ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Semester Thesis

Code Refactoring and Memory Optimization for the Trace-based Simulation Framework

Daniel Matter

Supervisors: Dr. Iuliana Bacivarov, Kai Huang
Professor: Prof. Dr. Lothar Thiele
Computer Engineering Group

Semester Thesis SA-2007-63
September 2007 to February 2008

Abstract

The trace-based simulation is a useful tool to simulate hardware-software codesign at system-level for best trade-off between accuracy and speed. The existing trace-based simulation developed for the DOL framework has some disadvantages concerning the extendability with new schedulers, the usability in terms of controlling the simulation and the high demand of memory capacity.

This thesis proposes design modification and presents their implementation to overcome these problems. Moving the design towards a more object-oriented style simplifies the task of adding a new scheduler by providing a common interface for all concrete schedulers and by rendering unnecessary macros. The usability is improved by an XML configuration file which can control the simulation. The memory optimization is achieved by two design changes: The postponing of the trace decomposition from the initialization phase to the simulation phase and the separation of the shareable and non-sharable information of the trace events, namely the flyweight design pattern.

The main achievement of the memory optimization is that the memory consumption is independent of the accuracy of the simulation and that the memory usage is reduced to at least 25% while the simulation speed could be maintained.

Contents

List of Figures

Listings

Chapter 1

Introduction

This thesis describes code modification proposals for the trace-based simulation framework.

1.1 Context

Nowadays, the trend of system design is moving from single processor architecture toward multiprocessor System-on-Chip (MpSoC) architecture. This shift can not only be noticed in the newer generation of general-purpose processors like the quad core processors of Intel, but also in the area of the embedded systems the heterogeneous multiprocessor systems are required by the ever increasing complexity of the embedded applications. Although these MpSoCs offer high scale integration and high computing power, the challenge is to find a scalable HW/SW design style to alleviate the complex design procedure. The SHAPES [1] (scalable software hardware architecture platform for embedded systems) project proposes a tiled architecture as well as complete solution for the programming of such architecture.

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 HW/SW cosimulation, be it at register-transfer level, cycle accurate, or instruction accurate level, is too slow to be included in an iterative performance optimization process. Therefore, performance evaluation at a higher abstraction level, for example at the system level, is necessary to minimize the modeling effort and to get the best trade-off between accuracy and speed.

A trace-based simulation framework has been newly developed for the DOL environment, which is a part of the framework of SHAPES European project. The DOL aims at reducing significantly the effort of mapping applications onto a SPAPES multiprocessor hardware platform under certain mapping constraints [2].

The trace-based simulation framework aims at solving the problem for performance evaluation at system level. It takes application specification, architecture specification and mapping specification as input and provides system performance statistics, such as processor load, bus load, maximal backlogs and overall system execution time. They can be used to guide the designers to improve the design of the system.

1.1.1 Traced-based Simulation Framework for Multiprocessor Systems

The traced-based simulation achieves a speed-up with respect to cycle-accurate simulation by abstracting the computation behavior as high-level execution traces. Therefore, to simulate the computation behavior effects on the target architecture, one does 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 diagram depicted in Figure 1.1 reveals that the framework consist of two parts:

- The first part is the modified functional simulation of the DOL which is responsible to generate the execution traces. The role played by the functional simulation is depicted in Figure 3.3 of [4]. How the functional simulation is modified to generate the trace is presented in section 4.2 of [4]. The format of the trace can be found in this section as well. Important to know is that the trace consist of three different trace event types: Computation events, read events and write events.

- The second part is the core of the trace-based simulation which can evaluate different mapping and architecture specifications. The working model of the trace-based simulation is outlined in section 3.1 of [4] and the output of the simulation is the topic of subsection “Simulation Settings”.

The framework takes as input the application specification, architecture specification and mapping specification (see Fig. 1.1). Details for the specifications can be retrieved from sections 4.1.1 - 4.1.4 in [4].

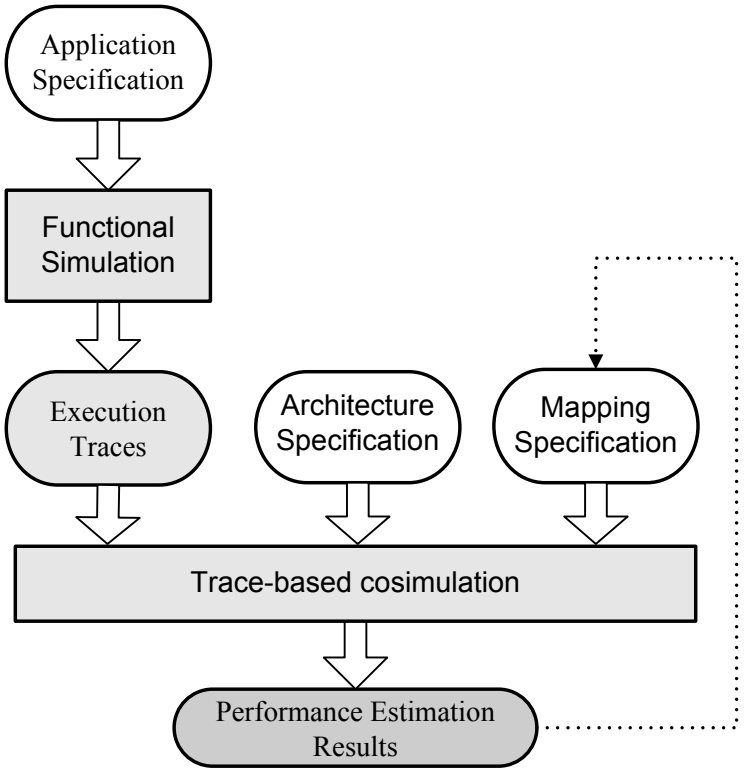


Figure 1.1: Simulation methodology[4]

Trace Transformation

An important issue is the trace transformation or decomposing which concerns the communication events (read and write trace events). In the process of trace transformation, each communication events containing more data than the atomic data size will be divided into smaller trace events which do not exceed the atomic data size. (see Fig. 3.6 of [4]). The atomic data size defines the granularity level of the simulation and can be defined by the user. The flexibility of the atomic data size provides the designer with a trade-off between accuracy and efficiency.

The introduction of the atomic data size allows to reflect the actual behavior of the hardware component, e.g the bus width or the fixed packet length of a device driver. The bus width is then also considered in the scheduler because date with the length of the bus width cannot be preempted.

Simulation Settings

The trace-based simulation provides two different simulation settings, which results in two different outputs, that we will call simulation modes. One is the “Performance Statistic” feature and the other is the “VCD Tracing”. Both modes allow to evaluate the mapping and architecture specification by tracking the simulation data.

The performance statistic mode keeps account of the read, write, computation and total execution time of each processor, the processing time on each bus and the computation, read, write and processing time on buses of each process. Moreover, information about the maximal backlogs of the software channels and of the ready queues of the buses are logged. After the simulation, all figures mentioned before, as well as the maximum values of workload on resources and the estimated execution time are prompted to the shell and stored in a file. An example of the shell output can be seen in Figure 1.2.

```
Performance statistics:
-----------------------------------------
C1 max backlog: 8
C2 max backlog: 8
++++ Bus: in_tile_link ++++
queue processor1 max backlog: 8
queue processor2 max backlog: 16
process: generator
computation(NS): 800| Read: 0| Write: 400| Sum: 1200
processing time on buses (NS): 400
--------------------------
process: consumer
computation(NS): 820| Read: 400| Write: 0| Sum: 1220
processing time on buses (NS): 400
--------------------------
process: square
computation(NS): 1220| Read: 400| Write: 400| Sum: 2020
processing time on buses (NS): 800
--------------------------
processor: processor1
computation time (NS): 1220
|Read: 400 |Write: 400 |Sum: 2020
total execution time including idle (NS): 2480
-----------------------------------------
processor: processor2
computation time (NS): 1620
|Read: 400 |Write: 400 |Sum: 2420
total execution time including idle (NS): 2540
-----------------------------------------
bus: in_tile_link
processing time (NS): 1600
-----------------------------------------
Estimated execution time (NS): 2540
-----------------------------------------
Max computation time of processors (NS): 1620
-----------------------------------------
Max processing time of buses (NS): 1600
-----------------------------------------
Time used for trace-based simulation (s): 0
```

Figure 1.2: Performance statistics

The VCD tracing builds a file with wave forms according to the Verilog VCD format. These waves represents the activities of the processes and hardware resources in the temporal domain. Figure 1.3 shows the waves of a simple application example.

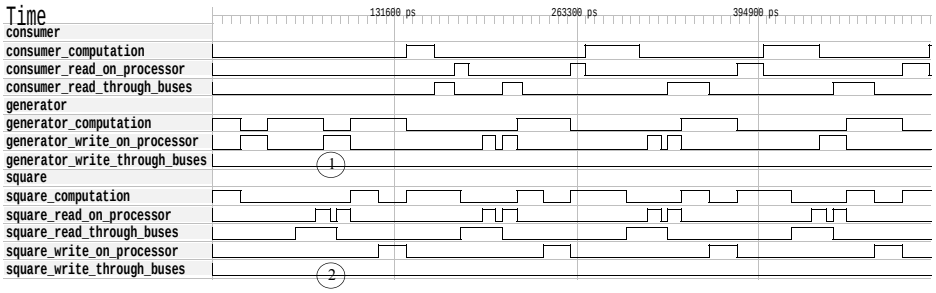


Figure 1.3: VCD trace [4]

1.2 Motivation

A trace-based simulation framework should fulfil several conditions. The system level performance evaluation for an application mapped onto a multiprocessor embedded system architecture, should be efficient in terms of time and memory usage, and at the same time maintain sufficient accuracy for making design decisions.

Furthermore, a modular framework allows for making extensions in a fast way. Implementing a new scheduler, for example, should not involve the task of adding code at several classes or files.

Additionally, an efficient use of the framework asks for the possibility of simulating autonomously various mapping and architecture specifications. After running through all predefined specifications, the output of the performance data should be well structured and maintain predicative information for each mapping.

1.3 Task Description

The first step of the practical work described in this thesis is to review the existing codes of the trace-based simulation framework. In a second step, suggestions for optimizations and updates should be made. A designer may discover the following points during testing the simulation framework:

- Adding a new scheduler, like a static scheduler for example, requires explicit extensions distributed allover the code. Pieces of code can be copied from other implementations, but in a more sophisticated design, this should be avoided.
- The usability in terms of switching between simulation modes or other parameters i.e atomic data size, is cumbersome. A new code compilation is required for every setup.
- The simulation can only be controlled by hand by passing parameters over the command line. The circumstantial work flow is described below.
    1. The example number wished to simulate has to be feed. Afterwards, the trace for the example is loaded.
    2. The user can decide between the option “exit” or “continue”.
    3. In case of selecting “continue”, the mapping specification can be entered by the user.
    4. After the finish of the simulation, it jumps back to step 2.
- The memory overruns during loading the trace. This problem emerges by trying to simulate a large trace or by using a small atomic data size.

This thesis proposes and implements optimizations for the existing trace-based simulation framework with respect to memory usage and refactores the trace-based simulation library towards a more object-oriented style to solve these problems.

1.4 Overview

In Chapter 2, the design modification towards a more user-friendly and modular framework are proposed. Moreover, the approach to reduce the memory usage is presented. Chapter 3 provides detailed information about the implementation of the modifications stated in the previous section. The evaluation result of the modified framework are compared with the original framework and presented in chapter 4. In the last chapter, we give the conclusions of this thesis.

Chapter 2

Design

This chapter reviews the code of the existing framework and discusses refactoring proposals which bring improvements in terms of memory usage, performance, modularity and usability. The first section presents the modifications which lead towards a more user-friendly and modular simulation environment. The second section introduces a design strategy solving the memory problem which might result in a performance improvement as well.

2.1 Code Refactoring

This section suggests some design changes to improve the usability of the framework in terms of simulation control and modular extensions (like schedulers). The first subsection explains how the macros in the code can be avoided. The next subsection states how the scheduler class hierarchy should be organized and the last subsection introduces a configuration file. In each subsection, the existing code is reviewed, the proposal solution and potential impact are presented as well.

2.1.1 Avoidance of Macros

In the current state, a simulation mode can be activated by defining the according macro in the Makefile, for deriving the VCD tracing or performance statistics. As a counterpart in the code, calls to the functions which log the data for the VCD tracing or performance statistics appear as conditional inclusions (see Listing 3.1 for an example). The problem is the code modularity and scalability: For example, in the scheduler classes, these conditional inclusions have to be placed in the code of every scheduler.

A first approach to avoid this was to try moving the macros into simulation mode functions. If the inclusion is not activated, then just empty functions will be called, e.g. the scheduling function of the scheduler classes. But the following problem comes up: The processor, bus and process class have a pointer to an object, which stores the data for the performance statistics. These objects are set in a member function of the performance statistic class. Not running the function which sets the performance log objects would produce an error as soon as the framework accesses these objects. Running this function and instantiate all objects would produce unnecessary overhead. Thus, another solution is needed.

The proposed modification contains two classes. A parent class containing empty functions which do not do anything and a subclass containing functions with the effective implementation. Thereby, the parent class instantiate dummy objects of classes without any member variable but with empty functions. Details about the implementation can be found in the next chapter in section 3.1.2 where also a class diagram (Fig. 3.1) is shown to gain more insight. By applying this modification, we do not only solve the issue regarding the conditional inclusions in the scheduler class. Moreover, this solution has the advantage that the simulation mode can be switched using the same code compilation.

```
...

#ifdef INCLUDE_PERFORMANCE_STATISTICS
        _current_ready_queue_ptr->current_backlog -= data_num;
#endif

...

#ifdef INCLUDE_VCD_TRACE
        vcd_tracing->read_from_buses_start(current_trace_event_ptr);
#endif

        wait(current_trace_event_ptr->data_transfer_time , SC_NS);

#ifdef INCLUDE_VCD_TRACE
        vcd_tracing->read_from_buses_end(current_trace_event_ptr);
#endif

...
```

Listing 2.1: Conditional inclusions in the scheduler class

2.1.2 Reorganisation of the Scheduler Class Hierarchy

In the original version, adding a new scheduler type demands to insert code in several files and classes like the library file and resource classes. In order to facilitate the insertion of a new scheduler in the trace-based simulation framework, we propose the following: The definition of a consistent interface for schedulers and arbiters in order that all the instances of schedulers or arbiters can be accessed in the same way. The insertion of the new scheduler should be done just by coding the new scheduling subclass and adding the corresponding lines in the mapping XML file parser, where the scheduler is declared.

In order to implement these new facilities, the default implementation of the scheduler is coded in a main scheduler class, including parts of the scheduler which do not typically vary. A subclass can then either inherit the function if the standard implementation is needed or override the parent function with the own functionality. Hence, the workload for implementing a new scheduler is reduced further.

The basic concept for this refactoring step was found during developing a static scheduler for the framework. At the moment, we considered how the code should be redesigned in order to allow implementing a new scheduler type in a straightforward manner.

2.1.3 Configuration File

To control the trace-based simulation framework and define, for example, which mapping specification should be evaluated next, the intervention of the designer is required after each simulation. This results in the need of supervising the simulation while running through different mapping files what might be time consuming and does not allow the fully automation of the trace-based simulation. As a consequence of the implementation of the above mentioned modification, more parameters must be passed at the beginning to select all options. The passing of all parameters every time is tedious and can be error-prone. Therefore, a script or a configuration file is needed in order to support the handling of the simulation.

The suggested solution allows to choose flexibly between automatic and manual operation modes. Simulation settings, architecture and mapping files can be predefined in a XML configuration file before running the simulation. Inputs like example and mapping file to simulate, which not have been specified in the file, will be asked as before, during the simulation. If the simulation modes or architecture specification is not defined in the configuration file, then default value will be used for simulation, e.g. no simulation mode and the “architecture.xml”. We have chosen an XML format for the configuration file. This is convenient because no additional parser must be included in the framework since all existing specification files are stored in a XML representation.

2.2 Memory Optimization

As we have mentioned in section 1.3, a main disadvantage of the trace-based simulation framework is the high memory demand for the simulation. Analyzing the framework reveals that the trace events are the main responsible part for the memory consumption. A trace event instance allocates 50 bytes of memory and the application we intend to simulate consist of several millions of events. Multiplying these two numbers implies that a smart design for the trace storage is important. In the next two subsections, modifications towards a better memory usage are proposed and discussed: The postponing of the trace decomposition and the exploitation of repetitive trace events.

2.2.1 Postponing of the Trace Decomposing

In the actual design, the application event trace is decomposed during the initialization phase, as it is illustrated in Figure 2.1. Each communication event, which is originally meant to transmit more data than the defined atomic data size, is split into c_i trace events.

$$c_i = \left\lceil \frac{\text{actual amount of data to communicate}}{\text{atomic data size}} \right\rceil$$

Consequently, the number of trace events is given by N.

$$N = \sum_i c_i + \text{number of computing events}$$

This may result in a infeasible amount of trace events in terms of memory usage if the atomic data size is very small. To solve this problem, it is proposed to postpone the decomposition of the communication events to the time just before the event is dispatched to the scheduler (see Fig. 2.2). So, the additional trace events resulted from the decomposition into atomic data sizes only need to be temporarily created before the dispatch. As a result, the trace size should shrink significantly for high granularity levels. However, the decomposition needs to be done for each mapping specification, whereas the original framework does the decomposition only once during the initialization.

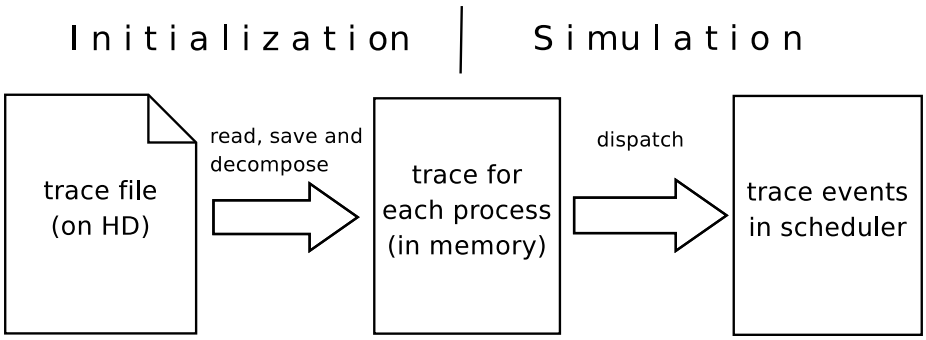


Figure 2.1: Position of the decomposition in the original version

2.2.2 Exploitation of Repetitive Trace Events

A problem not solved yet is that some applications that we would like to simulate, the execution trace contains too many events even without the decomposition into atomic sizes. Thus, a better method to store the trace events has to be found. It can be observed that many trace events are identical. This fact leads us to the next step of optimization.

A pool with reference items including the trace event details is created. For all “equal” trace events only one reference item needs to be allocated. This reference saves only the essential information. The variables used for the simulation, like for example target computation or communication resource, which can be derived from other available information, do not need to be part of the event. In the new coding, all trace events can be specified by a pointer to the according reference

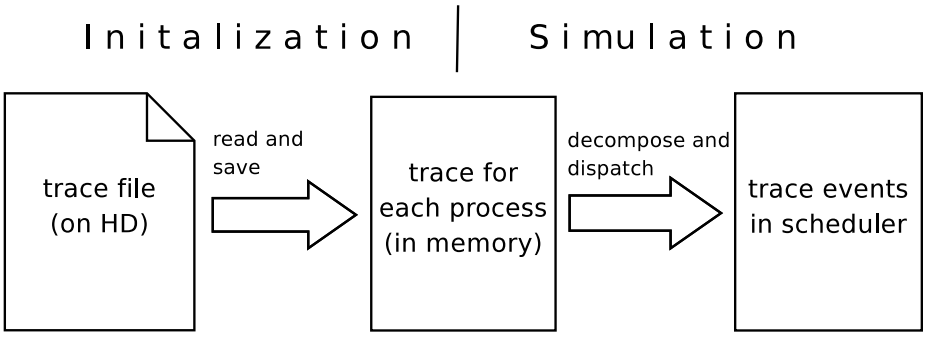


Figure 2.2: Proposed position of the decomposition in the modified version

item. Figure 2.3 illustrates the design approach.

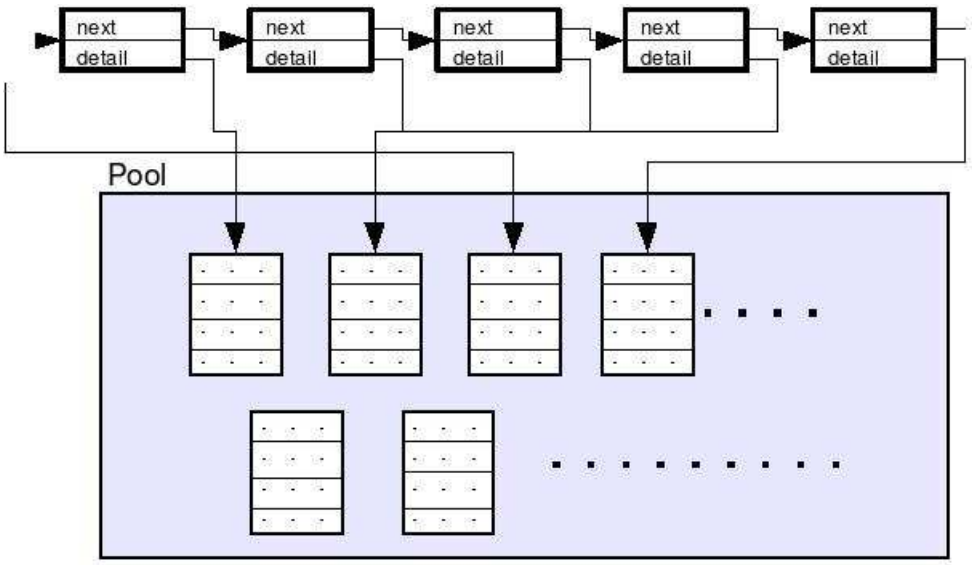


Figure 2.3: Trace list and pool

As result of the modification, the trace event list needs only 8 bytes per element as follows: 4 bytes for the pointer to the next element and 4 bytes for the pointer to the item with the detailed information. This should effect a drastic reduction of memory requirement, especially if the simulated application is highly repetitive in terms of trace events. This design is derived from the so-called “flyweight” design pattern [3].

Chapter 3

Implementation

This chapter presents implementation details of the design modifications proposed in Chapter 2. The first section covers the code refactoring points which effect a more user-friendly style for controlling and extending the framework. The second chapter presents the memory optimization features.

3.1 Code Refactoring

This section discusses first the configuration file and the new added Config class which allow a more user-friendly usability of the simulation. The next two subsections contribute to an easier expendable framework, i.e. to add a new scheduler class. Subsection 3.1.2 explains how we can avoid the conditional inclusions in all the scheduler classes, section 3.1.3 presents the new organisation of the scheduler class tree.

3.1.1 Control of the Simulation

This section presents the configuration file, i.e the `config.xml` and the `Config` class which parses the corresponding XML file and controls the simulation.

Configuration File

An example of the `config.xml` can be seen in the Listing 3.1. On the first hierarchy level of the XML structure, we have the `mode` nodes and an example node.

There are two different `mode` nodes:

- One node has the *name* attribute `VCD_Tracing`. The attribute *value* of this node triggers the simulation mode which enables collecting information for VCD tracing. If the *value* is set to 0, the VCD file will not be generated. If it is set to 1, the VCD tracing will be done.
- The `Perfomance_Statistic` node has besides the *name* attribute, another two attributes. The first attribute is again *value*, which defines if the simulation setting “performance statistic” should be activated or not. The second attribute named *type* is used to specify the output mode. The enumeration of the different types can be extracted from the comment line in the Listing.

The semantics for the example node and its child nodes are:

- The *name* attribute of the `example` node is used to select the example. Thereby, the value specified in *name* is concatenated to “example” and this is the name of the folder containing the example to simulate, e.g. for the value “1”, the example to simulate will be stored in the folder `example1`. The example node has the following child nodes:
- The sub-node `architecture` denotes the architecture specification to simulate. If it is not used, the `architecture.xml` is set as a default value.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <config>
3
4      <mode name="VCD_Tracing" value="0"/>
5      <!-- type=1 -> write output to console, type=2 -> write it to file -->
6      <mode name="Perfomance_Statistic" value="1" type="1"/>
7
8      <example name="1">
9
10         <architecture name="architecture.xml">
11             <mapping name="mapping_fifo.xml"/>
12             <mapping name="mapping_rr.xml"/>
13             <mapping name="mapping_rr2.xml"/>
14         </architecture>
15         <architecture name="architecture_2.xml">
16             <mapping name="mapping_rr3.xml"/>
17             <mapping name="mapping_tdma.xml"/>
18         </architecture>
19
20     </example>
21
22 </config>
```

Listing 3.1: `config.xml`

- The child node `mapping` can either be a child node of the architecture node or the example node in the case when the architecture node is not specified. The mapping nodes are used to enumerate the mapping specifications to simulate for each architecture specification if one or more architecture is listed or else for the default value `architecture.xml`.

The Config Class

The functionality of the `Config` class is explained in this subsection. In the initialization phase of the simulation, the configuration file is parsed, instantiating the `Config` object. The mapping and architecture file names are not stored, only boolean variables are set to `true` if mapping file names and architecture file names other than the default ones are available. The names are read as soon as the next mapping or architecture specification is due to be simulated. In case that no names are available, the two booleans remain `false`.

How the next mapping specification file name can be accessed appears in Listing 3.2. If file names are specified in the configuration file, i.e. the mapping variable is `true`, the next file name (at position `i`) will be read and returned. If no file name is given, it will ask the designer by a command line request and will return the entered name.

A similar function manages the architecture specification file names. The difference is only that a default value will be returned if no name is configured.
Listing 3.3 shows the frame of the root file of the simulation. It can be observed how the functions of the `Config` class are applied to control the workflow of the initialization and simulation. The file names for setting up the architecture (line 19) and mapping (line 29) objects are received from the `Config` class (as explained before) as well as the conditions for the `while` loops (lines 16, 26). The functions generating the value for the conditions of the `while` loops return `true` as long as other mapping or architecture specifications are listed in the configuration file. In case that no mapping file is configured, the client can decide to stop or proceed. The meaning of the lines 34 - 38 is investigated in the next section.

3.1.2 Avoidance of Macros

The implementation of the proposed design in section 2.1.1 is outlined next. Because we do not want to use the conditional inclusions anymore, a concept with two different classes is proposed. A parent class for the two simulation modes (see Section 1.1.1) must be composed which contain a blank function as implementation. Both classes are inherited by a subclass which overrides

```
1  const char *Config::get_mapping(int i)
2  {
3      if (mapping)
4      {
5          XMLNode xml_mode_exam_node;
6          xml_mode_exam_node = xml_mapping_nodes.getChildNode("mapping", i++);
7          cout << "\nmapping file name "
8              << xml_mode_exam_node.getAttribute("name") <<  "\n\n";
9          strcpy(map_name,xml_mode_exam_node.getAttribute("name"));
10         return xml_mode_exam_node.getAttribute("name");
11     } else
12     {
13         char temp[NAME_LENGTH];
14         cout<<"Please enter mapping file name:\n";
15         fgets(temp,NAME_LENGTH,stdin);
16         temp[strlen(temp)-1]='\0';
17         strcpy(map_name,temp);
18         return map_name;
19     }
20 }
```

Listing 3.2: `get_mapping` function form the `Config` class

these empty functions with the real implementation. The function with the actual functionality or the empty function will then be executed, depending on the instantiated class. Only the functions called from outside the `Performance_Statistic` or the `VCD_Tracing` class need be declared in the parent class.

In case of the simulation mode “Performance Statistic”, the setup is a little more complex than the “VCD Tracing” because each bus, processor and process have its own object to administrate the evaluation data. These objects are instantiated in the constructor of the `Performance_Statistic` class, thus the constructor of the parent class must not be empty but it has to construct a “light” version of the object for each resource or process. Figure 3.1 presents a class diagram demonstrating the schematic design. To keep the overview, the diagram is simplified and shows only the `Bus_Performance` objects.

Now, we come back to the lines 34 - 38 in Listing 3.3. Line 35 calls the function shown in Listing 3.4 and assigns the return value to the `performance_statistics` pointer. The function returns either a instance of the parent class `Performance_Statistic` or the child class `Performance_Statistic_Imp` depending on the boolean `with_perform` which is set according to the entry in the configuration file. A similar function is used to set the `vcd_tracing` pointer. Henceforth, all conditional inclusions can be avoided in the code. If a function of the `Vcd_tracing` or the `Performance_statistic` object is called, either the empty functions or the overridden functions will be executed according to the configuration.

3.1.3 Reorganisation of the Scheduler Class Hierarchy

This section investigates the new design of the scheduler class tree. The result of the reorganisation can be seen in the diagram of Figure 3.2. The diagram reveals that the attributes are declared at the highest level where they are commonly shared by all the classes and subclasses. Moreover, the implementations of the operations of a concrete scheduler are distributed over all levels. In the next paragraphs, it is reasoned why an implementation is put on a certain level.

The first two functions of the root class both named `get_ready_queue` are used to get the entire ready queue list of a scheduler or a certain ready queue specified by its name. Because all schedulers use the principle of ready queues, these functions can be implemented in the first level.

The `scheduling` function is the core of each scheduler type and must be implemented for each scheduler separately. The function declarations on the two upper layers are abstract and effects that these classes can not be instantiated. The function `add_trace_event` can not be equal for bus arbiters and processor schedulers, so the implementation is forced to be in the second layer.

```
1  int sc_main(int argc, char *argv[])
2  {
3      Config *conf;
4      strcpy(config_file_name, "config.xml");
5      conf = new Config(config_file_name);
6
7      strcpy(example_name, conf->getExample());
8
9      copy here the example_name to all other file name as prefix.....
10
11     /*Create application from process network file and trace file.*/
12     applic = new Application(app_file_name, trace_file_name);
13
14     int conf_arch = 0;
15
16     while (conf->loopArchi(conf_arch))
17     {
18         /*Create Architecture from architecture xml file.*/
19         strcat(arch_file_name, conf->getArchi(con_arch));
20         archi = new Architecture(arch_file_name);
21
22         /*Create mapping object.*/
23         mapping_ptr = new Mapping(application_ptr, architecture_ptr);
24
25         int conf_map = 0;
26         while(conf->loopMapping(conf_map))
27         {
28             /*Set mapping.*/
29             strcat(mapping_file_name, conf->getMapping(conf_map));
30
31             /*Do mapping.*/
32             mapping_ptr->set_mapping_from_file(mapping_file_name)
33
34             /*Create Performance_Statistic to collect evaluation results.*/
35             performance_statistics_ptr = conf->get_perform_obj(applic, archi);
36
37             /*Set the vcd tracing to get the waveform of the simulation. */
38             vcd_tracing = conf->get_vcd_obj(vcd_trace_file_name);
39
40             /*Start simulation.*/
41             simulate
42             output the evaluation data
43
44             conf_map++;
45         }
46         conf_arch++;
47     }
48
49 }
```

Listing 3.3: Frame of the `sc_main` function from the root file of the simulation

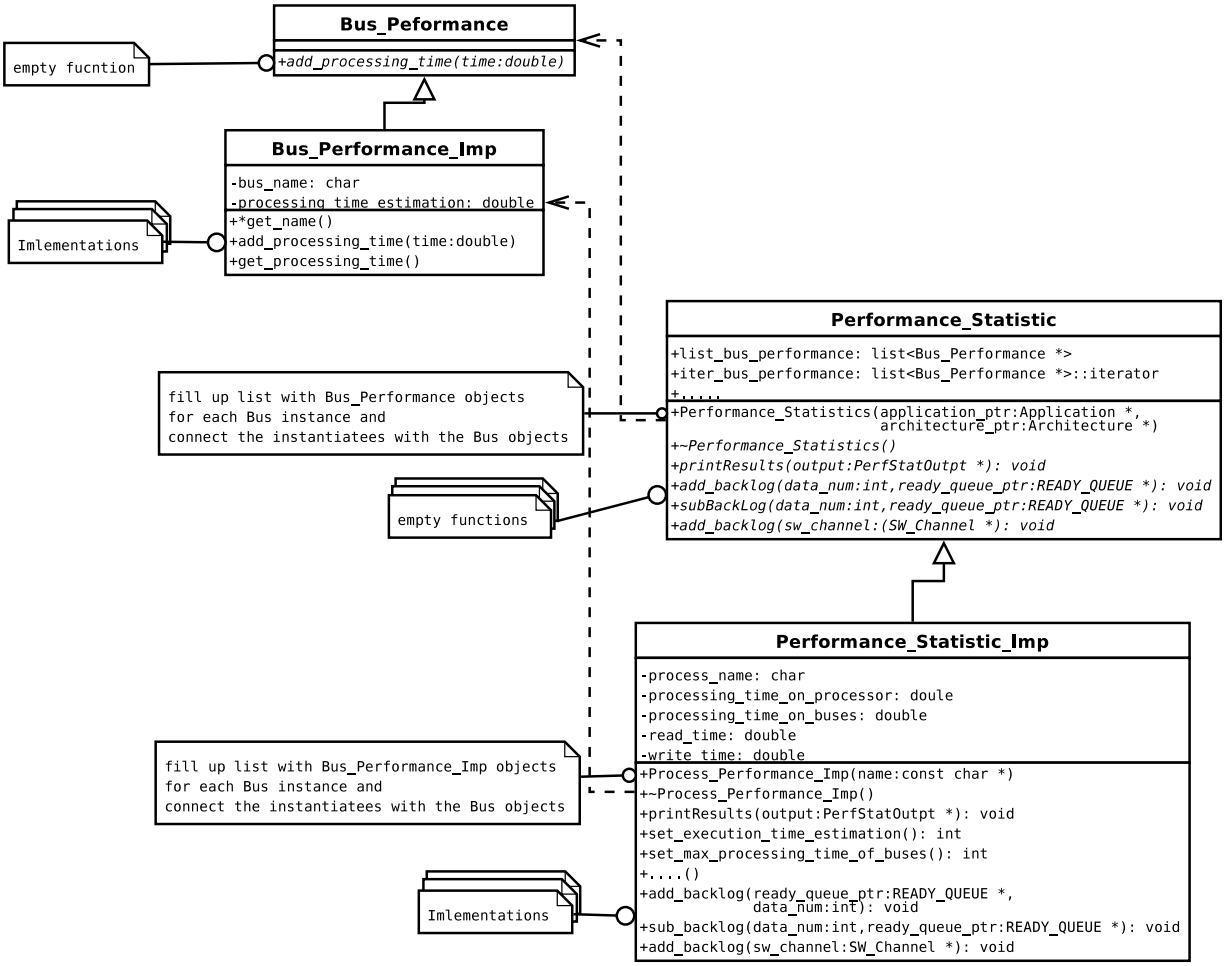


Figure 3.1: Diagram for the `Statistic_Performance` class

```
1  Performance_Statistics *Config::get_perform_obj(Application *app, Architecture *arch)
2  {
3          if (with_perform)
4          {
5                  return new Performance_Statistics_Imp(app, arch);
6          }
7          else
8          {
9                  return new Performance_Statistics(app, arch);
10         }
11
12 }
```

Listing 3.4: `get_perform_obj` function from `Config` class

This function can be totally different in a fancy scheduler and hence can be overridden by a leaf class function.

The two last functions named `set_recourse_belong_to` are both declared here despite they are implemented one level down. However, this design allows the `Resource` interface to use the same interface for both bus arbiter and processor scheduler.

The remaining functions are all coded in the basis class, but can be overridden by a lower scheduler class if necessary. The function `add_ready_queue` is called from the Mapping class to add a ready queue to a scheduler, according to the information obtained from the parsing of the mapping file. The other functions assist the scheduling function. They can differ from the standard implementation for some schedulers, so the concrete classes can override these functions from the `Bus_arbiter` or the `Processor_sched` classes as it can be seen in the diagram.

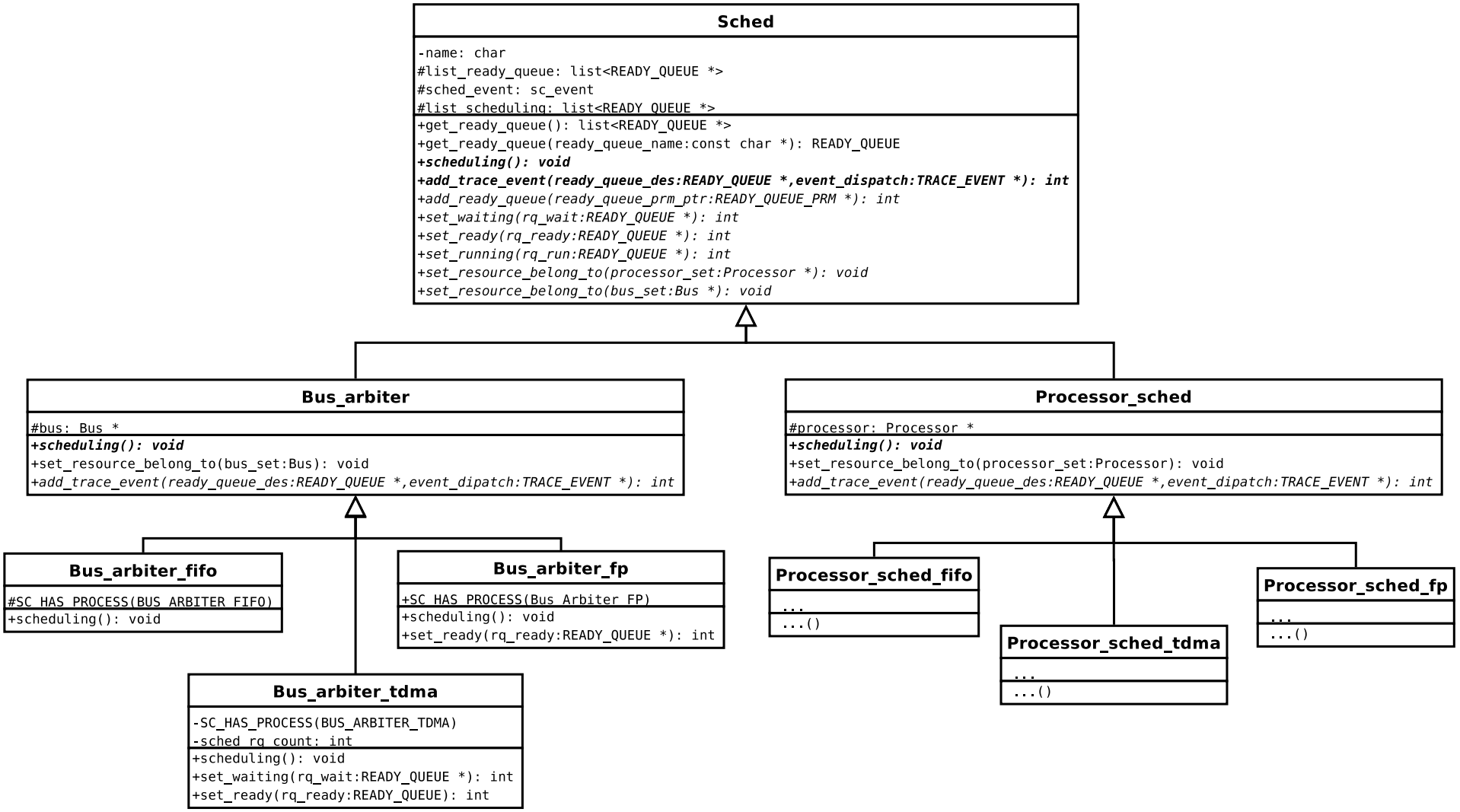


Figure 3.2: Diagram of the scheduler classes

3.2 Memory Optimization

This section presents the implementation of the proposed design modification in section 2.2. Both the initialization and simulation phases of the trace-based simulation have been modified. The first subsection discusses the changes in the initialization phase, while the second subsection discusses changes in the simulation phase.

3.2.1 Initialization Phase

The trace is stored in the initialization phase, therefore the realization of the trace storage method stated in 2.2.2 is presented in this subsection. Two steps lead to the new implementation of the new trace storage method.

1. Step: Avoidance of Redundancies

The first step analyzes which information for the trace event must really be stored in this phase and which can be attached later. Listing 3.5 shows the structure of the TRACE_EVENT from original framework. In the following list, we explain why we do not need every variable.

- The `event_flag` variable is used to mark the last event of the communication event transformation. Due to the postponing of the decomposition (which is explained later), the variable is dispensable until the simulation phase.
- The `compu_time_left` variable stores the simulated time remaining to finish the computation event and hence is not utilized in the initialization phase.
- The `compu_resource` pointer contains the address of the computation resource. The Process class saves this information as well and therefore it is enough to attach this information just before the dispatch to the scheduler, i.e in the simulation phase.
- The `data_transfer_time` is derived from the data_num variable later in the simulation mode and therefore it is not needed yet.
- The `commu_resource_head` and the `current_commu_resource` variables store redundant information with the `sw_channel_ptr` during the initialization phase. They are only used in the simulation phase.
- The `complete_event` is only used during the simulation phase and can be attached before the dispatch as well.

The result of this “slimming diet” can be extracted from Listing 3.6. The TRACE_EVENT_INT presents the structure storing the information in the main memory for each trace event in the new version. The consideration that many of these structure are identical leads to the implementation of the pool feature with the pool (presented in the next subsection). The main difference to the original version is that the information for the trace event is saved in two structures. One is the TRACE_EVENT_EXT structure and the other one the TRACE_EVENT_INT structure which has been mentioned before. The name TRACE_EVENT_INT stands for intrinsic trace event, meaning that in this structure only the common information with other trace events is saved. Whereas in the TRACE_EVENT_EXT structure, extrinsic data is saved. In our case, this is only a pointer to the next intrinsic trace event to maintain the order of events and a pointer to the characterizing intrinsic trace event. The structure of the extrinsic trace event is also shown in Listing 3.6.

2. Step: From File to the Trace List

This subsection presents the path of the trace events, from the trace file(s) to the trace list and pool of each process. Figure 3.3 illustrates which classes and functions read and pass along the trace events until saved ready for simulation:

- The first box represents the `get_trace_from_file` function from the `Application` class. This function reads all trace files and sends the trace event to the according `Process` object. Nothing of the content of the first box is modified in the new version with respect to the original.

```
typedef struct TRACE_EVENT
{
    char event_type;
    char event_flag;

    /*For computation.*/
    double *compu_time_ptr;
    double compu_time_left;
    COMPUTATION_RESOURCE *compu_resource;

    /*For communication.*/
    int data_num;
    double data_transfer_time;
    SW_Channel *sw_channel;
    COMMUNICATION_RESOURCE *commu_resource_head;
    COMMUNICATION_RESOURCE *current_commu_resource;

    sc_event *complete_event;
}TRACE_EVENT;
```

Listing 3.5: TRACE_EVENT

```
typedef struct TRACE_EVENT_INT
{
    char event_type;
    double *compu_time;

    int data_num;

    SW_Channel *sw_channel;
}TRACE_EVENT_INT;

typedef struct TRACE_EVENT_EXT
{
    TRACE_EVENT_INT *detail;
    TRACE_EVENT_EXT *next;
}TRACE_EVENT_EXT;
```

Listing 3.6: TRACE_EVENT_INT and TRACE_EVENT_EXT

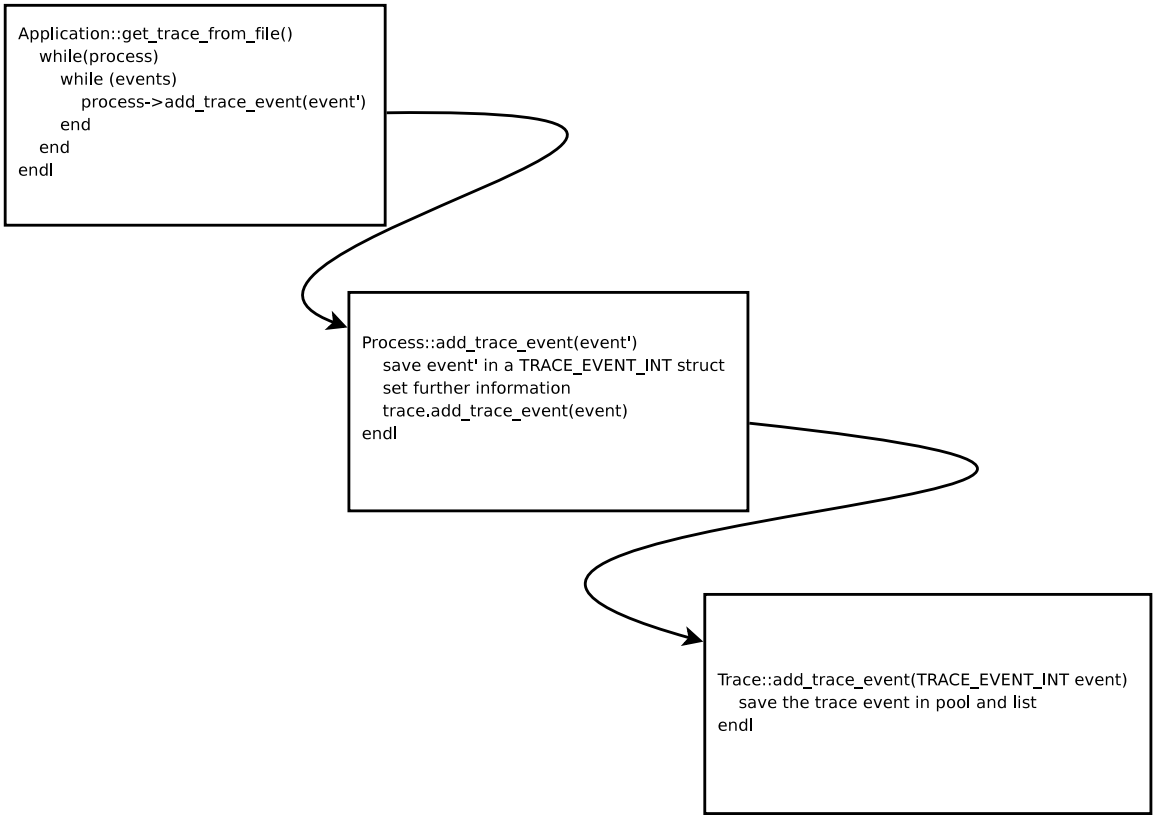


Figure 3.3: Path of the trace events during the initialization

- In the second box, the functionality of the `add_trace_event` of the Process class is:

  1. Store received information in a `TRACE_EVENT_INT` structure.
  2. Update the computation estimation table.
  3. Pass the trace event to the `Trace` object, a member variable of the `Process` class, where the events will be stored in the right order.

  The decomposing is replaced from this function running during the initialization to the function `running` of the `Process` running during the simulation in the new version.

- The functionality of the third box is more detailed demonstrated in the diagram of Figure 3.4. The intrinsic trace event information of the trace event is received as argument. This structure is compared with all other ”intrinsic objects already placed in the pool. If the object is new, i.e no identity is found, it will be stored in the pool as well. The pool is implemented as a linked list. Afterwards, the ”extrinsic“ structure is created which points to the according object in the pool and of course to the next event. Because the trace events are received in order, they only must be referenced to the next event pointer of the last arrived trace event.

3.2.2 Simulation Phase

All modifications done in the simulation phase code concern only the `running` function of the `Process` class. The modifications can be summarized in two points.

1. The re-transformation of the trace event to the old shape.
2. The decomposition

Figure 3.5 shows the flow of the trace events during the simulation. Because only the code represented by the first box is changed (in terms of memory optimization), we will not pay attention to the implementation of the other encircled boxes. The interaction of the `Scheduler` class and

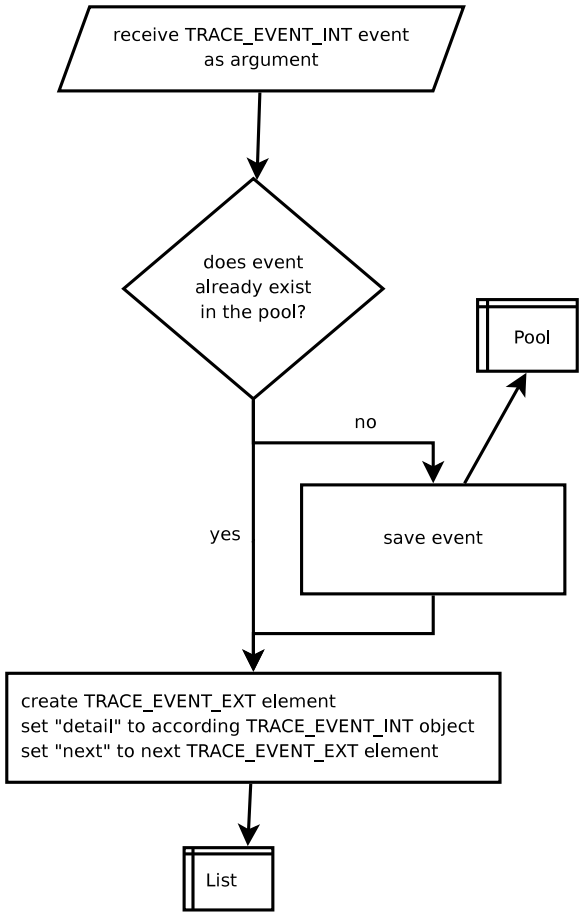


Figure 3.4: Diagram of the of the `add_trace_event` function of the `Trace` class

the running function of the `Process` class can be extracted from the schema. A simplification is made due the numbers of resources (encircled boxes) which can vary according to the number of different resources in the communication path.

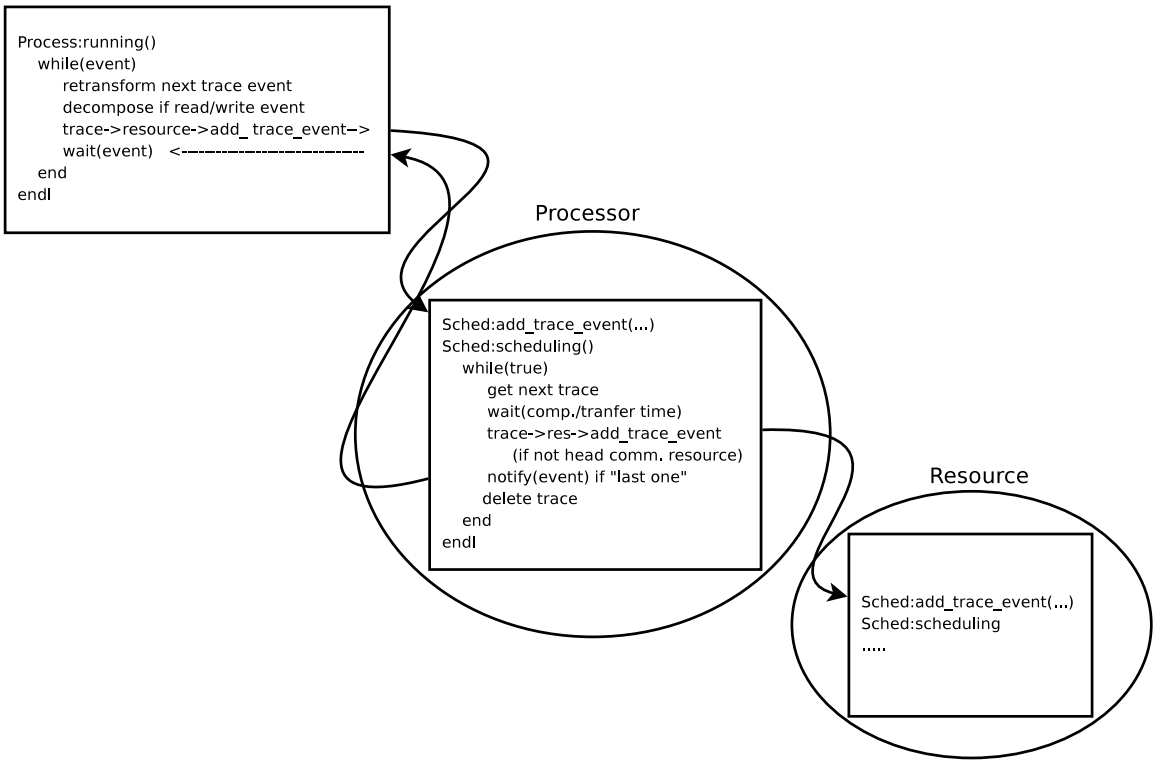


Figure 3.5: Trace event flow during simulation (simplified)

Re-transformation

The re-transformation ensures that the trace event received from the trace list, i.e from the `Trace` object, is copied back to a `TRACE_EVENT` structure of similar structure as in the original framework (see List. 3.5). The only difference is that the next pointer is not deployed anymore, because this information was only used to keep the order in the `Trace` class. All the information for the additional variables can be derived from the information stored in the variables of the `TRACE_EVENT_INT` structure. However, they are needed during the simulation phase to save temporary data.

Decomposition

The decomposition is only applied to the communication events. If the `data_num` value (amount of data to tranfer) exceeds the atomic data size, the structure created in the re-transformation will be cloned as many times as required to meet the corresponding granularity level given by the atomic data size. The copying and dispatching is done repeatedly: The fist events gets dispatched, then it is checked if a further instance is needed to meet the atomic data size. If so, a copy is made and dispatched. The proper rest value and the event flag `LAST_ONE` must be assigned to the `data_num`, respectively to the `event_flag` variable of the last object of the loop.

This two modifications effect that the workflow and the object representation from this point remains identical as in the original framework. Therefore, the code does not require any changes after the dispatch of the trace events.

Chapter 4

Evaluation and Results

In this chapter, we evaluate the performance of the suggested modifications to the existing trace-based simulation framework and discuss the results. The evaluation has been done with the MPEG-2 decoder application developed in a former diploma thesis [5]. All specification files and video clips[1] used for the evaluation can be found in the attached software CD-ROM. The following result are presented: In this first section, the memory requirement for the trace and memory usage during the simulation are compared between the modified and the original framework. The speed of both versions is analyzed next. The third section deals with some interesting aspects coming up in a case study of performance evaluation due to enhanced simulation possibilities.

4.1 Memory Optimization

The modifications presented in Section 2.2 show two effects: On the one hand, the memory used by the trace is independent of the atomic data size. On the other hand, the memory requirement for the application with highly repetitive traces i.g. MPEG-2 decoder is reduced significantly.

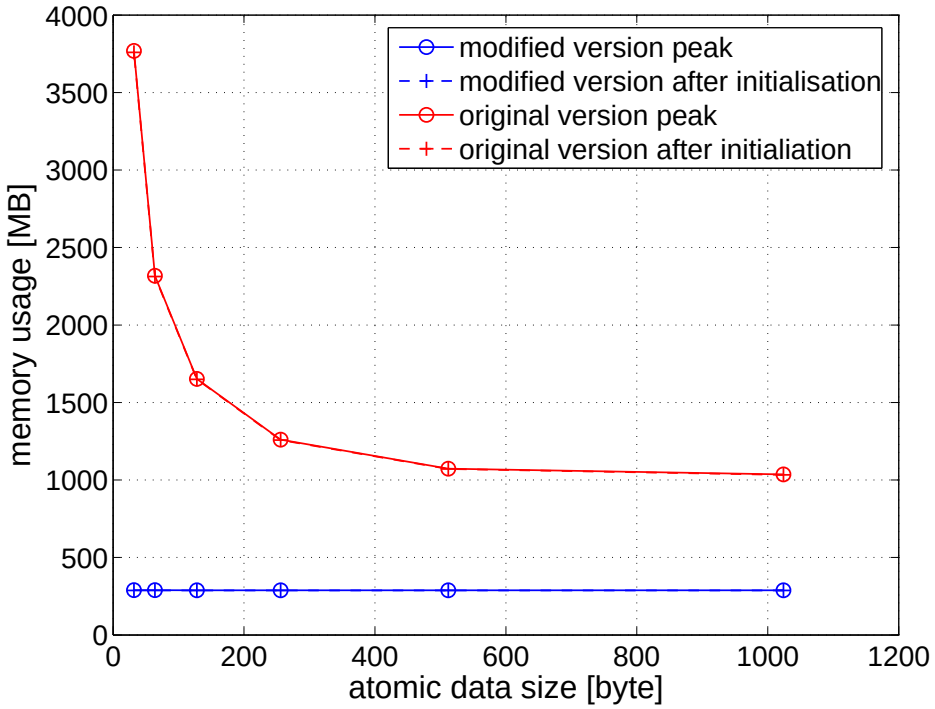
Figure 4.1 dipicts 4 different curves: A peak memory curve and a trace size curve for both the original version and the modified version. The trace size curves represent the memory usage after the initialization, whereas the the peak memory curves shows the maximal memory usage during the simulation. The figure reveals that for the original version, the memory usage rises exponentially for smaller atomic data sizes while the memory usage of the modified version is independent of the granularity level. For both framework, the difference between the peak memory and the trace size is less than 1%. These observations can be explained as follows:

Regularly, the communication trace events are split into separate trace events which present a transfer action of the size of atomic data size or less.

- In the original framework, this step is executed in the initialisation process, so the memory requirement rises significantly for small values.
- In the modified framework, the trace size remains stable over all atomic data sizes due to the postponing of the decomposing in atomics events.

The slight increase of the interval between the memory requirement only for the trace size and the peak memory point during the simulation can be explained by the fact that more trace events, with a smaller data size, must be copied temporarily and then dispatched to the scheduler before the next `wait` in the simulation. This effect can be recognized with both candidates but it is less distinct in the changed version. The difference is only 2MB whereas the the original framework allocates 10MB more memory for an example with an atomic data size of 32 bytes. Figure 4.2 shows in detail the difference between the peak memory usage and trace memory usage during simulation for the modified framework. Moreover, for the modified framework, the trace can be extracted down to very small atomic data size values, i.e. 2 bytes. In contrast, the original version

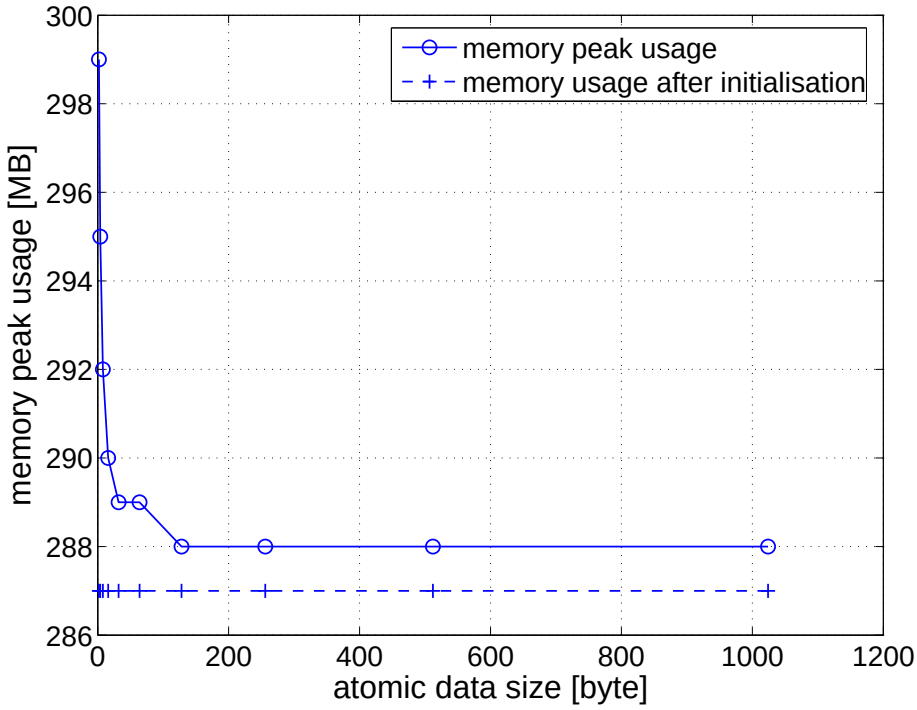
[1] Specification of MPEG-2 videos: Frames per second: 25; resolution: 704 x 576 pixels; length: 2 seconds (except the videos for the experiment generating Fig. 4.3 which evaluates videos with different lengths).



atomic data size [byte]	32	64	128	256	512	1024
memory peak [MB] (modified ver.)	289	289	288	288	288	288
trace size [MB] (modified ver.)	287	287	287	287	287	287
peak memory [MB] (original ver.)	3769	2318	1652	1260	1073	1036
trace size [MB] (original ver.)	3759	2313	1649	1257	1071	1033

Figure 4.1: Memory requirement comparison

will overfill the user memory[2] already on a granularity level of 16 bytes per communication trace.



atomic data size [byte]	2	4	8	16	32	64	128	256	512	1024
memory peak [MB]	299	295	292	290	289	289	288	288	288	288
trace size [MB]	287	287	287	287	287	287	287	287	287	287

Figure 4.2: More detailed memory requirement plot of modified version

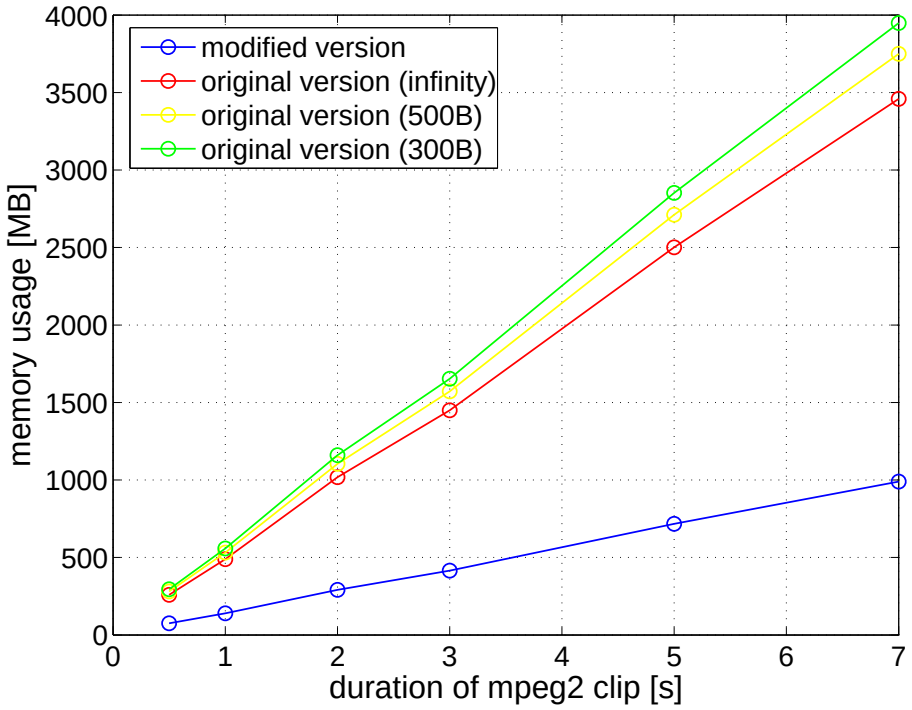
Figure 4.3 shows that the memory requirements of both frameworks depend linearly on the video clip duration. The fact that a pool of reference trace events has to be allocated in the modified version, can be neglected since already a short clip consists of a large amount of trace events, and the actual memory usage is essentially improved with the modified version. Because in the original version the memory usage depends on the atomic data size, several illustrative curves are drawn for different atomic data sizes. We can observe that the modified framework saves about 25% of memory compared with the original framework not splitting up any communication event (which corresponds to an infinite atomic data size). By reducing the atomic data size, the slope of the memory requirement curve produced by the original framework always rises because the communication events are decomposed to more and more fine grained events.

4.2 Speed Comparison

In this section, the speed performance of the proposed version is investigated and compared with the original one, for both initialization and simulation phases. First, we compare the initialization step. The duration of the initialization step is mainly defined by the procedure of loading the traces from the disk into the main memory (except of a small fraction of time to create the application and architecture objects). Second, we compare the simulation time of the modified version with respect to the original one, for different simulation modes.

As it can be seen in figure 4.4, the initialization time for the original framework is exponentially decaying and for the modified version it is constant (as it was the case of the memory usage curves in figure 4.1). In the original framework, the behavior is as follows: the higher the level of fragmentation of the communication events, the more time is used to load the trace into the

[2] 4GB in case of a 32bit application running on a 64bit server



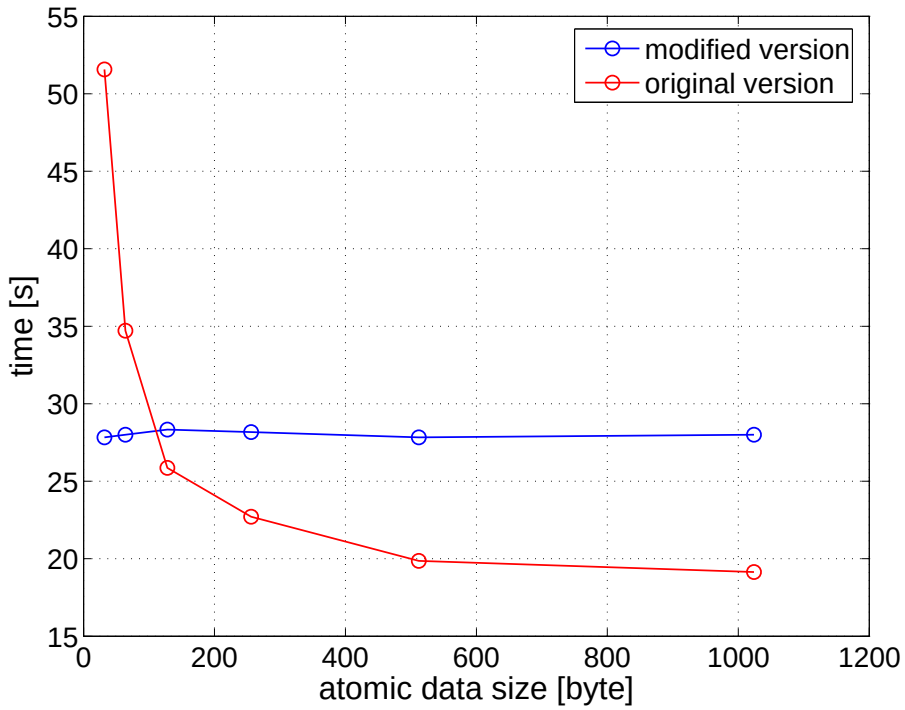
duration mpeg2 clip [s]	0	1	2	3	5	7
trace size [MB] (modified ver.)	75	140	291	415	717	990
trace size [MB] (infinite)	259	489	1018	1450	2501	3459
trace size [MB] (500Bytes)	281	531	1104	1572	2711	3750
trace size [MB] (300Bytes)	295	558	1160	1653	2852	3948

Figure 4.3: memory usage comparison over different clip duration. Three curves with three different atomic data size values are drawn for the original versions

memory for the simulation because of the decomposition. Following the curve of the new version, it can be noted that the initialization duration remains equal. However, the performance of the memory optimized version is poorer for atomic data sizes higher than 128 bytes. This loss is introduced by the new design for storing the trace events which is described in the subsection “2. Step” of section 3.2.1.

The comparison of the simulation time is shown in the next three figures 4.5, 4.6 and 4.7, for different simulation modes (see section 1.1.1). Fig. 4.5 and Fig. 4.6 illustrate the simulation time for both frameworks in two distinct modes, i.e. “performance evaluation” and “VCD tracing” modes. For both modes, the speed of the modified framework is slower than the original. The exact speed loss or gain can be extracted from the figures.

The third figure (Fig. 4.7) shows the simulation speed of both framework having both simulation features enabled, i.e. performance evaluation and VCD tracing. In this third mode, the modified framework is a little faster, i.e 4%. The speed improvement can be explained by the code design. As explained in section 3.1.2, the new framework uses empty functions for the inactive simulation mode. This seems to slow down the simulation speed for single mode simulations (Fig. 4.5 and Fig. 4.6). In exchange, the advantage is that switching between modes does not need a different code compilation.

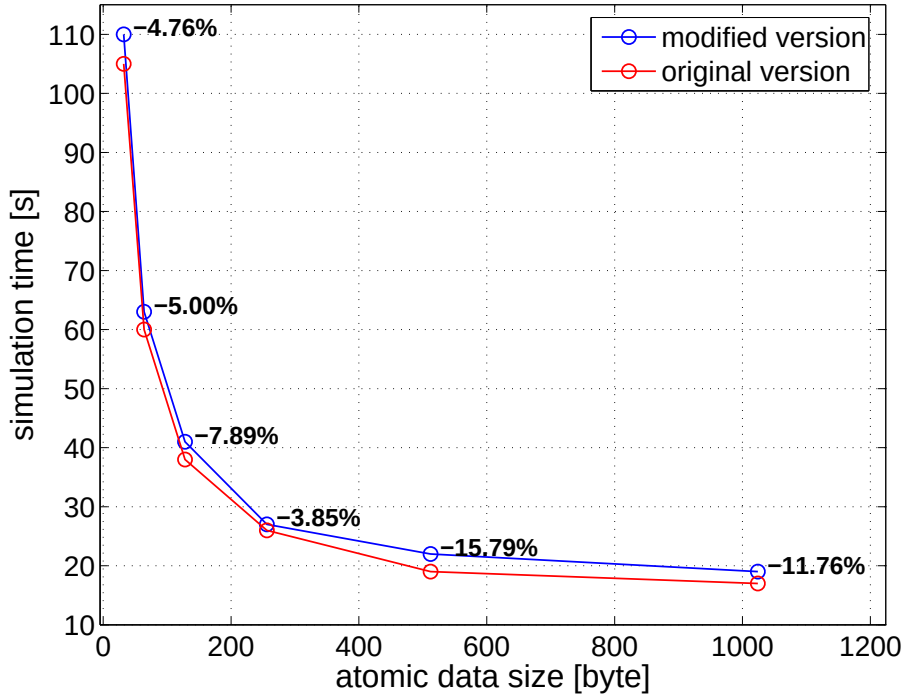


atomic data size [byte]	32	64	128	256	512	1024
time [s] (modified ver.)	28	28	28	28	28	28
time [s] (original ver.	52	35	26	23	20	19

Figure 4.4: Initialisation time

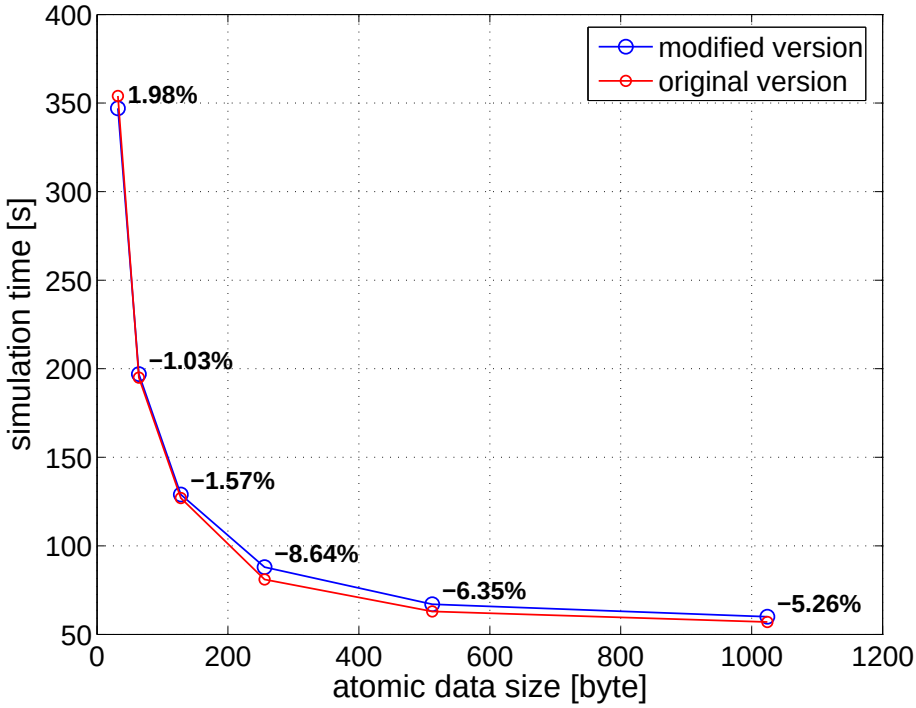
4.3 MPEG-2 Case Study

So far, the performance improvement of the modified framework has been discussed. Making use of the memory optimization, simulation can be done with a highly increased granularity. Hence, the trace-based simulation provides more accurate results. This may evince some new findings in the performance evaluation.



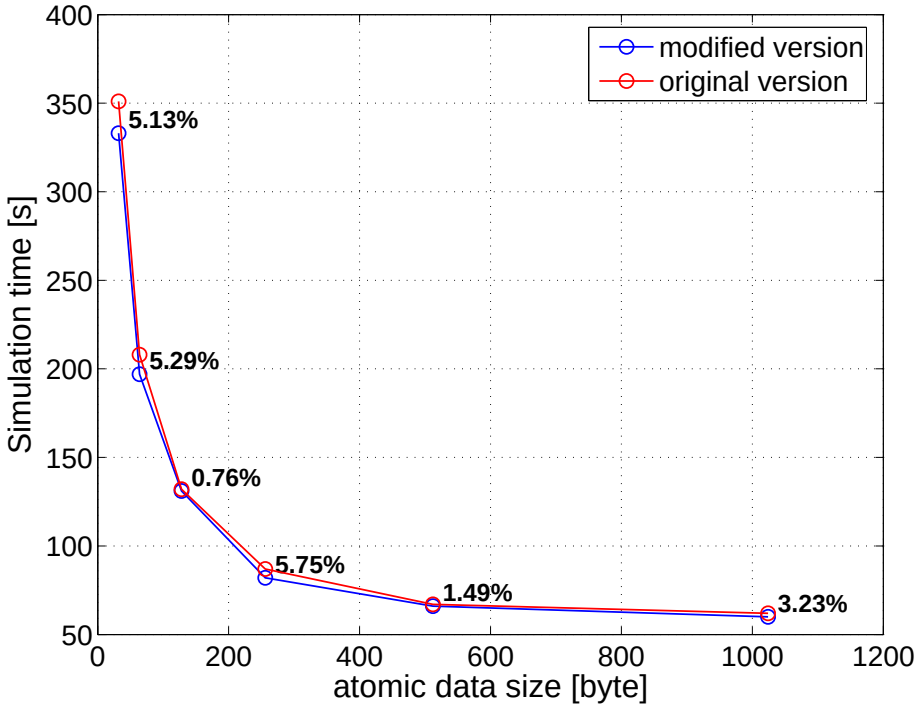
atomic data size [byte]	32	64	128	256	512	1024
time [s] (modified ver.)	110	63	41	27	22	19
time [s] (original ver.	105	60	38	26	19	17

Figure 4.5: Simulation time (with performance statistics)



atomic data size [byte]	32	64	128	256	512	1024
time [s] (modified ver.)	347	197	129	88	67	60
time [s] (original ver.	354	195	127	81	63	57

Figure 4.6: Simulation time (with VCD tracing)



atomic data size [byte]	32	64	128	256	512	1024
time [s] (modified ver.)	333	197	131	82	66	60
time [s] (original ver.	351	208	132	87	67	62

Figure 4.7: Simulation time (with VCD tracing and performance statistics)

In this experiment, the estimated execution time of the MPEG-2 application to decode a video clip with a duration of two seconds is inspected for different levels of granularity and two mapping specifications. One mapping uses the fixed priority scheduler for all resources, while the other mapping uses the TDMA scheduler. However, we used the same architecture and binding specifications for both mappings. By studying the plot in Figure 4.8, two aspects can be observed:

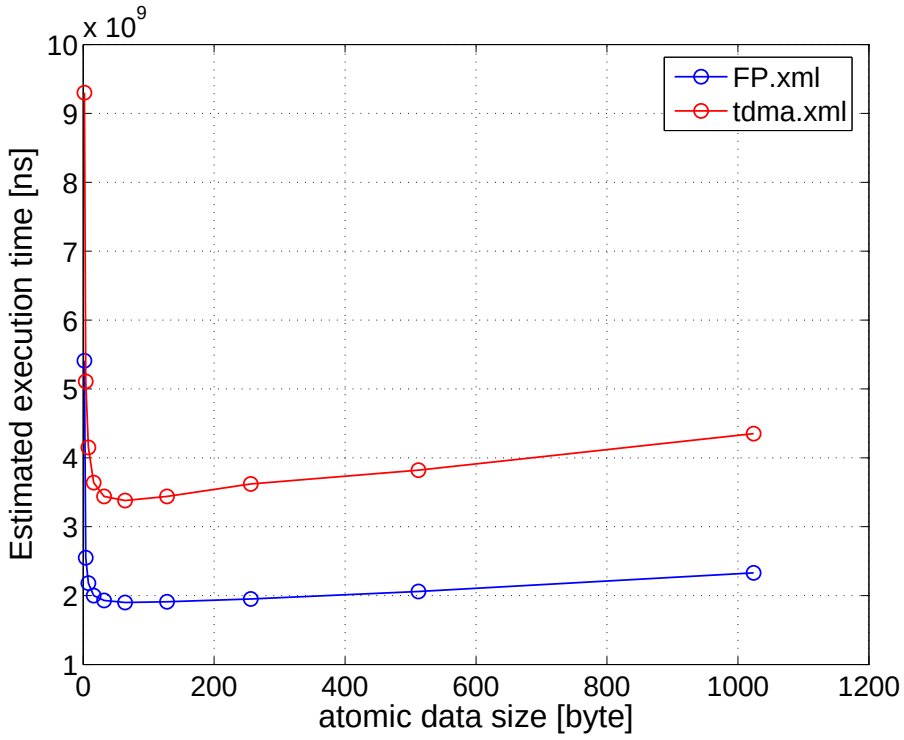
1. Both curves show a global minimum at the atomic data size value of 64 bytes, by decreasing or increasing the data size, the estimated execution time increases.

2. The estimated execution time to decode the video clip is about twice as much for the TMDA scheduler than for the FP scheduler over all atomic data sizes. So the TDMA scheduler improves one second by lowering the atomic data size from 1024 to 64 bytes whereas the FP scheduler improves a half second.

The effects observed previously can be explained as follows:

1. Decreasing the atomic data size allows more switches between processes, so the probability of a blocked process, which for example wastes the time slot in case of TMDA scheduling, diminishes. Coming to a certain point in the atomic data size granularity, the context switch overhead becomes dominated and therefore, the estimated execution time increases again.

2. The explanation for the second occurrence is that preemptive schedulers are more efficient. But the effiency of the TDMA scheduler increases more by lowering the atomic data size than the efficiency of a preemtive scheduler.



atomic data size [byte]	2	4	8	16	32	64	128	256	512	1024
estimated execution time [ns] (FP.xml)	5.41e+09	2.55e+09	2.18e+09	2.00e+09	1.93e+09	1.90e+09	1.91e+09	1.95e+09	2.06e+09	2.33e+09
estimated execution time [ns] (tdma.xml)	9.30e+09	5.11e+09	4.15e+09	3.64e+09	3.44e+09	3.38e+09	3.44e+09	3.62e+09	3.82e+09	4.35e+09

Figure 4.8: Estimated execution time to decode a MPEG-2 video clip with a duration of 2 seconds

Chapter 5

Conclusion and Future Work

This chapter gives a conclusion for the whole thesis and proposes some future work. In the first section, the major design changes and their impacts are summarized. The second section gives some ideas for future work to improve the framework further.

5.1 Conclusion

In this thesis we have proposed various design modifications for the trace-based simulation framework to optimize the memory usage and the usability in terms of extending the simulation with additional schedulers and controlling the simulation. The following points summarize the modifications which lead towards a better usability:

- Avoidance of macros: Adding an additional class with the same interface as the simulation settings classes `Perfomance_Statistics` and `VCD_Tracing`, but with empty functions allows to avoid the conditional inclusions in the source code. Moreover, no new code compilation is necessary to change the settings and therefore, they can be specified in the configuration file.
- Configuration file: The simulation can run without any designer intervention if all parameters are specified in the `config.xml` before the simulation. The file is then parsed by the `Config` class and takes over the control of the simulation.
- Scheduler class tree: The reorganisation of the scheduler class tree enabled to extend the framework with new schedulers with a smaller effort and without “copy-paste” programming style.

The next two points sum up the modifications which optimize the memory usage and the their achieved results:

- Postponing of the decomposition: The postponing of the trace transformation from the initialization phase to the simulation phase offers the capability of the simulation with a arbitrarily small atomic data size, because the memory demand does not depend on the atomic data size anymore.
- Introduction of a new trace storage method: The idea of the partition of shareable and non-shareable information of the trace events and the consideration of which information really needs to be saved, effected a memory reduction of 25%.
- The speed performance could be maintained despite the memory reduction.

5.2 Future Work

Ideas for future work:

- It can be foreseen that further main memory optimization cannot be achieved anymore because only the minimum amount of information is stored with the new modifications. In order to simulate larger traces, other memory optimization solution have to be investigated, like using the disc as a cache memory. Reading the trace on-the-fly is such an option. Consequently, the trace events are only read from the file when needed. This will allow the simulation of traces as large as the disc capacity could hold. However, attention must be spent to an efficient implementation which could be achieved by reading the trace from the file in a separate thread.
- Changing the atomic data size still needs a new code compilation. The usability of the simulation could be improved further by making it possible to define the atomic data size in the configuration file as well. The `Config` class must then be extended accordingly to parse the value for the atomic data size and assign it to a global variable for example.
- To improve the accuracy of the simulation, bounded hardware buffer sizes should be introduced, so processes get blocked if the buffer is full. The implementation could be done in the same manner as the software channel buffers, e.g. by constructing a semaphore mechanism with the SystemC `wait` and `notify` functions.

Appendix A

CD-ROM Content

The following list describes the content of the CD-ROM. Archives contain a file `README.txt` which further explains their content.

`doc/presentation.pdf`	Thesis final presentation PDF version
`doc/presentation.ppt`	Thesis final presentation PowerPoint version
`doc/tex/howto.zip`	HOW-TO LaTeXsource
`doc/tex/thesis.zip`	Thesis LaTeXsource
`doc/thesis.pdf`	Thesis PDF version
`doc/thesis.ps`	Thesis PostScript version
`dol_ethz.zip`	DOL package
`experiments/example1_2.zip`	Files for experiments of Section 4.1 and 4.2
`experiments/example3.zip`	Files for experiment of Section 4.3
`howto.pdf`	Trace-based simulation HOW-TO
`README.txt`	This list
`systemc-2.2.0.tgz`	OSCI SystemC 2.2.0
`traceSim.zip`	Trace-based simulation source code (with an example)

Bibliography

[1] Shapes project website. `http://www.shapes-p.org`. This is an electronic document. Date retrieved: December 30, 2007.

[2] Shapes@tik website. `http://www.tik.ee.ethz.ch/~shapes`. This is an electronic document. Date retrieved: December 30, 2007.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlisides. *Design Patterns*. Addison-Wesley, 1995.

[4] Jun Liu. A modular trace-based simulation framework for multiprocessor systems. Master's thesis, ETH Zurich, August 2007.

[5] Simon Mall. Mpeg-2 decoder for shapes dol. Master's thesis, ETH Zurich, April 2007.