I am confronted with challenges.

Bibliography

[1] L. Thiele, I. Bacivarov, W. Haid, and K. Huang, “Mapping applications to tiled multiprocessor embedded systems,” in *ACSD ’07: Proceedings of the Seventh International Conference on Application of Concurrency to System Design*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 29–40.

[2] SystemC Community Website. [Online]. Available: http://www.systemc.org/

[3] A. Baghdadi, N.-E. Zergainoh, W. O. Cesário, and A. A. Jerraya, “Combining a performance estimation methodology with a hardware/software codesign flow supporting multiprocessor systems,” *IEEE Trans. Softw. Eng.*, vol. 28, no. 9, pp. 822–831, 2002.

[4] I. Bacivarov, A. Bouchhima, S. Yoo, and A. A. Jerraya., “Chronosym: a new approach for fast and accurate soc cosimulation,” *International Journal of Embedded Systems.*, vol. 1, pp. 103–111, 2005.

[5] T. Wild, A. Herkersdorf, and R. Ohlendorf, “Performance evaluation for system-on-chip architectures using trace-based transaction level simulation,” in *DATE ’06: Proceedings of the conference on Design, automation and test in Europe.* 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2006, pp. 248–253.

[6] P. Kalla, X. S. Hu, and J. Henkel, “A flexible framework for communication evaluation in soc design,” in *ASP-DAC ’05: Proceedings of the 2005 conference on Asia South Pacific design automation.* New York, NY, USA: ACM Press, 2005, pp. 956–959.

[7] A. D. Pimentel, C. Erbas, and S. Polstra, “A systematic approach to exploring embedded system architectures at multiple abstraction levels,” *IEEE Trans. Comput.*, vol. 55, no. 2, pp. 99–112, 2006.

[8] G. Kahn, “The Semantics of a Simple Language for Parallel Programming,” in *Proc. IFIP Congress*, North Holland Publishing Co, 1974.

[9] K. Lahiri, A. Raghunathan, and S. Dey, "System-Level Performance Analysis for Designing On-Chip Communication Architectures," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 20, no. 6, pp. 768–783, June 2001.

[10] SHAPES Project Website. [Online]. Available: http://www.shapes-p.org

[11] P. S. Paolucci, A. A. Jerraya, R. Leupers, L. Thiele, and P. Vicini, "SHAPES: A Tiled Scalable Software Hardware Architecture Platform for Embedded Systems," in *Proc. of the 4th Int. Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'06)*, Seoul, Korea, Oct. 2006, pp. 167 –172.

[12] "XML Schema,"
`http://www.w3.org/XML/Schema`.

[13] Official Website of the MPEG Committee. [Online]. Available:
http://www.chiariglione.org/mpeg

[14] MPEG Software Simulation Group (MSSG). [Online]. Available:
http://www.mpeg.org/MSSG

[15] *ISO/IEC 13818-2: Information technology — Generic Coding of moving pictures and associated audio information — Part 2: Video*, International Organization for Standarization, 1995.

[16] E. Zitzler and L. Thiele, "Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach," *IEEE Trans. Trans. Evol. Comput.*, vol. 3, no. 4, pp. 257–271, Nov. 1999.