ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Andreas Huber

MPSoC Testbed for Analyzing Execution of Process Networks

Semester Thesis, SA-2008-26
September until December 2008

Advisors: Wolfang Haid, Kai Huang
Professor: Prof. Dr. Lothar Thiele

Abstract

Real-time applications are increasingly implemented on multi-processor system-on-chips. It is a challenge to develop applications for these systems. Many steps are necessary until an application for a specific multi-processor architecture can be executed and analyzed.

A testbed is implemented in this semester thesis that allows to execute process networks in a cycle-accurate Multi-Processor ARM (MPARM) simulator. As a result, the execution trace is extracted and a basic modular performance analysis model is created.

The traces allow insight into the execution of a real-time application on the MPARM platform. A computation and approximation for workload curves based on simulation results is introduced in this thesis. Workload curves are used to model the processes in the analysis model.

Acknowledgements

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

I would also like to thank my advisors Wolfgang Haid and Kai Huang for their constant support during this semester thesis. They helped me to solve the difficult problems of this thesis. Without their assistance, this work would not have been possible.

Furthermore, I would like to thank my family and my friends for supporting and motivating me during this thesis.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Real-time systems are implemented on MPSoC (multi-processor system-on-chip) today. Multi-processor systems are attractive because they offer higher performance and lower energy consumption than single-processor systems.

On the other hand, the development and analysis of an application implemented for a MPSoC is difficult. One needs to have a specific run-time system for the MPSoC in question and the application has to be implemented for that system. Further, a development environment has to be set up to be able to run or simulate the system to obtain relevant data about the execution.

An application developer would like to concentrate on modeling his problem. He will appreciate a development system that allows to analyze the real-time behaviour of his MPSoC in a easy way.

1.2 Goal

The goal of this project is to provide an environment that facilitates the analysis of real-time applications on MPSoC by assembling a tool chain that given a system specification allows to easily obtain system properties from low-level simulation.

In particular, the goal is to extract the system execution trace, to characterize the task workloads quantitatively and to obtain arrival curves for each task.

The target platform is Multi-Processor ARM (MPARM), which is a homogeneous multi-processor platform that integrates up to 26 ARM7 processors [2]. The real-time operating system running on this platform is RTEMS (Real-Time Executive for Multi-Processor Systems) [6].

1.3 Contribution

The described goal is reached by setting up a testbed which allows to execute a system specified by a DOL specification [8] mapped to the MPARM architecture, in a cycle accurate simulator.

The execution trace is extracted from simulation and converted to a file in VCD format (value change dump) [9]. Such a trace can be displayed with an open source software like GTKWave [3].

Further, workload curves are computed from the simulation results to characterize the task workload and data is extracted, that allows to compute the arrival curves for each task.

The testbed is described in detail in the chapter 2.

In chapter 3, the computation of the workload curves is described and an approximation is introduced that addresses the trade-off between the tightness and the complexity of the workload curves.

Chapter 2

MPSoC Testbed

2.1 Structure of the Testbed

The testbed embeds various tasks in an Apache Ant script [1]. An overview of the basic structure of the testbed is given in Fig. 2.1. The individual parts of the testbed that are used for the MPARM simulation and the analysis of the log file are listed in more detail in Fig. 2.2. In the following sections, a closer look at the different parts is taken.

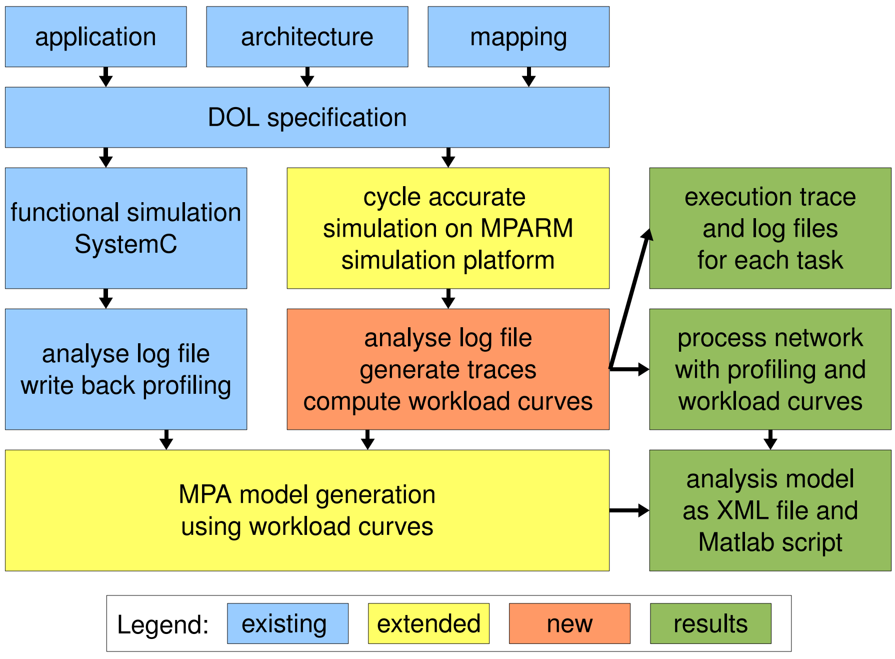

application
architecture
mapping
DOL specification
functional simulation SystemC
cycle accurate simulation on MPARM simulation platform
execution trace and log files for each task
analyse log file write back profiling
analyse log file generate traces compute workload curves
process network with profiling and workload curves
MPA model generation using workload curves
analysis model as XML file and Matlab script
Legend: existing extended new results

Figure 2.1: Basic structure of the testbed.

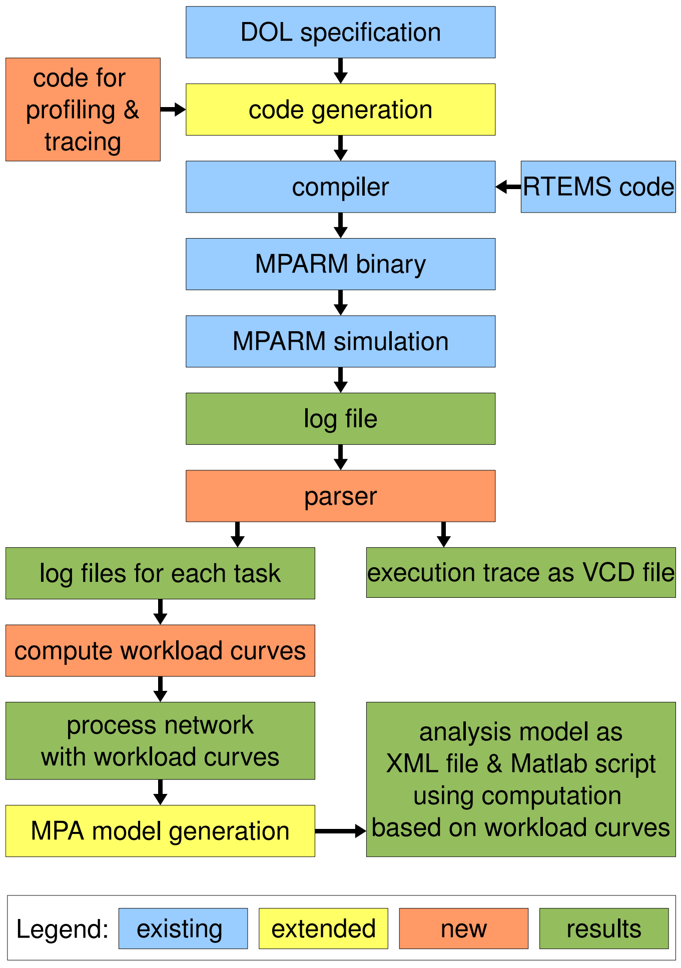

DOL specification
code for profiling & tracing
code generation
compiler
RTEMS code
MPARM binary
MPARM simulation
log file
parser
log files for each task
execution trace as VCD file
compute workload curves
process network with workload curves
analysis model as XML file & Matlab script using computation based on workload curves
MPA model generation
Legend: existing extended new results

Figure 2.2: Detailed steps of the testbed.

2.2 Required Input: DOL Specification

As input for the testbed, a DOL (distributed operation layer) specification is required as defined in [8]. A DOL specification consists of three XML files, process_network.xml, architecture.xml and mapping.xml as well as the code files for the defined processes. These files describe the application in terms of a process network, the hardware architecture and the mapping of the processes onto the processors.

The process network file defines the processes and the channels that interconnect them. These channels are FIFO queues with a maximum fill size. For each process, a C or C++ source file is needed. The code has to match the requirements given from the DOL framework, mainly by defining the required functions *processname*_init() and *processname*_fire() and by using the functions DOL_read() and DOL_write() to communicate over the channels.

The architecture file describes a hardware architecture by defining the available processors. A type is assigned to each processor. In this work, always the same architecture file is used which describes an MPARM [2] architecture consisting of eight RISC processors, connected over a shared bus.

The mapping file finally maps each process onto one of the processors. Additionally, it is possible to define a scheduling policy for each processor. Only the preemptive fixed priority scheduling policy is taken into account in the current implementation of the DOL code generation for the MPARM simulation platform. The same priority is used for each process if no scheduling policy is explicitly defined in the mapping file.

An example for such a DOL specification is given in Fig. 2.3.

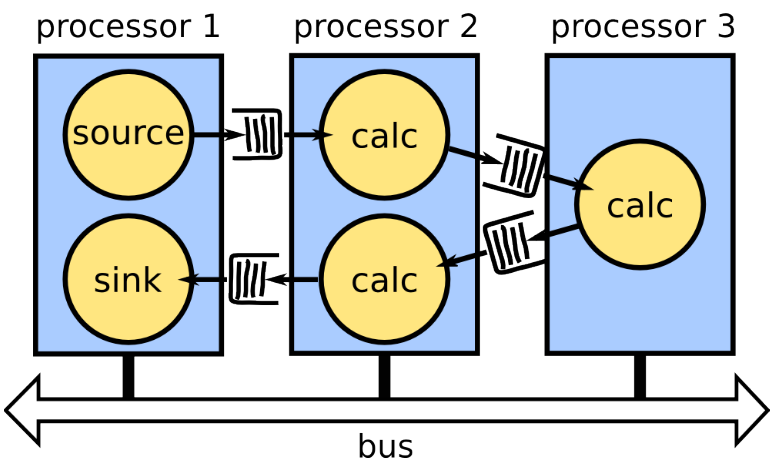


Figure 2.3: Example of a DOL specification for the MPARM architecture.

The input sources for the MPSoC system testbed can be specified with the attribute *-Dsource=/path/to/source/dir* when calling the ant script. The specified source folder has to contain the three files *process_network.xml*, *architecture.xml* and *mapping.xml* as introduced above as well as a folder named *src* containing the source files for the processes.

After the sources are copied, the three XML files are validated against the DOL schemata and the process network is flattened.

2.3 Functional Simulation

As the first step in the testbed, the application is compiled and executed on the host machine using a functional simulation.

In the functional simulation, only the application, the process network and the source files, are taken in to account.

This simulation is done with the DOL framework [8] and SystemC [7]. In the DOL framework, the code generation is implemented. The resulting application is compiled and executed. During the execution, some profiling information is written to a log file. This information is extracted and back annotated to the process network XML file again using the DOL framework.

The functional simulation is executed because some of the profiling data is later used to generate the basic MPA model, namely the amount of data transferred over each channel and the token production and consumption rates of processes.

2.4 MPARM Simulation

The low level simulation is done with a cycle-accurate MPARM simulator. The operating system RTEMS is used [6]. RTEMS is a real-time operating system for multi-processor systems.

The code generation is done with the DOL framework. The application *dol.Main* reads in the process network, the architecture and the mapping.

Then, the source code is generated for the target architecture. All necessary sources that are needed in this process are located in the packet *dol.visitor.rtems*.

We want to find out which process is running at which time. For that, a function is implemented that is called from the operating system on each context switch. The current clock count and the process id of the process that starts running is logged in this function, such that it can be determined which process is running at any point in time from the logging of these context switches.

The clock count is logged at further points in the simulation, to get information of the different processes. On the begin and at the end of each firing, reading and writing of each process the clock count is logged.

The necessary commands to log this information are included in the code generation if the compiler flag WORKLOAD_EXTRACT is set (which is done by default).

This output is logged to the file log.txt during the simulation.

2.4.1 Logging in the MPARM Simulator

The standard printf cannot be used to log data in the MPARM simulation because printf is rather computation-intensive and would have a considerable effect on the execution. Instead, two of the debug functions available in the MPARM simulator, SHOW_DEBUG and SHOW_DEBUG_INT, are used.

These two functions allow to dump messages to the shell from which MPARM has been started. The function SHOW_DEBUG allows to log strings, the function SHOW_DEBUG_INT logs integers in hexadecimal format. On each output line, additionally a text like "Processor 0 - " is prepended, indicating the number of the processor that printed this message.

It is still useful to use printf in some situations. But the text printed by printf is forwarded to a serial UART device and not the shell from which MPARM has been started. It may be useful to redirect the output of printf to the shell, if the application in question uses the printf to print out some results. This redirection is done by redefining printf to use the function sprintf to generate the message string and the function SHOW_DEBUG to print it to the shell.

The compiler flag PRINT_TO_DEBUG has to be set to use this redirection. It is turned off by default.

The two debug functions SHOW_DEBUG and SHOW_DEBUG_INT are written in assembly code and it takes about 50 cycles to log a string of length 11 and an integer. About 5000 cycles would be necessary to log the same information as a formatted string including an integer using the function printf.

The only draw back is that the parsing of the log file will be more complicate by using multiple SHOW_DEBUG and SHOW_DEBUG_INT commands to log one message because then the different parts are logged on separate lines and a single message may be interleaved with messages from other processors.

2.5 Parsing of the Log File

The log files are parsed in several steps. In a first step, the logged messages are filtered out and ordered correctly. In a second step, the logged messages are analysed and the results are generated.

2.5.1 Parsing the Log File

Each logged message spreads over two or three lines in the log file because of the used logging technique, as mentioned above. So in the first step, the lines in the log file that belong to the same logged message are recombined and a new log is created, where each message is exactly one line.

The messages contain a time stamp, the processor number, the message type and additional optional information.

The recombined messages are written back to the file log.parsed.txt.

One would expect, that the messages are logged in the sequence as they are raised during the simulation, meaning that the time stamps of the messages in the log file should be always increasing. But this is actually not true. While the messages coming from one processor are always logged in the file in the sequence as they are raised, it happens that messages from different processors are not logged exactly in the sequence of their time stamps.

It is necessary to process the messages in the correct sequence for the further processing of the log file. The messages in the log file are sorted according to their time stamps to bring them into the correct sequence.

This is done with the unix command sort. The sorted log file is stored in log.parsed.sorted.txt.

Now all messages are listed in sequence and ready to be analysed in this log file.

2.5.2 Analysing the Log Messages

Now, the different logged messages are evaluated. As results, two trace files are generated, as well as different log files for each process.

In the traces, the processes and the channels are displayed. They are explained in more details in the next section.

For each firing, the execution demand is computed and logged to the file *processname*_load.txt. The execution demand of a firing is computed as the number of cycles between the corresponding start and end fire entries in the log minus the time when the process was not running in this interval. This data will be used to compute the workload curves.

Further, various log files are generated for each process. They each contain all the time stamps of one type of event, as indicated by their name. They can be used for example to compute the arrival curve for this process. The files are *processname*_start_fire.txt, *processname*_end_fire.txt, *processname*_start_read.txt, *processname*_end_read.txt, *processname*_start_write.txt and *processname*_end_write.txt.

2.6 Traces

The trace files are exported as VCD files. VCD stands for value change dump and is an IEEE standard [9]. The traces can be displayed using for example the application GTKWave [3].

Two different trace files are generated. They both contain the same information, but have different graphical representations. In Fig. 2.4 and 2.5, an example is shown for both representations.

The example shows three processes (generator, square and consumer) running on one processor. They communicate over the channels 1 and 2.

The trace file contains an entry for each process and each channel.

For the processes, the trace indicates when they are activated and deactivated, the start and end of each firing, the start and end of each reading as well as the start and end of each writing.

In the first representation (stored in trace1.vcd), the start and end of the firing, reading or writing are marked with peaks. Upward peaks stand for start, downward peaks stand for end. The peaks have different sizes. The largest peaks stands for the firing, the medium peak for the reading and the small peak for the writing.

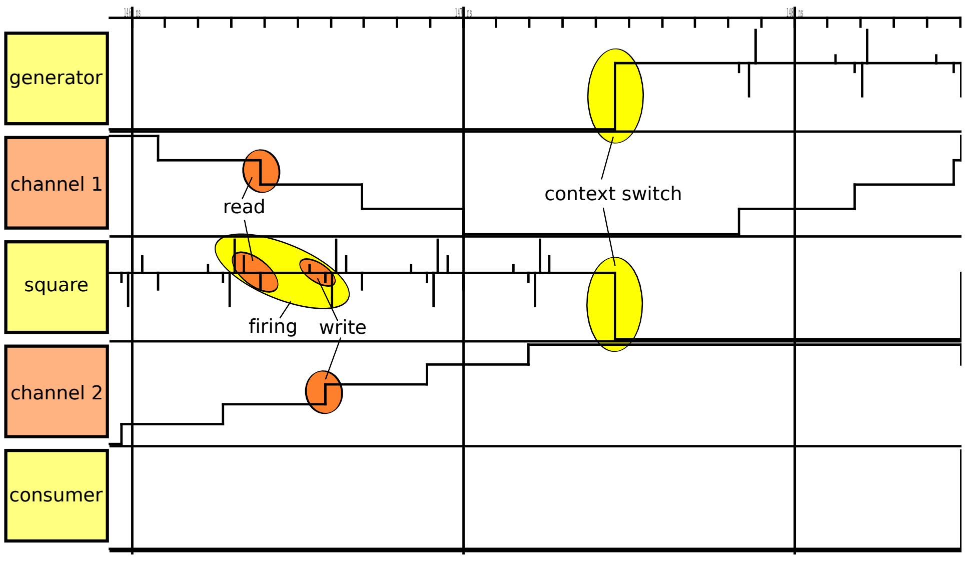


Figure 2.4: Example trace file with the first representation.

In the second representation (stored in trace2.vcd), each event type is assigned a different bit. On the start event, the bit is set, on the end event it is cleared.

In both cases, the activation is indicated with the highest bit set, and the deactivation of a process, with the clearing of the highest bit.

The first representation is more compact, because all information about a process is combined in one trace. The second representation is handy if it is preferred to split the firing, read and write into separate traces.

For each channel, the fill level of the queue is displayed at each point in time. The exact moment, when a read or a write to a queue happens in the simulation is somewhere between the logged start and end time of the reading or writing of the corresponding process.

The software tries to mark the reads and writes to the channel at the most reasonable time. This means that in the normal case the channel is increased and decreased on the time of the end write and end read. The time of the start read and start write are not used because the read and write commands are blocking, meaning that the read blocks if the channel queue is full and the write blocks if the channel queue is empty.

But there is a case where the update of the channel at the end read or end write is not reasonable, when using fixed priority scheduling. This is, if for example, a process with high priority is blocked because it wants to read from an empty queue. If another process is running on the same processor and writes a token to this channel, it will be preempted by the process with higher priority immediately after writing a token to the queue, but before the end write was logged.

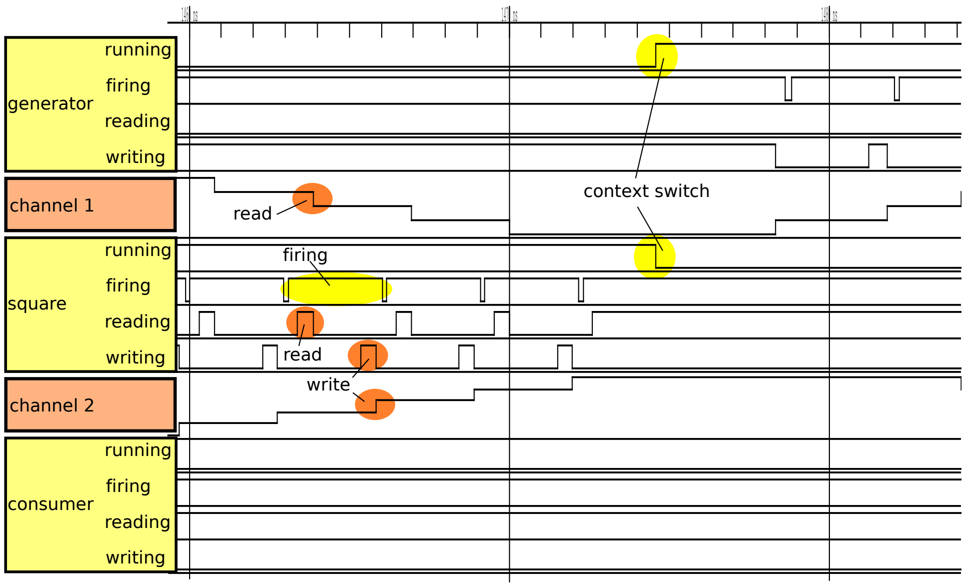


Figure 2.5: Example trace file with the second representation.

If the channel is not increased in the trace until the logged end write, the channel will be decreased to a fill level of minus one token by the process with higher priority. The opposite may happen with a process with high priority that is blocked by writing to a channel with a full queue.

These unreasonable fill levels are avoided by using an earlier time of the read and write to the channel in these cases. The fill level of the queues are updated in such cases at the time of the context switches that happens because a process with higher priority is no longer blocked.

The channels are identically displayed in both trace files.

2.7 Workload Curve Computation

As the next step in the testbed, the workload curve is computed for each process from the corresponding file *processname*_load.txt, which contains the execution demand for each processed event by this process.

The workload curves are computed using Matlab with the function workloadcurve.m which was implemented for this task and which will be explained in the chapter about workload curves.

The workload curves are written back to the process network as profiling data, to be used for the generation of the modular performance analysis model [5].

2.8 MPA Model

As the last step in the testbed, a basic modular performance analysis model (MPA) is created from the given DOL specification and the computed workload curves.

In the analysis model, each processor is modeled by a resource and one additional resource is added to model the bus. Each process is modeled as a greedy processing component (GPC) and is mapped to the corresponding resource. Each channel is also described as a GPC bound to the resource modeling the bus. The scheduling is taken into account, too.

The MPA model is exported into the XML file dolanalysismodel.xml. This file is finally transformed into a Matlab script for the MPA Matlab toolbox. In particular, two different Matlab models are built, one using the workload curves (dolanalysismodel.m), and one using just the worst-case / best-case execution demand (dolanalysismodel_iwl.m).

2.9 Examples

In this section, the process network given in Fig. 2.3 is executed as an example to show the resulting execution trace and the generated MPA model.

The example is executed with two different scheduling policies.

In the first case, equal priorities are used for all processes. The resulting trace is shown in Fig. 2.6. As can be seen in the trace, context switches happen whenever a process is blocked either by reading from an empty queue or writing to a full queue.

Different properties can be easily seen in the trace: For example, the mutual exclusion of the processes sink and source running on the same processor. Or that the process in the middle executes whenever possible, but occupies the third processor only 50% of the time. We can further see, that the fill level of the last channel is never greater than two. So the queue size of the last channel could be reduced to two without any loss in performance.

In the second case, different priorities are chosen for the processes. The resulting trace is shown in Fig. 2.7. The process source has higher priority than the process sink on the first processor, and the process square 1 has higher priority than the process square 3 on second processor. Process square 2 is the only process running on the third processor, so no scheduling is needed.

The effect of the scheduling is clearly visible. The first three processes fire whenever they are not in a blocking write. The first three queues are filled up with tokens at the beginning. From then on, as soon as a token is read out of one of these first three queues, the process writing to that queue is activated and writes a new token to the queue.

We can further see, that the fill level of the last channel is never greater than one. So the queue size of the last channel could be reduced like in the first example without any loss in performance.

The MPA model that is generated for the example using preemptive fixed priority is shown in Fig. 2.8. To give a feeling for the generated files, dolanalysismodel.xml, containing the MPA model in an XML file, as well as dolanalysismodel.m, containing the generated Matlab script, are displayed, too.

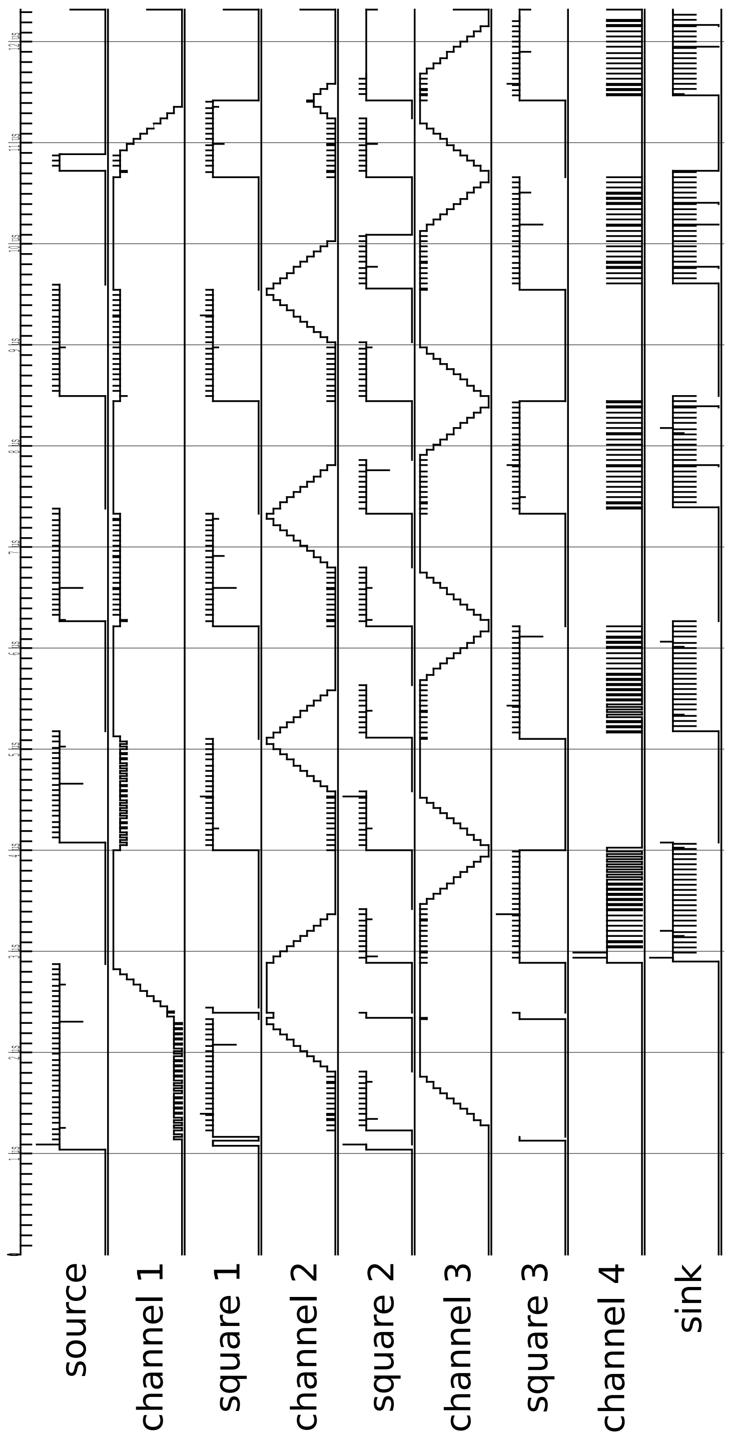


Figure 2.6: Execution trace using cooperative scheduling.

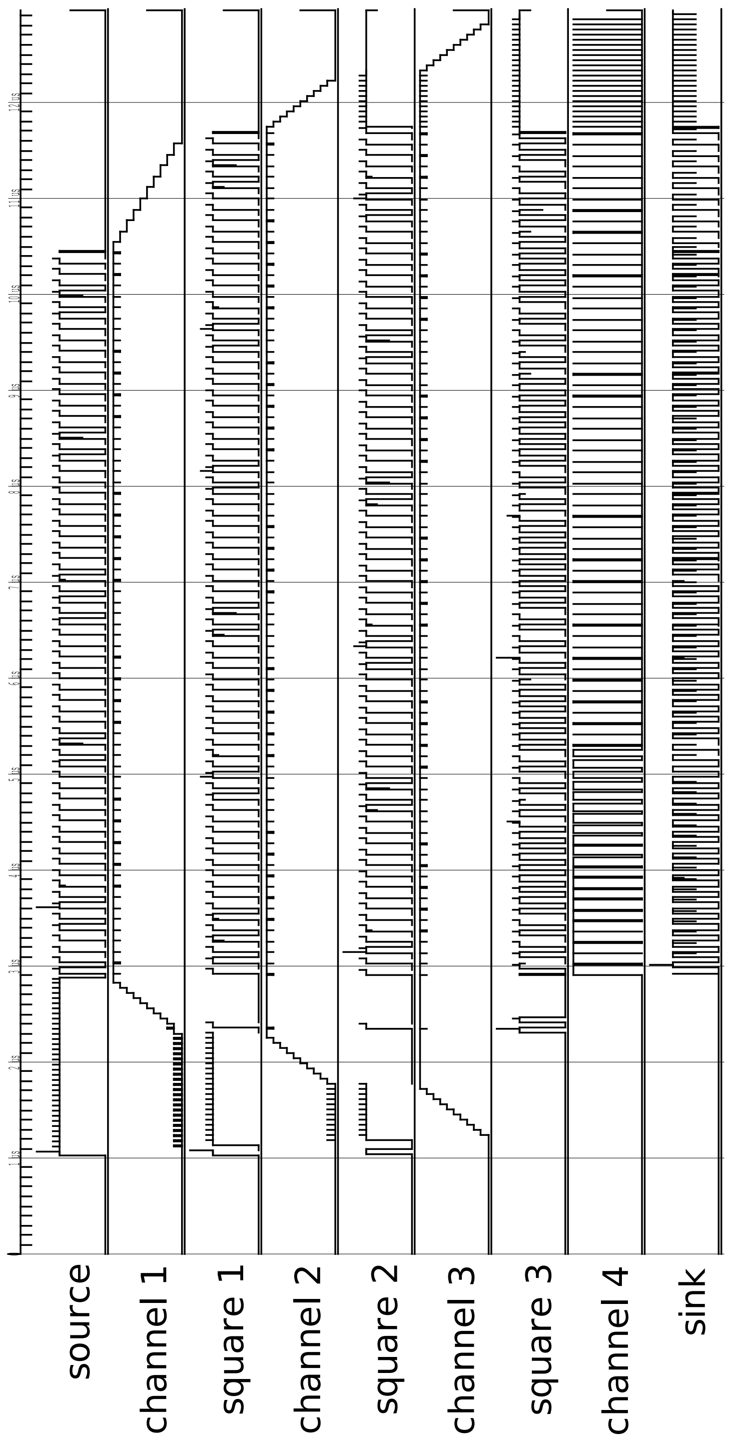
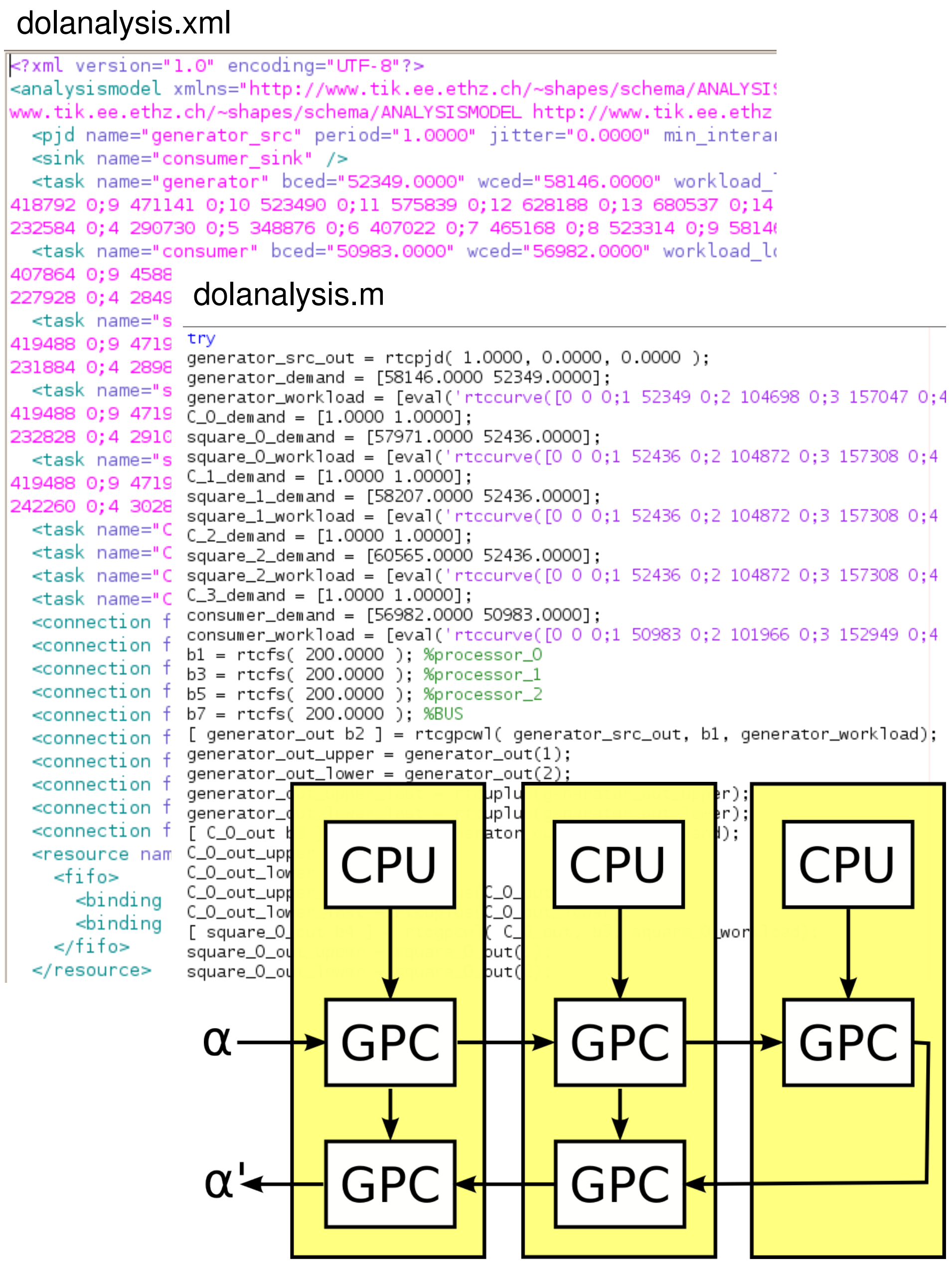


Figure 2.7: Execution trace using preemptive fixed priority scheduling.



```
<?xml version="1.0" encoding="UTF-8"?>
<analysismodel xmlns="http://www.tik.ee.ethz.ch/~shapes/schema/ANALYSIS
www.tik.ee.ethz.ch/~shapes/schema/ANALYSISMODEL http://www.tik.ee.ethz
  <pjd name="generator_src" period="1.0000" jitter="0.0000" min_intera
  <sink name="consumer_sink" />
  <task name="generator" bced="52349.0000" wced="58146.0000" workload_
418792 0;9 471141 0;10 523490 0;11 575839 0;12 628188 0;13 680537 0;14
232584 0;4 290730 0;5 348876 0;6 407022 0;7 465168 0;8 523314 0;9 5814
  <task name="consumer" bced="50983.0000" wced="56982.0000" workload_l
407864 0;9 4588
227928 0;4 2849
  <task name="s
419488 0;9 4719
231884 0;4 2898
  <task name="s
419488 0;9 4719
232828 0;4 2910
  <task name="s
419488 0;9 4719
242260 0;4 3028
  <task name="C
  <task name="C
  <task name="C
  <task name="C
  <connection f
  <connection f
  <connection f
  <connection f
  <connection f
  <connection f
  <connection f
  <connection f
  <connection f
  <connection f
  <resource nam
    <fifo>
      <binding
      <binding
    </fifo>
  </resource>
```

```
try
generator_src_out = rtcpjd( 1.0000, 0.0000, 0.0000 );
generator_demand = [58146.0000 52349.0000];
generator_workload = [eval('rtccurve([0 0 0;1 52349 0;2 104698 0;3 157047 0;4
C_0_demand = [1.0000 1.0000];
square_0_demand = [57971.0000 52436.0000];
square_0_workload = [eval('rtccurve([0 0 0;1 52436 0;2 104872 0;3 157308 0;4
C_1_demand = [1.0000 1.0000];
square_1_demand = [58207.0000 52436.0000];
square_1_workload = [eval('rtccurve([0 0 0;1 52436 0;2 104872 0;3 157308 0;4
C_2_demand = [1.0000 1.0000];
square_2_demand = [60565.0000 52436.0000];
square_2_workload = [eval('rtccurve([0 0 0;1 52436 0;2 104872 0;3 157308 0;4
C_3_demand = [1.0000 1.0000];
consumer_demand = [56982.0000 50983.0000];
consumer_workload = [eval('rtccurve([0 0 0;1 50983 0;2 101966 0;3 152949 0;4
b1 = rtcfs( 200.0000 ); %processor_0
b3 = rtcfs( 200.0000 ); %processor_1
b5 = rtcfs( 200.0000 ); %processor_2
b7 = rtcfs( 200.0000 ); %BUS
[ generator_out b2 ] = rtcgpcwl( generator_src_out, b1, generator_workload);
generator_out_upper = generator_out(1);
generator_out_lower = generator_out(2);
generator_c                  uplu                  er);
generator_c                  uplu                  er);
[ C_0_out b                  ator                  d);
C_0_out_upp
C_0_out_low
C_0_out_upp                  C_0_
C_0_out_low                  C_0_
[ square_0_                  ( C_                  wor
square_0_ou                  out(
square_0_ou                  out(
```

Figure 2.8: MPA model for example process network.

Chapter 3

Workloads

3.1 Introduction

It is important to have good estimation methods to characterize the behaviour of a given system in the design process of real-time embedded systems. Such systems are often characterized with a process network that contains multiple processes interconnected by channels. These processes execute iteratively and read a token from one or more incoming channels and write a token to outgoing channels in each iteration. These tasks may be mapped to the same processing unit or to different ones in a multi-processor environment. Typical examples of such systems are multimedia applications.

The performance analysis of such systems is used in the design space exploration to find good implementations and to verify that certain system properties are within some requested limits.

For this, mainly three classes of methods exist: Analysis, simulation and statistics. In this work, we set up a testbed where the execution times of the individual tasks of a specified system can be obtained using low-level simulation as introduced in chapter 2. This information will then be used to calibrate an analytic performance analysis model.

In the analytic performance analysis of real-time systems, one is usually interested in finding lower and upper bounds of various system properties such as the workload that an event stream imposes on a certain processing unit or the overall delay and required buffer sizes for the channels.

The obtained list of execution times from the simulation has to be transformed into an abstract representation of the workload that such an event stream imposes onto a processor. This abstract representation is used in the analytic performance analysis to describe the workload of a process for an incoming event stream. This abstraction is done by computing a lower and an upper bound for the execution demand for the different processes.

The analysis then leads to guaranteed bounds under the assumption that we have obtained a representative sample of execution times in the simulation.

We will use the modular performance analysis toolbox (MPA) [5] for the analytic performance analysis.

The main question addressed next is how the obtained execution demands are used to formulate some lower and upper bounds for the execution demands of the tasks. A simple way to do this, is just to take the observed worst-case and best-case execution demand for each task. But this may lead to over-pessimistic bounds, because in many situations most of the events for a given task will be between the worst- and best-case and not at the lower or upper bounds. Further, it may be that consecutive events have some dependencies between each other, e.g. that the worst-case can strictly not happen more than twice in a row. This is taken into account in a more complex model using workload curves.

Workload curves offer for each number of consecutive events a lower and an upper bound for the workload that these events cause on the processor used. The workload curve will always start with the worst-case and best-case for one event, like the simple solution. But for a number of e consecutive events, data dependencies lead to tighter bounds than e times the worst-case and best-case.

The workload curves can be computed from the observed execution times, as described later.

As a drawback, using these workload curves increases the complexity of the performance analysis. Instead of two values one now has to deal with a whole set of values, one pair for every number of consecutive events. Thus, a way to approximate the workload curves is needed.

This leads to a trade off between the tightness of the bounds that can be obtained and the complexity of the performance analysis.

In the following section, workload curves are introduced formally and an approximation that bounds the workload curve is defined.

3.2 Workload Curves

Given an event stream s, which is translated into a workload stream w. Let $W(e)$ be a counter which accumulates the total workload imposed to a processing unit by e consecutive events of the event stream.

Definition 1 (Workload Curve [10]). *For any workload stream w, the lower workload curve γ^l and the upper workload curve γ^u satisfy the relation:*

$$\gamma^l(u-v) \leq W(u) - W(v) \leq \gamma^u(u-v) \qquad \forall u, v \in \mathbb{R} \text{ and } 0 \leq v \leq u$$

Therefore, any sequence of e consecutive events on the workload stream w, will impose a total workload of at least $\gamma^l(e)$ and at most $\gamma^u(e)$.

According to that definition we can compute the workload curves from the observed workload stream w by choosing the maximum and minimum of workload imposed for an certain number of consecutive events.

$$\gamma^u(e) = \max_{t\in[0,\infty)} \{W(t+e) - W(t)\} \qquad e \in [0,\infty) \tag{3.1}$$

$$\gamma^l(e) = \min_{t\in[0,\infty)} \{W(t+e) - W(t)\} \qquad e \in [0,\infty) \tag{3.2}$$

In our case, $W(\cdot)$ is a staircase function such that $W(x) = W(\lfloor x \rfloor)$. This leads to workload curves which are staircase functions too:

$$\gamma^u(e) = \max_{n\in\mathbb{N}_0} \{W(n+\lceil e \rceil) - W(n)\} \qquad e \in [0,\infty) \tag{3.3}$$

$$\gamma^l(e) = \min_{n\in\mathbb{N}_0} \{W(n+\lfloor e \rfloor) - W(n)\} \qquad e \in [0,\infty) \tag{3.4}$$

It follows that the function $W(\cdot)$ only has to be evaluated at integer points for the computation of the workload curves.

3.2.1 Sub- and Superadditivity

From the definition of the workload curves follows that the upper workload curve is a subadditive and the lower workload curve is a superadditive function.

$$\gamma^u(u+v) \leq \gamma^u(u) + \gamma^u(v) \tag{3.5}$$

$$\gamma^l(u+v) \geq \gamma^l(u) + \gamma^l(v) \tag{3.6}$$

These inequalities can be formulated slightly differently, as in lemma 1.

Lemma 1. *The value of the upper workload curve at k times x is upper bounded by k times the value at x. Equally holds that the value of the lower workload curve at k times x is lower bounded by k times the value at x:*

$$\gamma^u(k\cdot x) \leq k\cdot\gamma^u(x) \qquad k \in \mathbb{N}_0 \tag{3.7}$$

$$\gamma^l(k\cdot x) \geq k\cdot\gamma^l(x) \qquad k \in \mathbb{N}_0 \tag{3.8}$$

Proof. This can be shown for the upper workload curve with the subadditivity property by induction.
Basis:

$$k = 1 \qquad \gamma^u(1\cdot x) = 1\cdot\gamma^u(x)$$

Step:

$$\begin{aligned}
\gamma^u((k+1)\cdot x) &= \gamma^u(k\cdot x + x) && \text{(subaddiditivity)}\\
&\leq \gamma^u(k\cdot x) + \gamma^u(x) && \text{(induction)}\\
&\leq k\cdot\gamma^u(x) + \gamma^u(x) &&\\
&= (k+1)\cdot\gamma^u(x) && \square
\end{aligned}$$

The proof for the lower workload curve follows analogously by changing the inequality and using the property of superadditivity.

3.3 Computation

The execution demands of the events can be extracted from the low level simulation of a given application as explained in chapter 2. This list will be used to compute the workload curves.

The computation can be done according to equations (3.3) and (3.4). The maximum and minimum execution demand is searched for each number of consecutive events in the observed event stream.

An important remark is, that if the workload curves are computed using an event stream containing n events, the workload curves can only be computed up to n consecutive events. In this case, the computed workload curves are not defined for $e \geq n$ consecutive events, and it is further not possible to extend them as introduced in the next section.

If the generated workload curves have to offer valid bounds for an event stream that contains any number of consecutive events, then it is necessary to use an infinite event stream to compute the workload curves. This means that the observed finite event stream has to be extended to infinity. A reasonable way to do that is to simply repeat the finite event stream.

Workload curves computed from a periodic event stream will be periodic too, so it is only necessary to compute the first period to get the full workload curves.

The following simple example shows the justification for doing this. The workload curves should be computed for a very short event stream with the loads $[1, 10, 1]$. If the workload curves are computed just for these three values, the resulting workload curve will have the values $[10, 11, 12]$ for the upper and $[1, 11, 12]$ for the lower workload curve. If we first extend the event stream by repetition, the resulting workload curves will have the values $[10, 11, 12, 22, 23, 24, \dots]$ for the upper and $[1, 2, 12, 13, 14, 24, \dots]$ for the lower workload curve.

The problem of the computation without repetition is, that the lower workload curve has an invalid lower bound for two consecutive events. The same happens with the upper workload curve if a stream with the loads $[10, 1, 10]$ is used.

3.4 Approximation

In the last section, the computation of the workload curves was introduced and it was argued why the observed event stream has to be periodically extended in order to compute the workload curves correctly.

In this section a way is introduced to bound the workload curves with linear functions. Later, the fully computed workload curve and the linear approximation will be combined to a approximation for the workload curves. There are two reasons why this approximation is preferred over the fully computed workload curves.

First, there exists a trade-off between the complexity of the representation and the tightness. The representation of the workload curve is less complicate the more of the values are replaced by a linear approximation.

In a real example, an event stream with hundreds or thousands of events may be used to compute the workload curves, but the workload curves can often be reasonably accurate approximated linearly for more than a few consecutive events.

The second reason is that the workload curves should not be computed for too many consecutive events because the workload stream obtained from simulation may not be representative for many consecutive events. The extreme case happens if the workload curve is fully computed. In a periodic extended event stream of n events are the best-case and worst-case workload for n consecutive events equal, just because only one observation of n consecutive events exists in the obtained event list.

Therefore, the workload curves should be computed only up to a fraction of the number of events in the list to get reliable workload curves.

Next, the approximation of the upper workload curve is introduced in details.

3.4.1 Linear Approximation of the Upper Workload Curve

The upper workload curve is approximated with the linear function $\tilde{\gamma}^u(\cdot)$.

Definition 2. *The linear approximation of the upper workload curve for a chosen $x \geq 0$ is defined as:*

$$\tilde{\gamma}^u(e) := g^u \cdot e + d^u = \frac{\gamma^u(x)}{x} \cdot e + d^u$$

The approximation is constructed by selecting a point x of the exact workload curve and computing the gradient $g^u = \gamma^u(x)/x$. After that, d^u is chosen such that the approximation is an upper bound for the exact workload curve within $[0, x)$:

$$d^u = \max_{0 \leq v < x} \{\gamma^u(v) - g^u \cdot v)\} \tag{3.9}$$

Because the workload curve is a staircase function, only a finite set of points has to be checked to compute d^u.

An example is shown in Fig. 3.1.

It is shown next that the exact workload curve is upper bounded by the constructed linear approximation. For that the following lemma is defined.

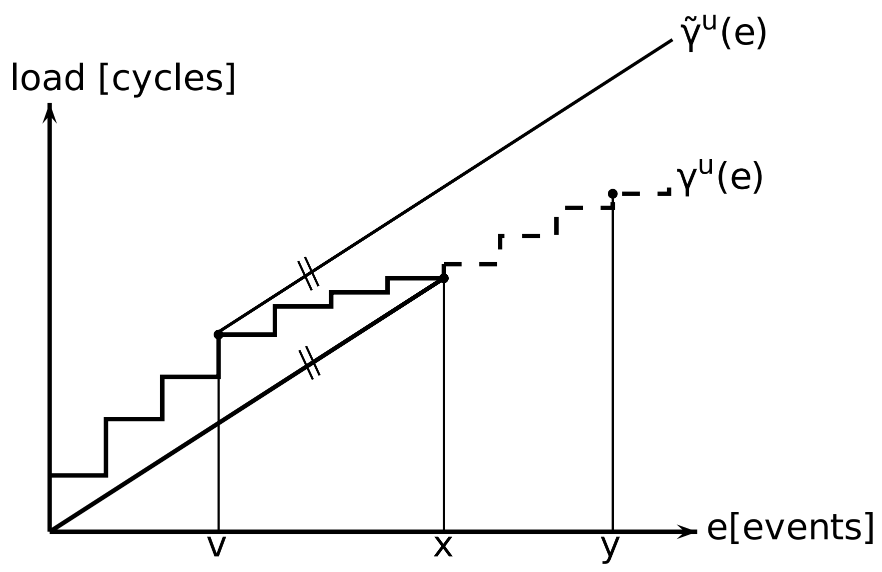


Figure 3.1: Approximation of the upper workload curve.

Lemma 2. *$\tilde{\gamma}^u(e)$ is a valid bound for $\gamma^u(e)$ for all $e \in [0, x)$:*

$$\tilde{\gamma}^u(e) \geq \gamma^u(e) \qquad \forall e \in [0, x)$$

This holds by construction as d^u is computed such that $\tilde{\gamma}^u(e)$ is greater or equal to $\gamma^u(e)$ for e smaller than x in (3.9).

Theorem 1. *The linear approximation $\tilde{\gamma}^u(\cdot)$ is a valid upper bound for the upper workload curve $\gamma^u(\cdot)$*

$$\tilde{\gamma}^u(e) \geq \gamma^u(e) \quad \forall x \geq 0$$

Proof.

$$\text{for } e = k \cdot x + v' \qquad k = \left\lfloor \frac{e}{x} \right\rfloor, v' \in [0, x)$$

$$\begin{aligned}
\tilde{\gamma}^u(e) &= \tilde{\gamma}^u(k \cdot x + v') && \text{(definition 2)}\\
&= \frac{\gamma^u(x)}{x} \cdot (k \cdot x + v') + d^u \\
&= \frac{\gamma^u(x)}{x} \cdot k \cdot x + \frac{\gamma^u(x)}{x} \cdot v' + d^u && \text{(definition 2)}\\
&= k \cdot \gamma^u(x) + \tilde{\gamma}^u(v') && \text{(lemma 2)}\\
&\geq k \cdot \gamma^u(x) + \gamma^u(v') && \text{(lemma 1)}\\
&\geq \gamma^u(k \cdot x) + \gamma^u(v') && \text{(subadditivity)}\\
&\geq \gamma^u(k \cdot x + v') \\
&= \gamma^u(e) && \square
\end{aligned}$$

3.4.2 Linear Approximation of the Lower Workload Curve

The approximation of the lower workload curve is analog to the approximation of the upper workload curve.

Definition 3. *The linear approximation of the lower workload curve for a chosen* $x \geq 0$ *is defined as:*

$$\tilde{\gamma}^l(e) := g^l \cdot e + d^l = \frac{\gamma^l(x)}{x} \cdot e + d^l$$

The approximation is constructed by selecting a point x of the workload curve and computing the gradient $g^l = \gamma^l(x)/x$. After that, d^l is chosen such that the approximation is a lower bound for the lower workload curve within $[0, x)$.

$$d^l = \min_{0 \leq v < x} \left\{ \gamma^l(v) - g^l \cdot v \right\} \tag{3.10}$$

Analog to lemma 2 follows by construction:

Lemma 3. $\tilde{\gamma}^l(e)$ *is a valid bound for* $\gamma^l(e)$ *for all* $e \in [0, x)$*:*

$$\tilde{\gamma}^l(e) \leq \gamma^l(e) \qquad \forall e \in [0, x)$$

And finally the approximation:

Theorem 2. *The approximation* $\tilde{\gamma}^l(\cdot)$ *is a valid lower bound for the lower workload curve* $\gamma^l(\cdot)$

$$\tilde{\gamma}^l(e) \leq \gamma^l(e) \quad \forall x \geq 0$$

Proof.

$$\text{for } e = k \cdot x + v' \qquad k = \left\lfloor \frac{e}{x} \right\rfloor, v' \in [0, x)$$

$$\begin{aligned}
\tilde{\gamma}^l(e) &= \tilde{\gamma}^l(k \cdot x + v') && \text{(definition 3)} \\
&= \frac{\gamma^l(x)}{x} \cdot (k \cdot x + v') + d^u \\
&= \frac{\gamma^l(x)}{x} \cdot k \cdot x + \frac{\gamma^l(x)}{x} \cdot v' + d^u && \text{(definition 3)} \\
&= k \cdot \gamma^l(x) + \tilde{\gamma}^l(v') && \text{(lemma 3)} \\
&\leq k \cdot \gamma^l(x) + \gamma^l(v') && \text{(lemma 1)} \\
&\leq \gamma^l(k \cdot x) + \gamma^l(v') && \text{(superadditivity)} \\
&\leq \gamma^l(k \cdot x + v') \\
&= \gamma^l(e) && \square
\end{aligned}$$

3.4.3 Final Approximation

The introduced linear approximations $\tilde{\gamma}^u(\cdot)$ and $\tilde{\gamma}^l(\cdot)$ offer reasonable bounds for larger numbers of consecutive events. For very few consecutive events, the exact workload curves are much tighter than the linear approximation. Considering that, a combination of both is chosen as the final approximation of the workload curves. For our approximation, the exact computed workload curve is used up to a certain point and then extended to infinity with the linear approximation.

Definition 4. *The final approximation of the workload curves are defined as a combination of the exact workload curves and the linear approximations for some chosen $x \geq 0$, $x' \geq 0$ and $x'' \geq 0$.*

$$\bar{\gamma}^u(e) := \begin{cases} \gamma^u(e) & \text{if } 0 \leq e < x \\ \tilde{\gamma}^u(e) = g^u \cdot e + d^u = \frac{\gamma^u(x')}{x'} \cdot e + d^u & \text{if } e \geq x \end{cases} \tag{3.11}$$

$$\bar{\gamma}^l(e) := \begin{cases} \gamma^l(e) & \text{if } 0 \leq e < x \\ \tilde{\gamma}^l(e) = g^l \cdot e + d^l = \frac{\gamma^u(x'')}{x''} \cdot e + d^l & \text{if } e \geq x \end{cases} \tag{3.12}$$

3.5 Implementation

The computation of workload curves based on an event stream is implemented in the Matlab script *workloadcurve.m.* This script takes an array with the loads of the events in the event stream, and two integers, *start_approx* and *limit* as input.

The parameter *start_approx* corresponds to the value x in the defined approximation (4). It marks the point from where on the linear approximation will be used in the resulting workload curves instead of the computed staircase workload curve.

This parameter addresses the trade off between the complexity and the tightness of the computed workload curves. The run time to evaluate an MPA model will depend on it.

The parameter *limit* corresponds to the maximum value of the three parameters x, x' and x'' in the approximation (4). It marks how far the exact workload curve will be computed in order to search for the best linear approximation. x' and x'' are chosen such that the gradient g^u is minimized and the gradient g^l is maximized to get the optimal linear approximation for large numbers of consecutive events.

The fraction *limit* divided by the number of events in the event stream corresponds to the number of independent observations of *limit* consecutive events in the event stream. This fraction should be high enough to get reliable workload curves.

If the last parameter is omitted, the same value as for the second parameter is used such that the workload curves will be computed exactly to the point where the approximation starts.

The function returns the upper and lower workload curves as two strings, which evaluate each to an rtccurve, as defined in the RTC toolbox. Further the four parameters of the two approximations (d^u, g^u, d^l, g^l) are returned, too.

3.6 Example

As an example, the workload curve of the source process of the example process network introduced in the last chapter in Fig. 2.3 is shown.

In this process network, containing five processes mapped to three processors, data is generated at the source, squared in each calculation process and consumed in the sink.

Per event, four floats or 16 bytes are processed and transmitted over the channels. 1000 events were processed in the simulation. The first 20 and last 20 are not taken into account to compute the workload curves, to remove the effect of the start and end phase of the execution.

In Fig. 3.2, the resulting workload curves for the source process is shown. The curve just using best-case / worst-case is shown in red, two approximations using once 4 steps of the workload curve and once 8 in green and blue as well as the full workload curve in black.

3.7 Arrival Curves

Incoming event streams are often characterised by arrival curves [4]. From the simulation, the timestamps for each start or end fire, read or write is extracted in a separate text file. It would be useful if the arrival curves for the different tasks could be computed from such a list of timestamps observed in a simulation.

It turns out that arrival curves can be obtained from a list of timestamps using the same computation as for the workload curves. Only two parts of the computation have to be changed. First the list of timestamps has to be converted in a list of time difference. Then the function workloadcurve.m can be called to get the curves. At last, the computed curves have to be inverted to get the arrival curves. The inversion is necessary, because arrival curves measure events per cycles and workload curves measure cycles per event.

The computation of arrival curves based on a list of timestamps is implemented in the script arrivalcurve.m.

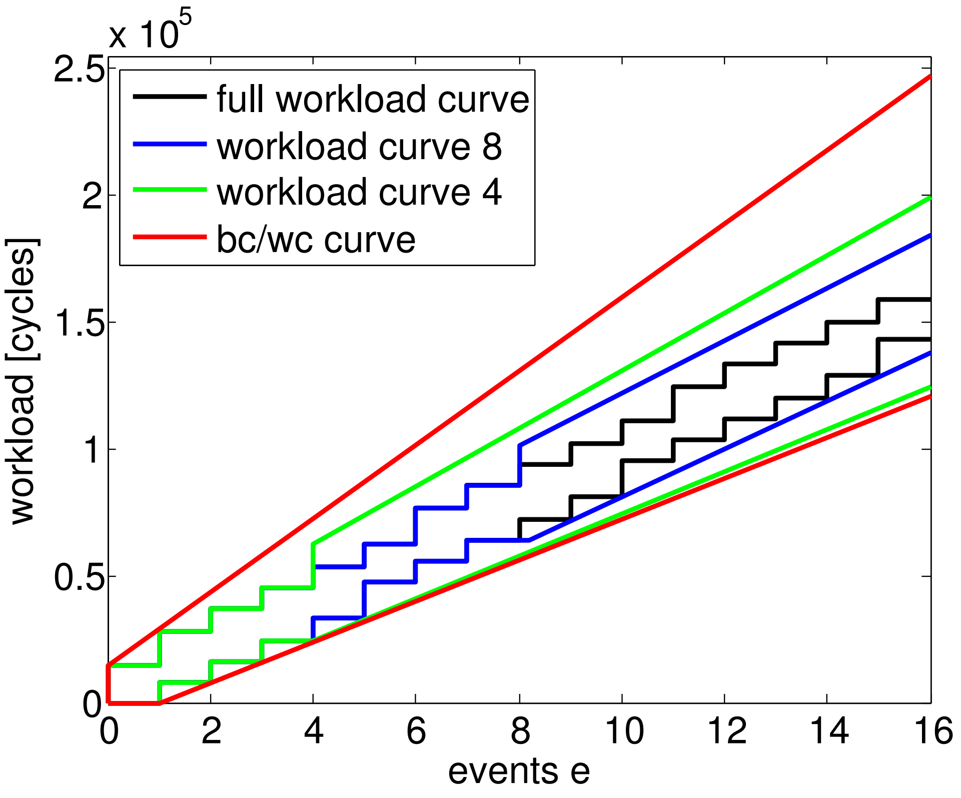


Figure 3.2: Example workload curves.

Chapter 4

Conclusion

4.1 Testbed

For any DOL application mapped to the MPARM architecture, detailed results about the execution can be obtained within minutes. The DOL specification can be executed with the implemented testbed in a cycle-accurate MPARM simulator.

The user gets back an execution trace that allows detailed insight into the execution of the application on the MPARM architecture without the need to adjust anything in the application code.

Further, a basic modular performance analysis model is created from the application specification and the results from the simulation.

4.2 Workload Curves

It is shown in this work how workload curves can be computed from the execution demand of a number of consecutive events. The introduced approximation offers a way to extend the workload curves to infinity and to limit the complexity of the computed workload curves.

The computation is implemented in the Matlab scripts workloadcurve.m and arrivalcurve.m. Further, the greedy processing component (GPC) of the RTC toolbox [5] was adapted to work with workload curves.

4.3 Future Work

The generated MPA model should be verified and compared to the results from simulation in future work using a reasonable real-time application as a test case.

More system properties like delays of the events or the utilization of the processors could be extracted from the simulation results in order to compare these with the analytic results from the MPA.

Further, a graphical user interface, for example a web front end, could be implemented and the testbed be set up on a server machine to offer easy access to the testbed.

Appendix A

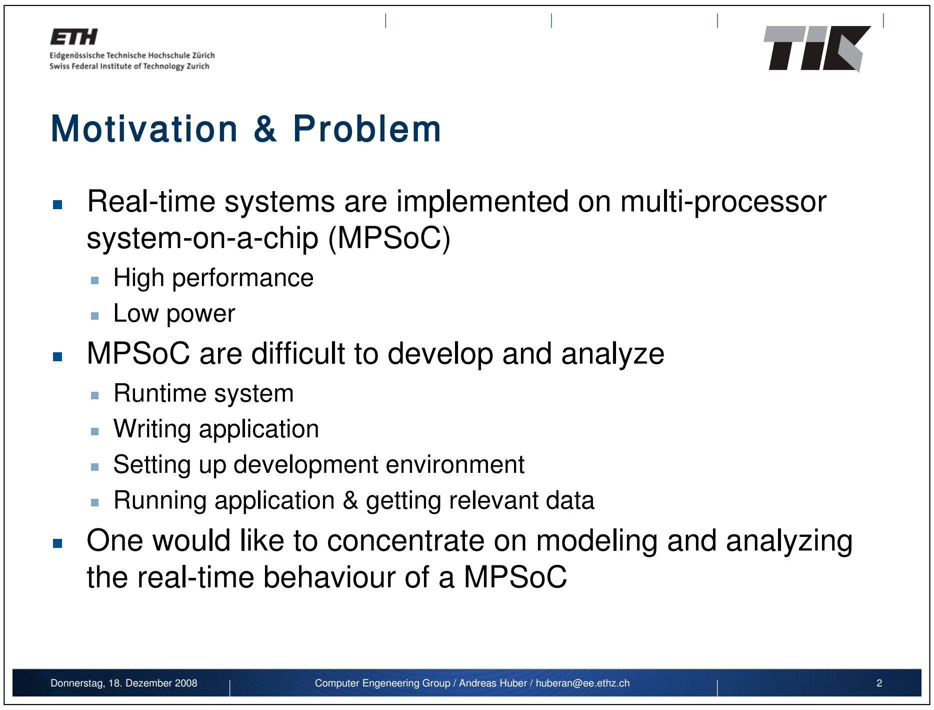
Presentation Slides

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

MPSoC Testbed for Analyzing Execution of Process Networks

Andreas Huber

Advisors:	Wolfgang Haid, Kai Huang
Professor:	Prof. Dr. Lothar Thiele

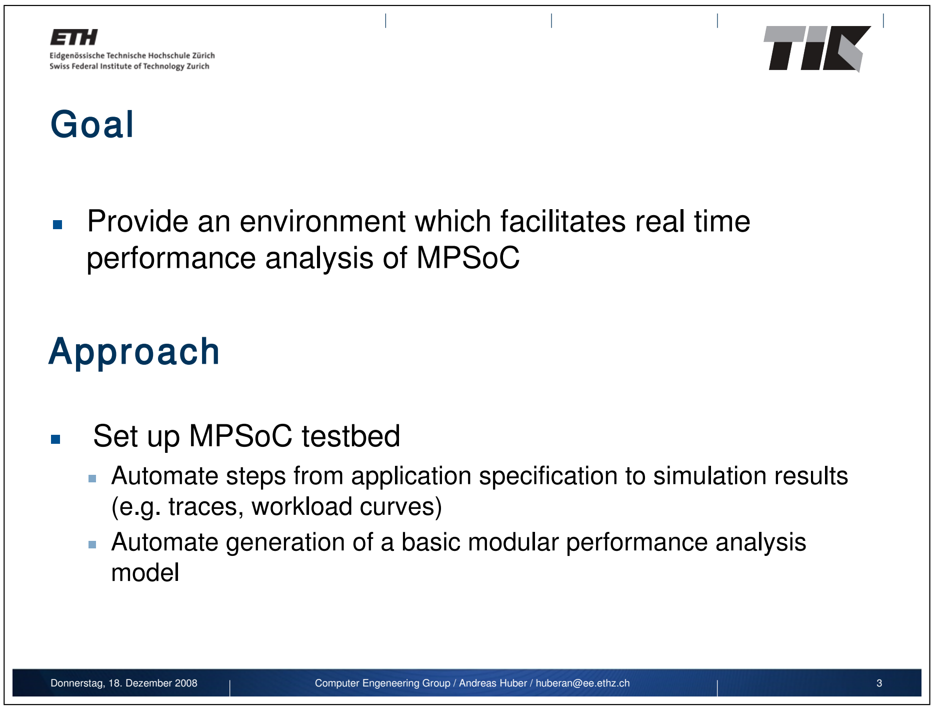


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Motivation & Problem

- Real-time systems are implemented on multi-processor system-on-a-chip (MPSoC)
  - High performance
  - Low power
- MPSoC are difficult to develop and analyze
  - Runtime system
  - Writing application
  - Setting up development environment
  - Running application & getting relevant data
- One would like to concentrate on modeling and analyzing the real-time behaviour of a MPSoC

Donnerstag, 18. Dezember 2008 | Computer Engeneering Group / Andreas Huber / huberan@ee.ethz.ch | 2



ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

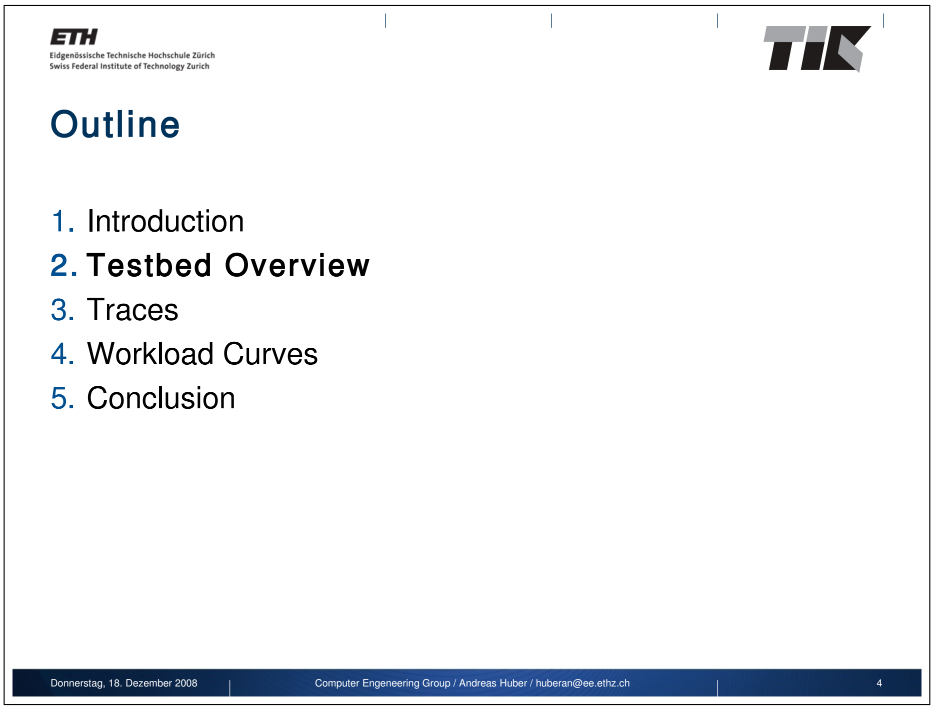
Goal

- Provide an environment which facilitates real time performance analysis of MPSoC

Approach

- Set up MPSoC testbed
  - Automate steps from application specification to simulation results (e.g. traces, workload curves)
  - Automate generation of a basic modular performance analysis model

Donnerstag, 18. Dezember 2008 | Computer Engeneering Group / Andreas Huber / huberan@ee.ethz.ch | 3

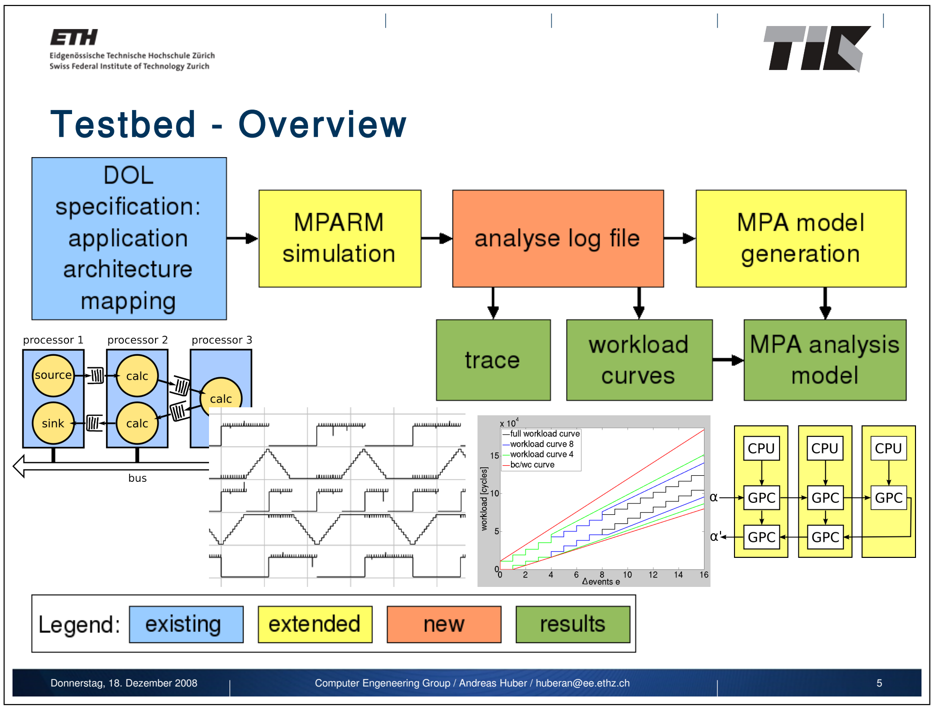


ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Outline

1. Introduction
2. **Testbed Overview**
3. Traces
4. Workload Curves
5. Conclusion

Donnerstag, 18. Dezember 2008 | Computer Engeneering Group / Andreas Huber / huberan@ee.ethz.ch | 4

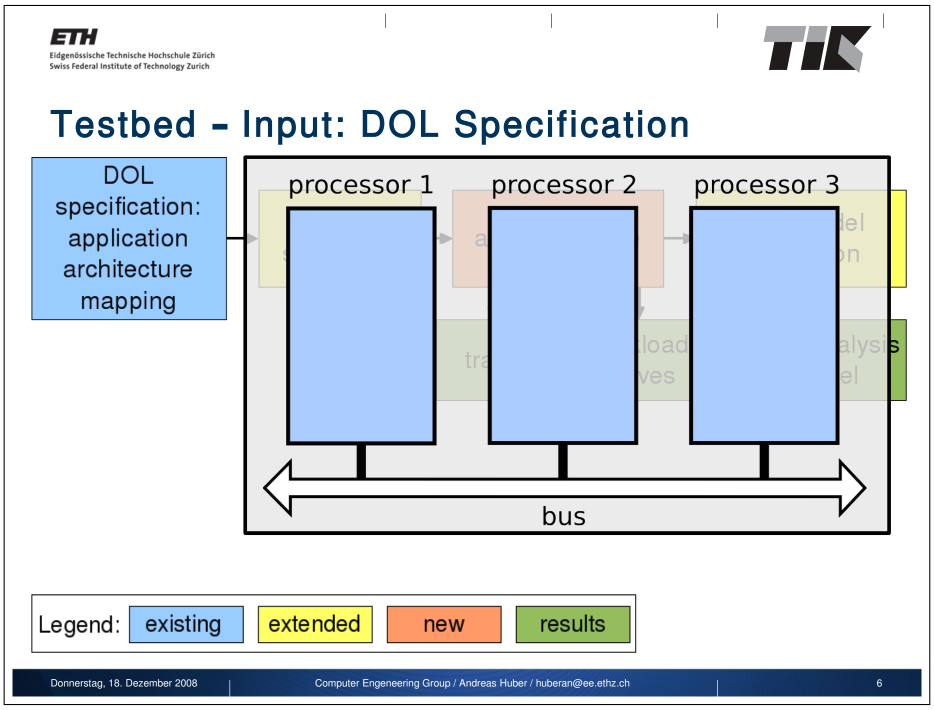


ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Testbed - Overview

Legend: existing | extended | new | results

Donnerstag, 18. Dezember 2008 | Computer Engeneering Group / Andreas Huber / huberan@ee.ethz.ch | 5

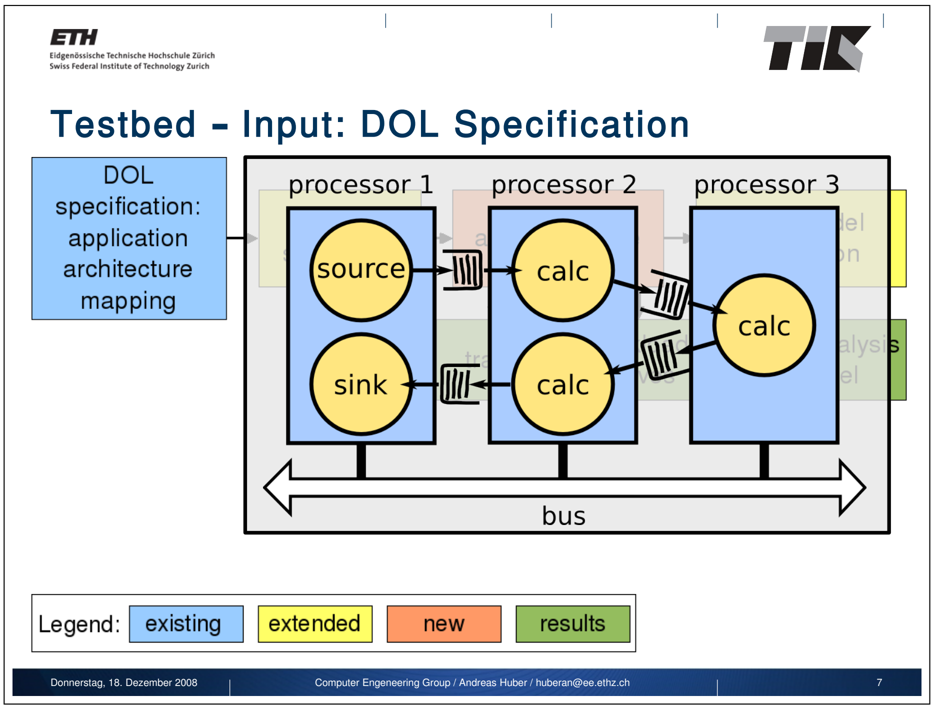


Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Testbed – Input: DOL Specification

Legend: existing | extended | new | results

Donnerstag, 18. Dezember 2008 | Computer Engeneering Group / Andreas Huber / huberan@ee.ethz.ch | 6



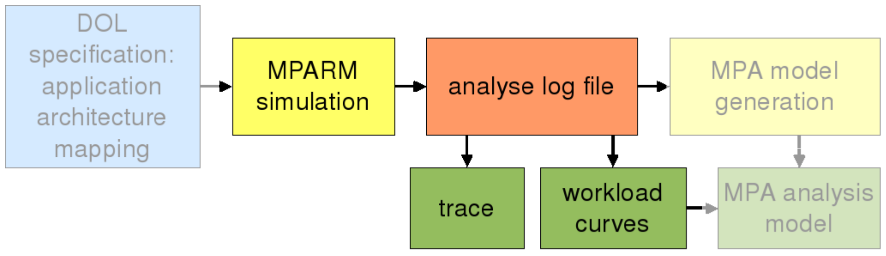
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Testbed – Input: DOL Specification

Legend: existing | extended | new | results

Donnerstag, 18. Dezember 2008 | Computer Engeneering Group / Andreas Huber / huberan@ee.ethz.ch | 7

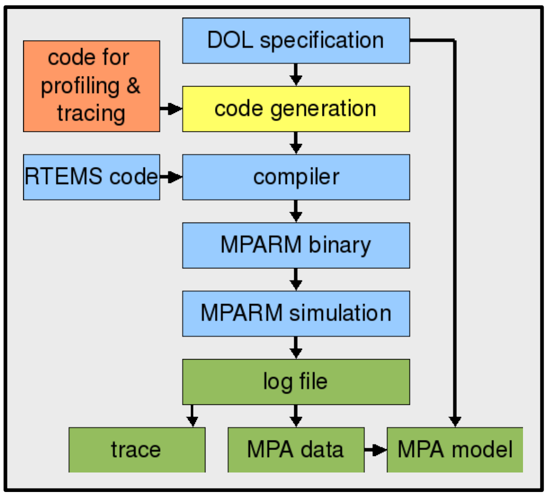
Testbed – MPARM Simulation

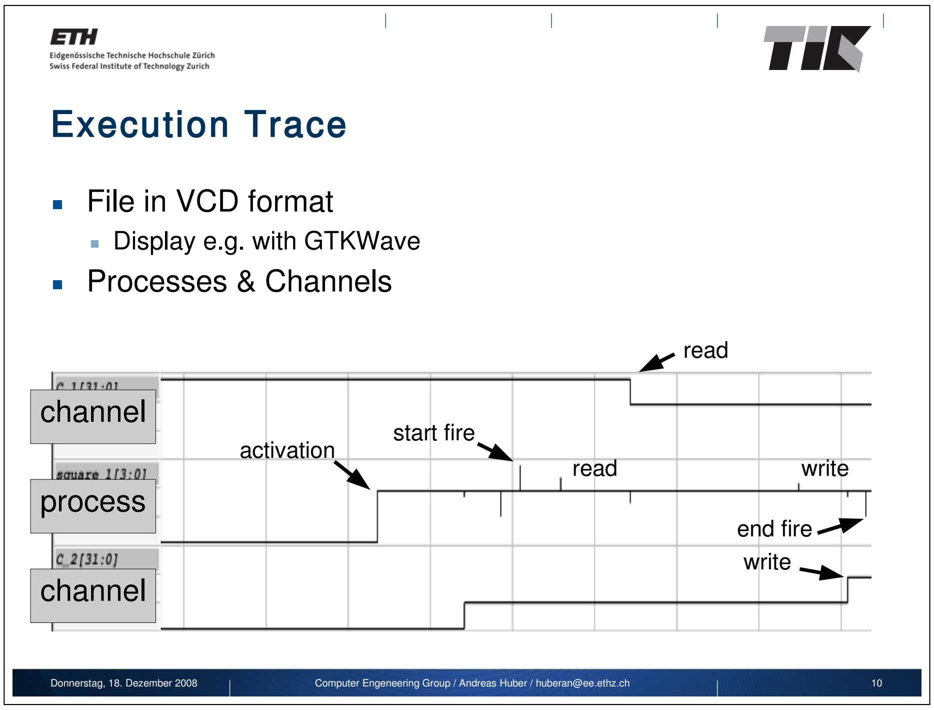


Legend: existing | extended | new | results

Testbed – MPARM Simulation

- Determine running process
  - Callback on context switches
- Keep influence of logging as small as possible
  - Lightweight print in ASM for logging of timestamps
  - Printf: 5000 cycles
  - ASM: 50 cycles

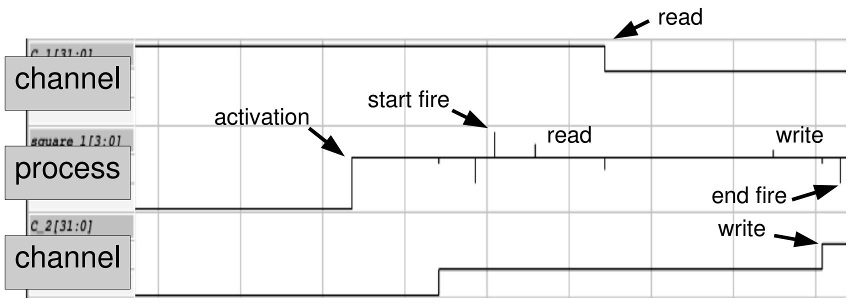




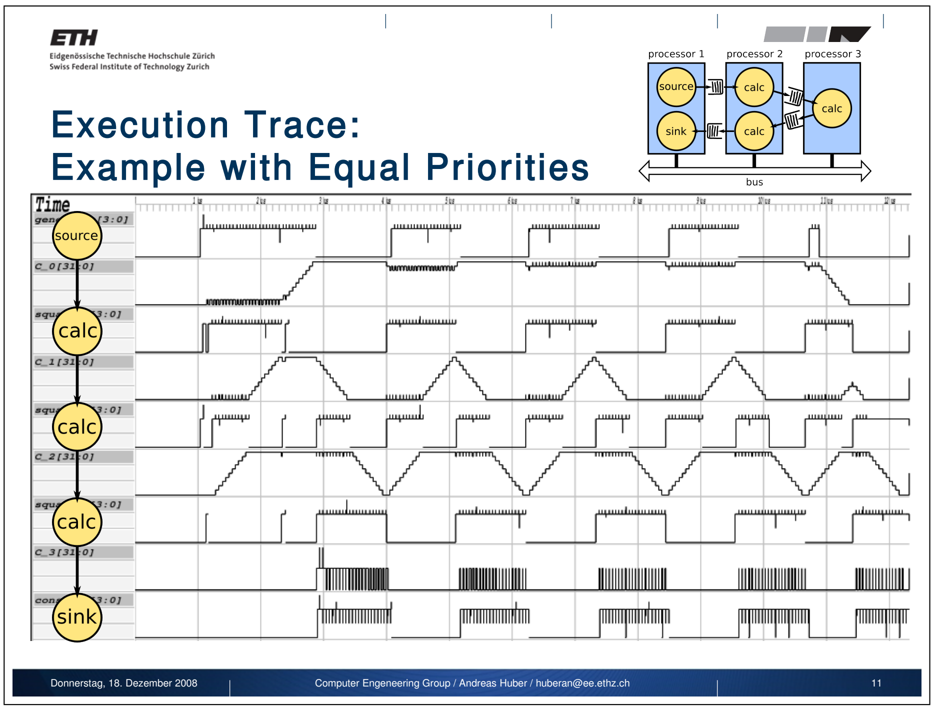
ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Execution Trace

- File in VCD format
  - Display e.g. with GTKWave
- Processes & Channels

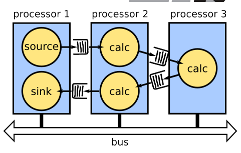


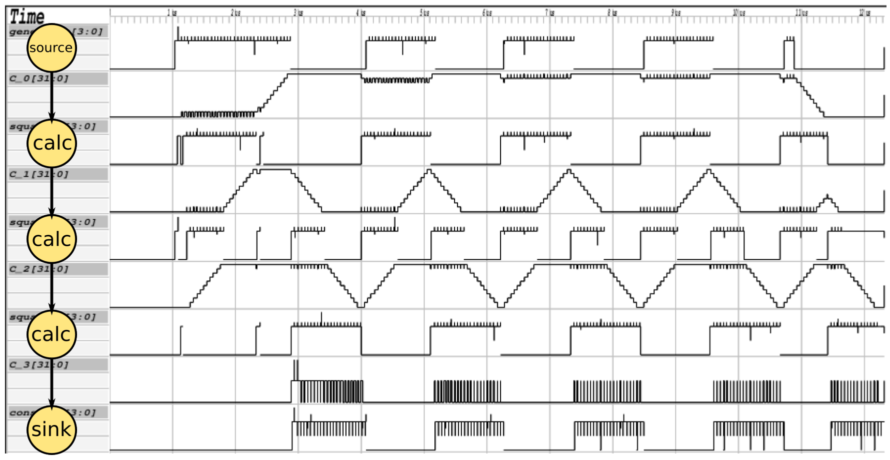
Donnerstag, 18. Dezember 2008 | Computer Engeneering Group / Andreas Huber / huberan@ee.ethz.ch | 10



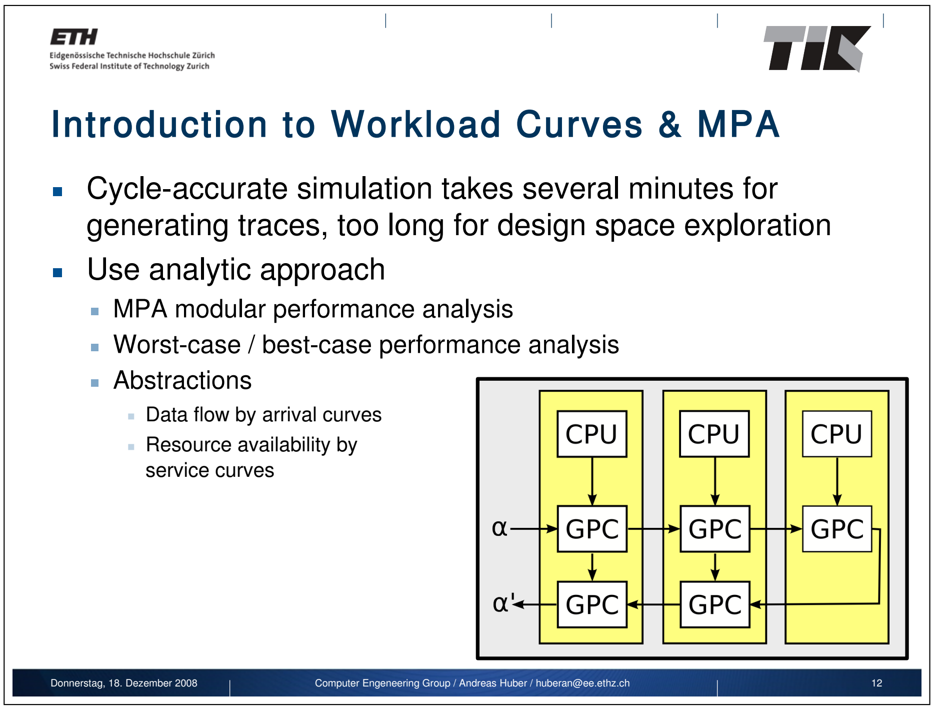
ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Execution Trace: Example with Equal Priorities





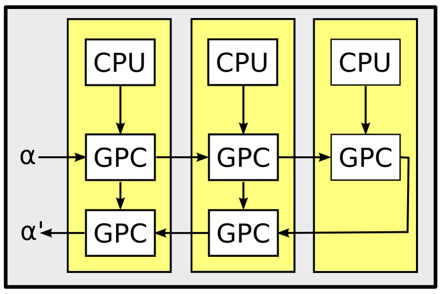
Donnerstag, 18. Dezember 2008 | Computer Engeneering Group / Andreas Huber / huberan@ee.ethz.ch | 11



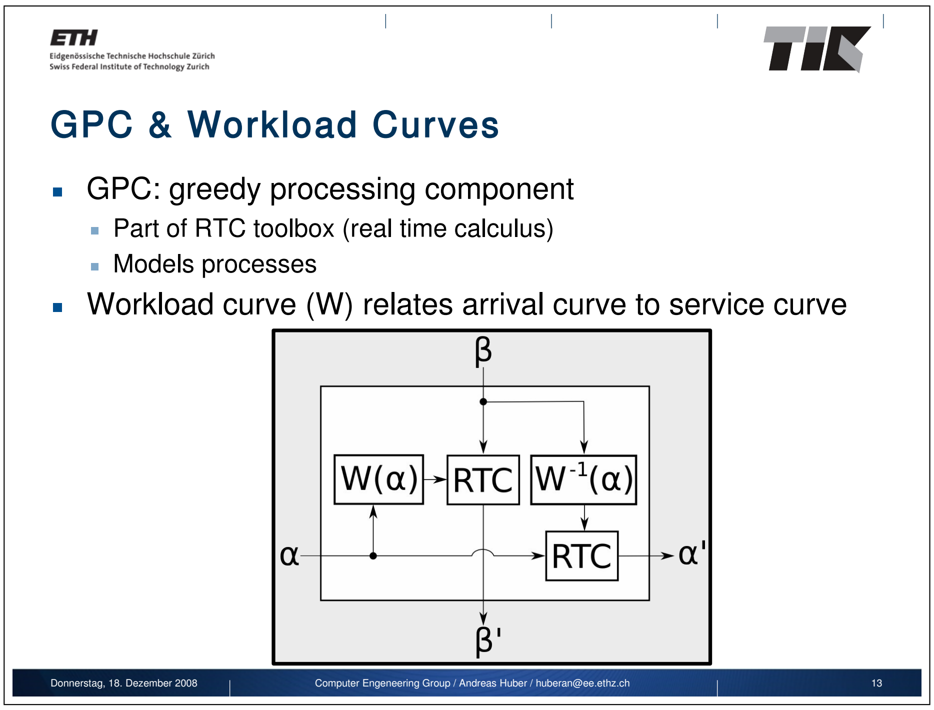
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Introduction to Workload Curves & MPA

- Cycle-accurate simulation takes several minutes for generating traces, too long for design space exploration
- Use analytic approach
  - MPA modular performance analysis
  - Worst-case / best-case performance analysis
  - Abstractions
    - Data flow by arrival curves
    - Resource availability by service curves



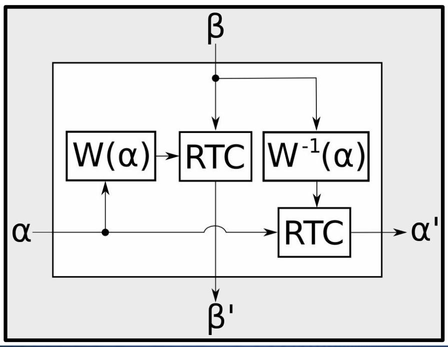
Donnerstag, 18. Dezember 2008 | Computer Engeneering Group / Andreas Huber / huberan@ee.ethz.ch | 12



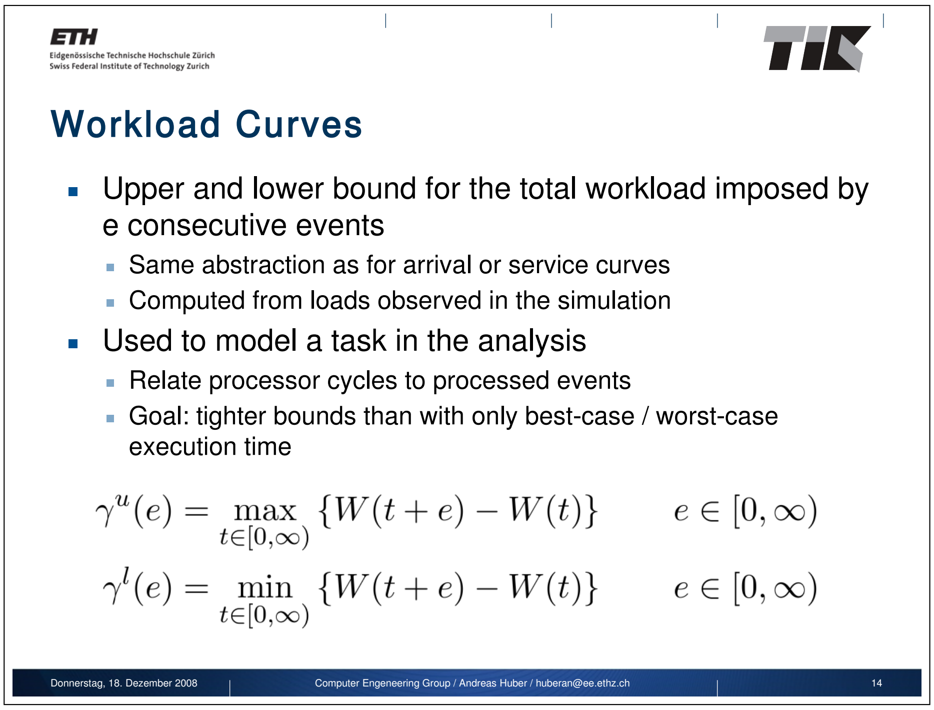
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

GPC & Workload Curves

- GPC: greedy processing component
  - Part of RTC toolbox (real time calculus)
  - Models processes
- Workload curve (W) relates arrival curve to service curve



Donnerstag, 18. Dezember 2008 | Computer Engeneering Group / Andreas Huber / huberan@ee.ethz.ch | 13



ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

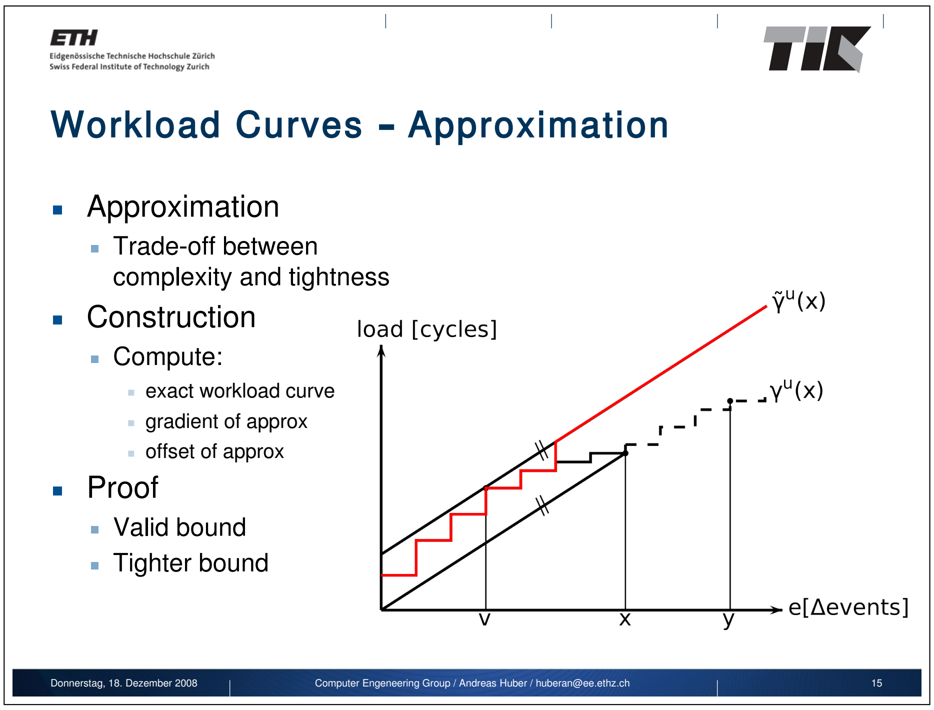
Workload Curves

- Upper and lower bound for the total workload imposed by e consecutive events
  - Same abstraction as for arrival or service curves
  - Computed from loads observed in the simulation
- Used to model a task in the analysis
  - Relate processor cycles to processed events
  - Goal: tighter bounds than with only best-case / worst-case execution time

$$\gamma^u(e) = \max_{t\in[0,\infty)} \{W(t+e) - W(t)\} \qquad e \in [0,\infty)$$

$$\gamma^l(e) = \min_{t\in[0,\infty)} \{W(t+e) - W(t)\} \qquad e \in [0,\infty)$$

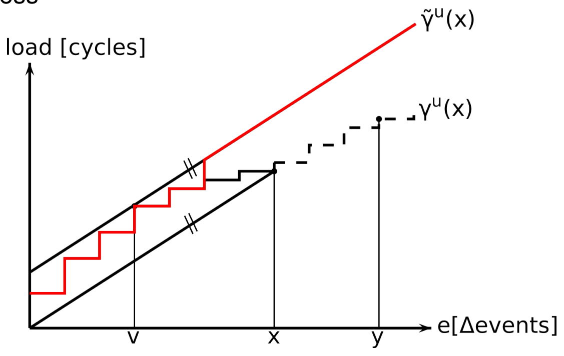
Donnerstag, 18. Dezember 2008 | Computer Engeneering Group / Andreas Huber / huberan@ee.ethz.ch | 14



ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Workload Curves – Approximation

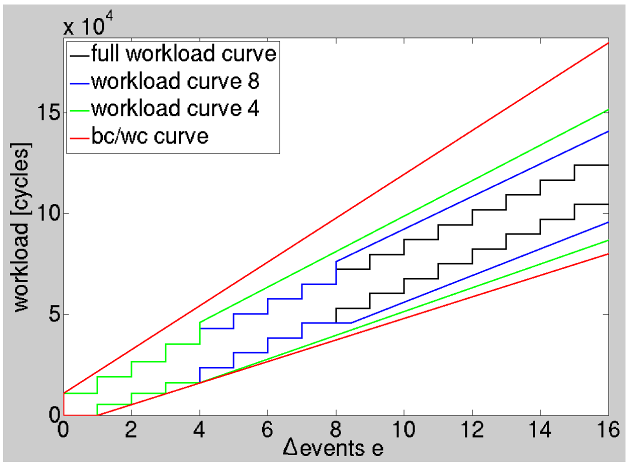
- Approximation
  - Trade-off between complexity and tightness
- Construction
  - Compute:
    - exact workload curve
    - gradient of approx
    - offset of approx
- Proof
  - Valid bound
  - Tighter bound



Donnerstag, 18. Dezember 2008 | Computer Engeneering Group / Andreas Huber / huberan@ee.ethz.ch | 15

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Workload curves - Example



MPA model with best-case/worst-case:

Delay: 483 µs

Backlog: 40 kByte

MPA model with workloadcurve (8):

Delay: 383 µs

Backlog: 36 kByte

Donnerstag, 18. Dezember 2008 | Computer Engeneering Group / Andreas Huber / huberan@ee.ethz.ch | 16

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Conclusions & Future Work

- Testbed
  - For any DOL application mapped to the MPARM architecture, detailed results about the execution can be obtained within minutes.
  - Traces allow insight into the execution of an application in the MPARM simulator.
  - A basic MPA analysis model is created from the application and the results from the simulation.
- Future work
  - Compare MPA model with results from simulation

Donnerstag, 18. Dezember 2008 | Computer Engeneering Group / Andreas Huber / huberan@ee.ethz.ch | 17

Appendix B

Testbed Readme

The paths set in the beginning of the testbed script (testbed/build.xml) have to be modfied in order to run the script on a different machine. This paths point to the different project libraries that were used in the testbed. These are in particular:

- dol
- analysismodel
- perfanalysis
- mparm
- mparmlog
- matlab

The input sources can be specified when the testbed is run with the attribute *-Dsource=/path/to/source/dir*. The specified source folder has to contain the three files *process_network.xml*, *architecture.xml* and *mapping.xml* as well as a folder named *src* containing the source files for the processes.

When the testbed is executed, the directories source, build, log and result are created in the directory where the script is located.

Following results are generated and stored in the result directory when executing the testbed:

- trace1.vcd
- trace2.vcd
- process_network_flattened_annotated_workload.xml
- dolanalysismodel.xml
- dolanalysismodel.m
- dolanalysismodel_iwl.m
- log/log.txt
- log/log.parsed.sorted.txt

And for each process:

- log/*processname*_load.txt
- log/*processname*_start_fire.txt
- log/*processname*_end_fire.txt
- log/*processname*_start_read.txt
- log/*processname*_end_read.txt
- log/*processname*_start_write.txt
- log/*processname*_end_write.txt.

If the testbed is executed with the attribute *-Dresult=/path/to/result/dir* set, these results are copied to the specified folder.

Appendix C

Created and Modified Code Files

Here follows an overview of the code files that were created or modified in this thesis.

C.1 Testbed

The testbed was written as an Apache Ant script.

- testbed/build.xml

C.2 DOL Framework

The code files in the package dol.visitor.rtems of the DOL framework were modified to include the necessary log statements in the code generation for the MPARM platform:

- dol/visitor/rtems/RtemsMakefileVisitor.java
- dol/visitor/rtems/RtemsModuleVisitor.java:
- dol/visitor/rtems/RtemsPropertiesVisitor.java
- dol/visitor/rtems/RtemsVisitor.java
- dol/visitor/rtems/lib/process_wrapper_template.c
- dol/visitor/rtems/lib/rtems_process_wrapper.c
- dol/parser/xml/mapschema/Xml2Map.java

C.3 Log Parser and Analyser

The application to parse and analyse the log file from the simulation was written in this work, using Java. The created files are located in the package mparmlog.

- Parser.java
- Analyser.java
- LogException.java
- ValueChangeDump.java
- ValueChangeDumpException.java
- Profiler.java

C.4 MPA Model

The analysis model was adapted to work with workload curves. And the used visitors were adapted.

The following files in the analysismodel project were modified:

- analysismodel/UserInterface.java
- analysismodel/datamodel/analysis/pn/Task.java
- analysismodel/datamodel/mpa/MpaCommand.java
- analysismodel/datamodel/mpa/MpaGpc.java
- analysismodel/util/AnalysisModelParser.java
- analysismodel/visitor/AnalysisModelMpaVisitor.java
- analysismodel/visitor/AnalysisModelXmlVisitor.java
- analysismodel/visitor/MpaDataMatlabVisitor.java
- schema/analysismodel.xsd

The following files in the perfanalysis project were modified:

- perfanalysis/ActionControl.java
- perfanalysis/util/DolConverterMparm.java

C.5 Matlab Scripts

The computation of workload curves from a list of task workloads and the computation of arrival curves from a list of timestamps were implemented. The computation for the greedy processing component was adapted to work with workload curves.

- workloadcurve.m
- arrivalcurve.m
- rtcgpcwl.m

Bibliography

[1] The Apache Ant Project. http://ant.apache.org/.

[2] Luca Benini, Davide Bertozzi, Alessandro Bogliolo, Francesco Menichelli, and Mauro Olivieri. MPARM: Exploring the Multi-Processor SoC Design Space with SystemC. *VLSI Signal Processing Systems*, 41(2):169–182, September 2005.

[3] GTKWave. http://sourceforge.net/projects/gtkwave/.

[4] Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus — A Theory of Deterministic Queuing Systems for the Internet*, volume 2050 of *Lecture Notes in Computer Science*. Springer Verlag, 2001.

[5] Modular Performance Analysis and Real-Time Calculus. http://www.mpa.ethz.ch.

[6] RTEMS Steering Committee. RTEMS Home Page. http://www.rtems.com.

[7] The Open SystemC Initiative (OSCI). http://www.systemc.org.

[8] Lothar Thiele, Iuliana Bacivarov, Wolfgang Haid, and Kai Huang. Mapping Applications to Tiled Multiprocessor Embedded Systems. In *Int'l Conf. on Application of Concurrency to System Design (ACSD)*, pages 29–40, Bratislava, Slovak Republic, July 2007.

[9] IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language.

[10] Ernesto Wandeler, Alexander Maxiaguine, and Lothar Thiele. Quantitative Characterization of Event Streams in Analysis of Hard Real-Time Applications. *Real-Time Systems*, 29(2):205–225, March 2005.