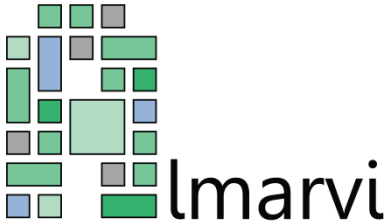


ALMARVI

"Algorithms, Design Methods, and Many-Core Execution Platform for Low-Power Massive Data-Rate Video and Image Processing"

Project co-funded by the ARTEMIS Joint Undertaking under the

ASP 5: Computing Platforms for Embedded Systems

ARTEMIS JU Grant Agreement no. 621439

D4.3 – *Design Space Exploration*

Due date of deliverable: April 1, 2016

Start date of project: 1 April, 2014	**Duration:** 36 months
Organisation name of lead contractor for this deliverable:	TUE
Author(s):	M. Hendriks, S. Seyedalizadeh Ara, A. Baghbanbehrouzian, J.v. Eijndhoven, B.v. Rijnsoever, D. Goswami, T. Basten, M. Geilen
Validated by:	Zaid Al-Ars (TUDelft)
Version number:	1.0
Submission Date:	31.03.2016
Doc reference:	ALMARVI_D4.3_final_v10.docx
Work Pack./ Task:	WP4 task 4.1
Description: ***(max 5 lines)***	This document describes the ALMARVI application development flow and design space exploration methodologies for performance optimization.

Nature:	**R**		
Dissemination Level:	**PU**	Public	**X**
	PP	Restricted to other programme participants (including the JU)	
	RE	Restricted to a group specified by the consortium (including the JU)	
	CO	Confidential, only for members of the consortium (including the JU)	

DOCUMENT HISTORY

Release	Date	Reason of change	Status	Distribution
V0.1	1/12/2015	First draft	Draft	CO
V0.2	13/3/2016	Second draft	Draft	CO
V0.3	21/3/2016	Final draft after revision with reviewer's comments	Draft	PU
V1.0	31/3/2016	Submitted to Artemis	Final	PU

©ALMARVI Consortium

1 Contents

1 Contents ... **3**

2 Summary ... **4**

3 Design-Space Exploration (DSE) ... **5**

3.1 V-model based Design-Space Exploration ... 5
3.2 ALMARVI specific adaptation of development flow ... 7
3.3 Relation to V-model in ALMARVI context ... 8
3.4 Organization ... 9

4 Model-based analysis ... **10**

4.1 Analysis and application mapping on shared resources ... 11
4.1.1 Motivation and Objectives ... *11*
4.1.2 Method ... *11*
4.1.3 Evaluation ... *14*
4.2 Tighter temporal bounds for dataflow applications mapped onto shared resources ... 15
4.2.1 Motivation and Objectives ... *15*
4.2.2 Method ... *16*
4.2.3 Evaluation ... *18*
4.3 Trace based analysis ... 19
4.3.1 Motivation and Objectives ... *19*
4.3.1 Metric temporal logic ... *19*
4.3.2 Examples ... *20*
4.3.3 Good, neutral, bad and informative prefixes ... *21*
4.3.4 Implementation in the TRACE tool ... *21*
4.4 Conclusions ... 22

5 Source code level analysis ... **23**

5.1 Pareon for design-point evaluation and trace visualization support ... 24
5.2 Floating-point to fixed-point Design Report C++ to FPGA conversion ... 25
5.2.1 Goal ... *25*
5.2.2 FAST requirements ... *26*
5.2.3 FAST Design ... *27*
5.3 Conclusions ... 32

6 Conclusions ... **33**

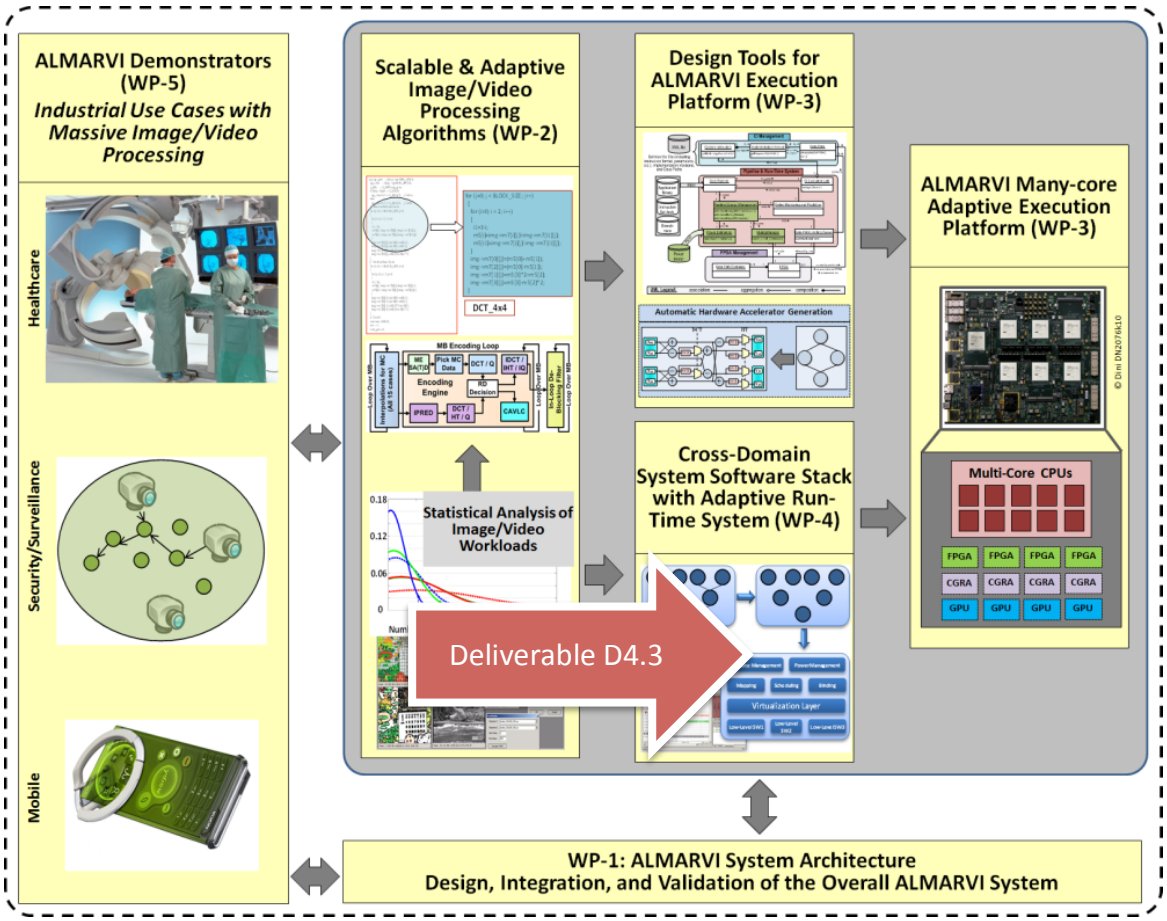
7 References ... **34**

2 Summary

The ALMARVI project aims to develop an approach that allows for portable application software, across a range of modern high performance and energy efficient heterogeneous computing architectures.

This report corresponds to deliverable D4.3 "*Design Space Exploration*" which is part of WP4, Task 4.1. The aim is to develop analysis techniques for systematic design space exploration (DSE) methods dealing with task mapping, scheduling and resource arbitration. This task is built upon the models developed in Task 1.3 to provide the right abstractions of the underlying heterogeneous hardware, applicable at the development level. The DSE targets multiple objectives, performance being the prime objective (often a constraint) in view of various tradeoff between resource usage (cores, memory, cost) and embbed performance.

The Figure below shows where the contributions described in deliverable D4.3 fit within the context of the ALMARVI project.



3 Design-Space Exploration (DSE)

3.1 V-model based Design-Space Exploration

The application development process of ALMARVI follows the V-model for performance engineering [16] as illustrated in Figure 1. The following elaborates various steps in the development process.

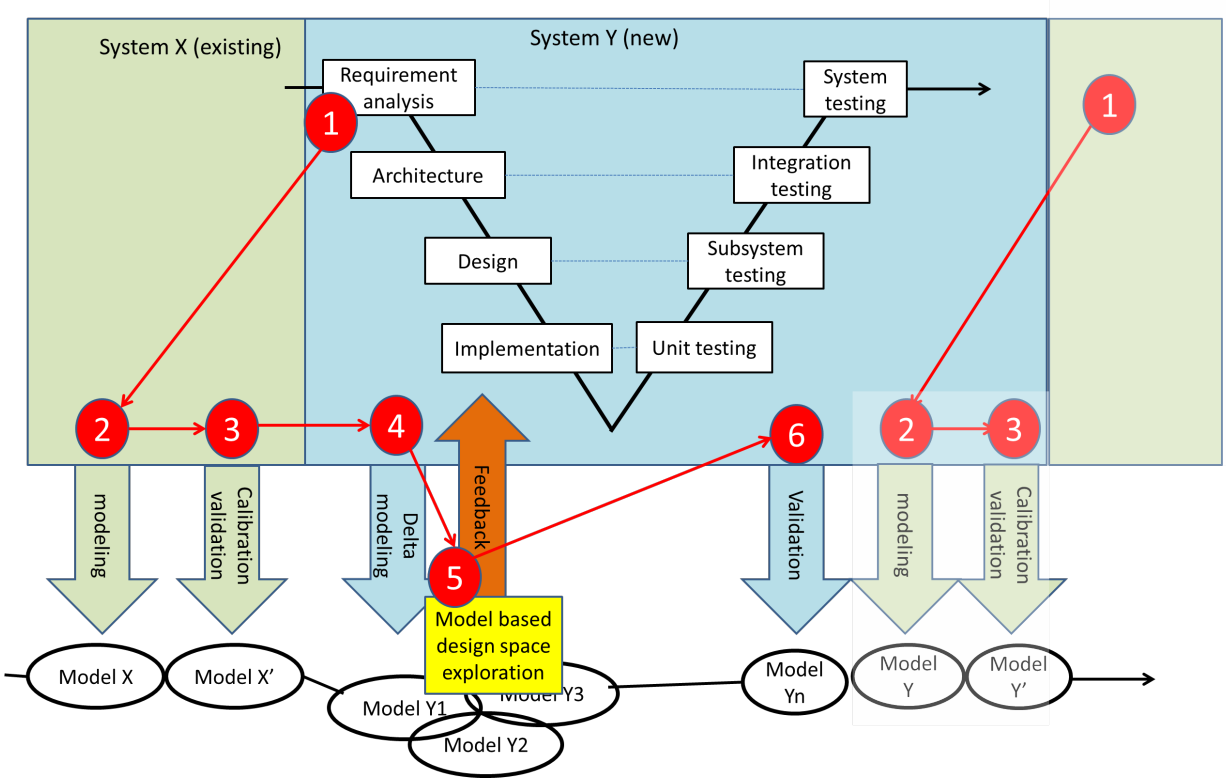


Figure 1: V-model development process [16]

1. **Requirement analysis**: Requirement analysis for a new system Y leads to a number of performance-related questions. Typically, identifying the bottleneck components of the existing system X with respect to performance metrics such as throughput is important for the overall development process.
2. **Predict the past**: An initial model of existing system X is built upon the initial performance related questions of step 1.
3. **Model calibration and validation**: The model of system X is calibrated and validated with respect to the requirements. This phase is performed by predicting the performance of the existing system X and comparing the prediction with the actual performance.
4. **Predictive models**: Based on the new requirements, certain changes are envisioned in the system X. The envisioned changes are incorporated in the calibrated and validated model in step 3. Thus, we obtain predictive models based on different design alternatives for system Y.

5. **Explore the future – model-based design space exploration**: We explore various design alternatives using model analysis based on the new model of system Y. The outcome of the design space exploration goes to the architecture and design steps.
6. **Implementation, validation and re-use**: After the system Y is realized, the predictive model of it can be validated using the actual realization. The validation allows for reconciliation of the model with reality. This completes the iteration. This model can be re-used for a new V-model development process with new requirements.

In this context, D1.3 introduces the models at three layers – component-layer, application-layer and multiple-applications layer. The models are obtained using the V-model development process by successive iterations over the above steps as illustrated in Figure 1. The application development environment envisioned in ALMARVI will utilize these models for characterization, optimization and trade-off analysis.

Table I summarizes our overall modeling approaches adopted in ALMARVI. Two levels of model abstractions:

- **Source code level:** the models derived based on the source codes running on certain computation platform, e.g., the experimental execution times of a code on a given platform.
- **Model level:** the higher level of abstraction based on a set of given source-code parameters, e.g., throughput analysis for a given task graph with execution times. Obviously, the parameters from the first category of modeling might be used in the second category.

Table I: models reported in D1.3 ALMARVI

Performance models		Abstraction		Validation		Tooling
		Source-code/ Model	Higher level	Analytical	Experimental	
Cross-layer	Multiple-applications -layer		TUE	TUE		SDF^3
	Application-layer	VF	TUE	TUE	VF	SDF^3+TRACE + Pareon tool
	Component-layer		TUE	TUE		SDF^3
Power models		**Abstraction**		**Validation**		**Tooling**
		Source-code/ Model	Higher level	Analytical	Experimental	
Cross-layer	Multiple-applications layer					
	Application-layer	VF			VF	Pareon tool
	Component layer				UTURKU	Spice
Error-resilience models		**Abstraction**		**Validation**		**Tooling**
		Source-code/ Model	Higher level	Analytical	Experimental	
Cross-layer	Multiple-applications layer					
	Application-layer		TUE	TUE		Matlab
	Component-layer	UTURKU			UTURKU	Spice

In view of Table I, Figure 2 illustrates the high level view on ALMARVI application development. DSE consists of **evaluating single design points** and **exploring the design space** of all possible design points.

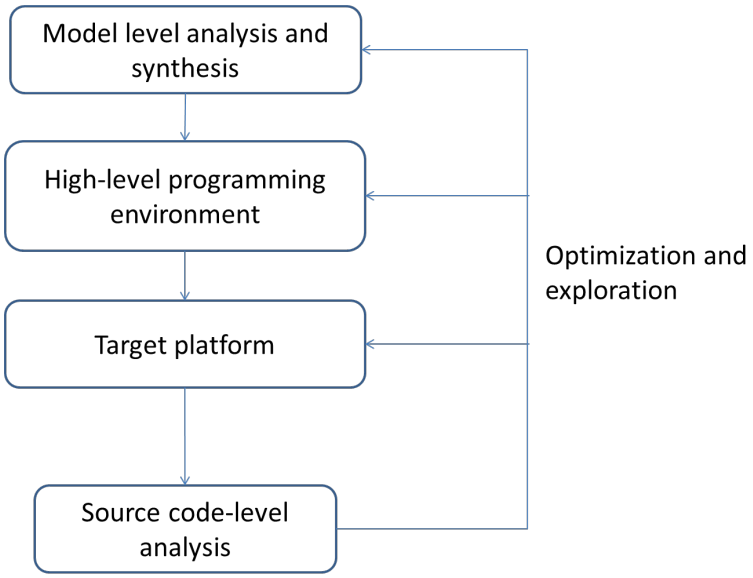


Figure 2: Application development flow

3.2 ALMARVI specific adaptation of development flow

Figure 3 provides the overview of ALAMRVI specific adapation of the development flow introduced in the previous section. The tools shown in Figure 3 are either used, developed, or extended by ALMARVI partners.

The presented DSE methodologies target timing analysis on multi-processors that share resources. In Figure 3, the bottom two boxes represents the existing state-of-the-art analysis tools and corresponding tool supports to explore certain design points (i.e., single point and design space) in terms models and implementation. The top left box in Figure 3 represents tools that deals with evaluation of single design points and targets single design point optimization, often manual in the current practice. The top right box in Figure 3 represents the possibilities of automated exploration of entire/a large part of the design space and targets optimization.

Major activities that will be reported in this deliverable deal with analysis and tool-support for evaluating single design point (i.e., top left box in Figure 3). A part of the reported activities utilized the state-of-the-art analysis and tools for design space exploration (i.e., bottom boxes in Figure 3) representing current industrial practices.

Further, a number of activities involving implementation of such optimized design points on target platforms (i.e., top and bottom left boxes in Figure 3) are also a part of ALMARVI and they are reported in D4.1 (Application Framework Control).

The development in the line of automated optimization of entire/a large part of the design space (i.e., top right box in Figure 3) is part of the future development since the state-of-the-art needs significant progress in terms of maturity along this direction for realizing them in ALMARVI context.

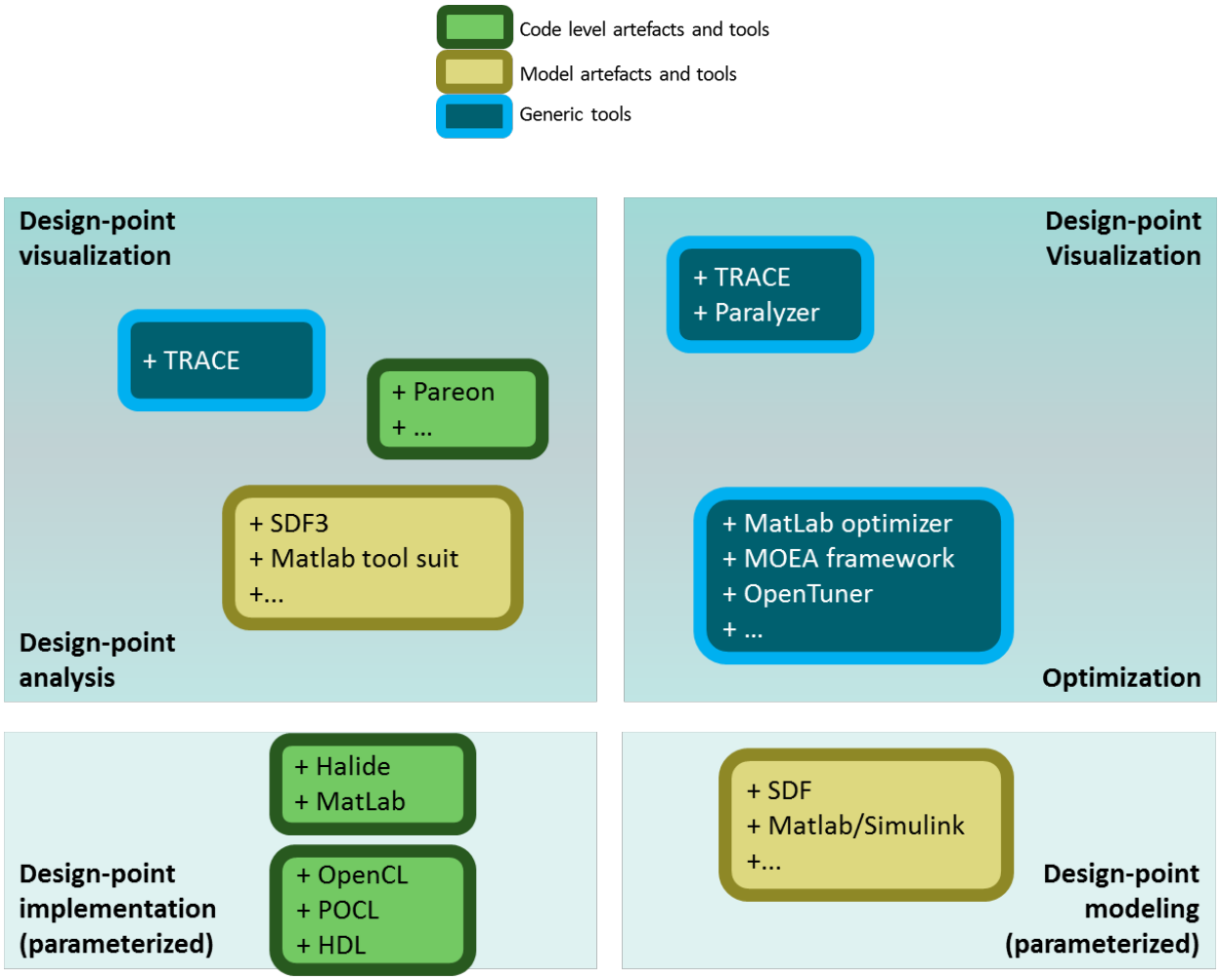


Figure 3: ALMARVI specific realization of application development flow shown in Figure 2

3.3 Relation to V-model in ALMARVI context

In what follows we describe how the ALMARVI development flow is related to the V-model process. The Figures 4a-4d show the links between steps 2 to 6 of the performance engineering approach as laid out in Figure 1, and the ALMARVI specific realization of the application development flow as shown in Figure 3. Note that the requirements step (step 1 in Figure 1) is not covered by tools and techniques that we report on; we assume that the performance-related questions are given.

The modeling steps in the performance engineering flow (steps 2 and 4) are accomplished using, e.g., the modeling formalism of Synchronous Dataflow (SDF) (see Figure 4a). Validation and calibration (steps 3 and 6) typically use models and implementations in order to calibrate model parameters to fit reality, and to evaluate the predictive power of the models (see Figure 4b). Next, we distinguish two ways of using the predictive models (or prototype implementations) to give feedback to the development process (step 5 in Figure 1). First, when the design space is manageable, all choices can be evaluated manually (or with a little bit of automation). E.g., the investigation of “what-if” questions such as “What happens when we add an additional processing step with these resource requirements?” falls into this category. We then typically consider a handful of design alternatives (we vary the estimated resource

requirements a bit in order to determine the sensitivity). We can thus typically do an exhaustive analysis by hand using our tools and do not need optimization libraries to search the design space (Figure 4c). Second, we consider optimization of all kinds of design parameters including application parameters such as buffer sizes and multiplicity of software components, platform parameters such as CPU type, and mapping parameters (which software task on which piece of hardware). The number of combinations grows exponentially, and exhaustive analysis quickly becomes impossible. In this case we use optimization libraries to find good solutions in the design space automatically (Figure 4d).

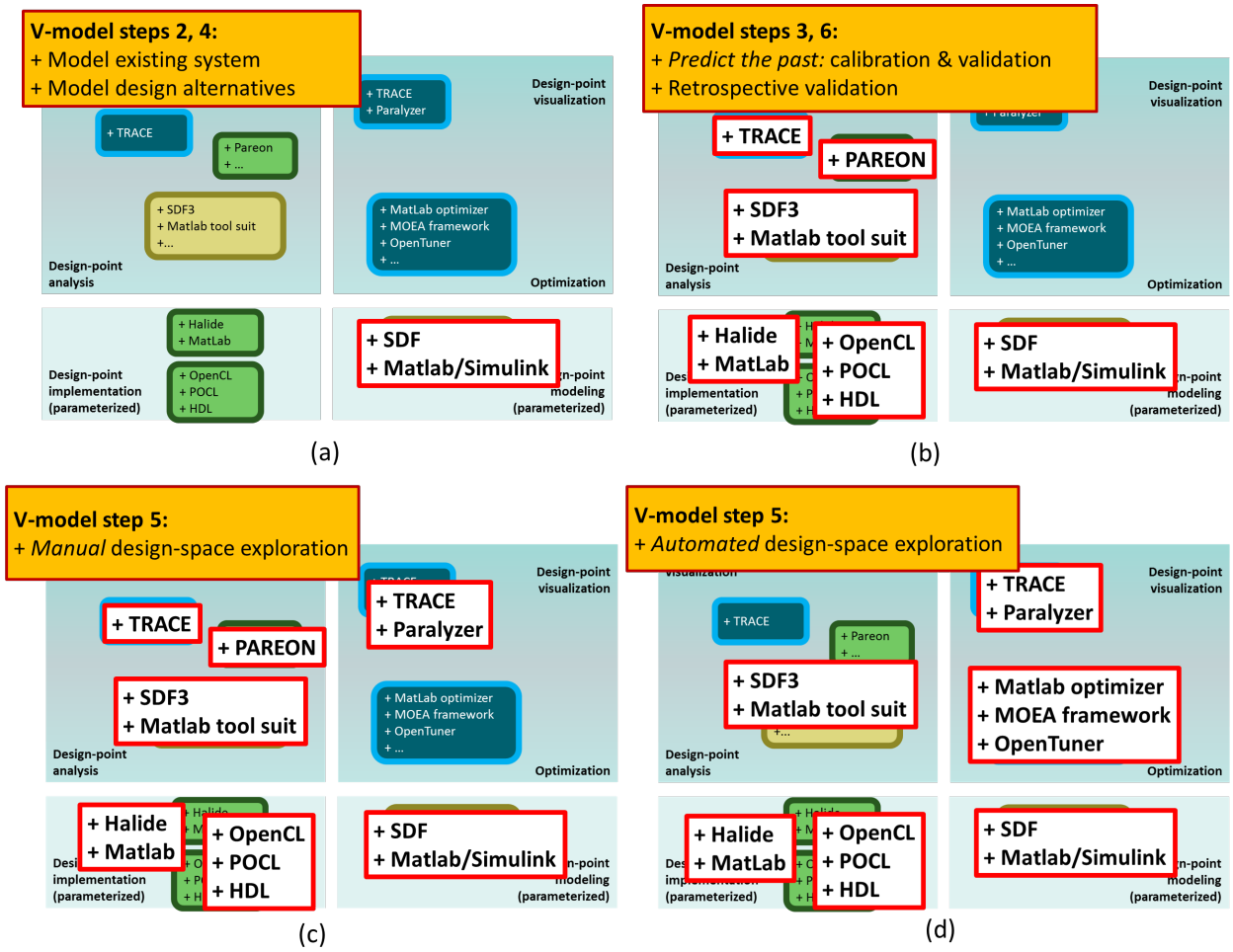


Figure 4: ALMARVI specific flow in view of V-model development process shown in Figure 1

3.4 Organization

As shown in Table I (from D1.3), the analysis methods are further classified based on nature of the target level: component, single-application and multiple-applications. The reported activities in this deliverable mainly target single application level while many of them are equally applicable or extendable to multiple-applications level. Generally, multiple-applications methodologies for resource allocation are reported in D1.3 for feedback control, streaming applications and combination for multiple of them. This deliverable is organized based on the two main research ingredients: Model level analysis (Chapter 4) and Source code level analysis (Chapter 5).

4 Model-based analysis

This chapter details refinement and improvement of the models at different layers reported in D1.3 and how they are used for performance analysis and design space exploration. Further, models of resource usage, timing behavior, power usage, error resilience, and performance are utilized to optimize the implementation of an applications. Various tool support used in this stage will be illustrated.

- **Section 4.1 (Analysis and application mapping on shared resource) –** This section will deal with modeling and analysis of mapping problem of a feedback control application on multi-processors. Modeling and timing analysis are performed to find the bound on deadline misses for a given resource allocation. Analysis method is aims single design point evaluation and optimization for a control application.
- **Section 4.2 (Tighter temporal bounds for dataflow applications mapped onto shared resources) –** This section reports a tighter analysis method for mapping problem streaming applications onto multi-processors. Modeling and analysis deals single design point evaluation of a steaming application.
- **Section 4.3 (Trace based analysis) –** This section reports the visualization of time-stamped execution traces obtained from model-driven methods (e.g., analysis in Section 4.1 & 4.2). An earlier version of TRACE tool [10] for visualization and analysis of execution traces was reported in D1.3. Under this deliverable, the TRACE is further extended with a well-defined syntax and semantics to enables the specification of a wide variety of quantitative real-time properties.

As already stated in Chapter 3, the main effort is to enhance the state-of-the-art analysis, evaluation and visualization of single design point as shown in Figure 5.

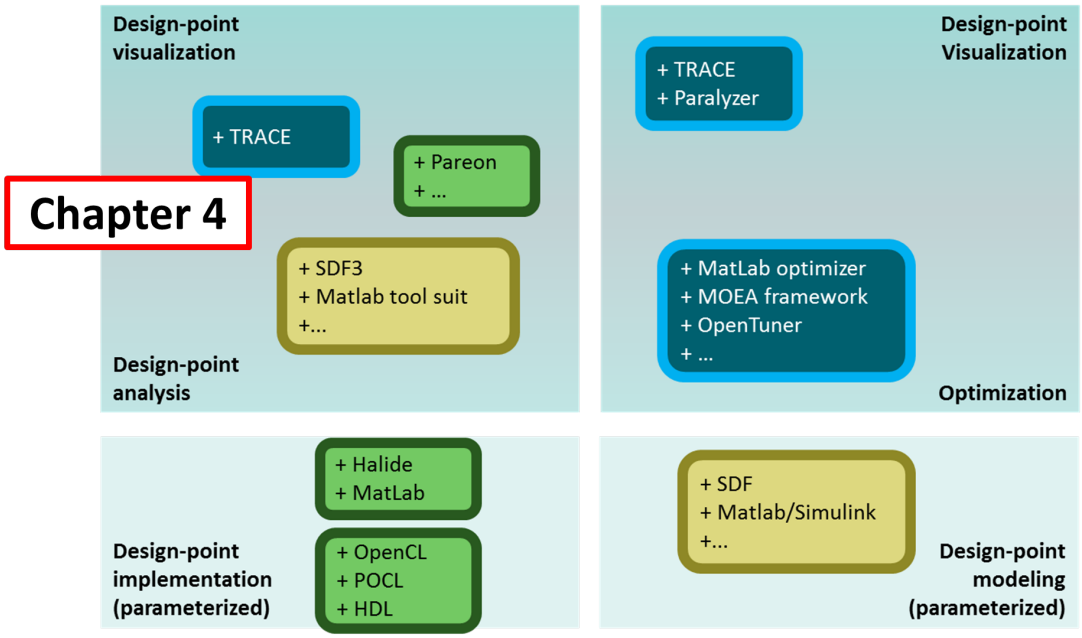


Figure 5: Chapter 4 overview

4.1 Analysis and application mapping on shared resources

4.1.1 Motivation and Objectives

This section focuses on the mapping problem of a feedback control applications onto multi-processors – modelling, analysis and evaluation of single design point. Generally, many application domains, including healthcare and automotive, require to run several applications simultaneously. Sharing resources among applications is a widely used trend towards a cost-efficient product development. This imposes new challenges in hardware and software design. Application interference on a shared resource is a potential issue. A *Budget scheduler* provides temporal predictability on a shared resource by guaranteeing a fixed access time for every scheduled application [1]. Time Division Multiple Access (TDMA) is a common scheduling policy for realizing temporal predictability for such applications [2] [3]. It allocates identical constant time slots to applications in a work cycle.

Due to the safety-critical nature of control applications, timing plays a key role in guaranteeing their Quality of Control (QoC) [2]. Running control applications on a shared processor for computation reasons can cause control samples to miss the computational deadline. This affects QoC. A sample should be processed before the next sample arrives and therefore, each sample has a computational deadline equals to the sampling period of the application. The samples with missed deadlines are referred to as Dropped Samples (DSs). A potential reason for a sample to miss a deadline can be that sufficient resources are not available for the application when the sample is ready for processing.

Under the *(m,k)*-firmness condition [4] a control application can still satisfy QoC requirements in presence of DSs. That is, at least *m* samples out of *K* consecutive samples must meet the computational deadline to satisfy application level requirements. In other words, *k-m* samples out of *k* consecutive samples can miss the computational deadline without violating the requirements.

We consider control applications running on a shared processor under a TDMA policy. We are particularly interested in a range of sampling periods that lies in between the best and the worst case response time of the control task. For such a range of sampling periods, a certain *(m,k)*-firmness condition is given for each control application. We aim to formally verify the satisfaction of such condition and in effect, guarantee QoC. We propose an analytic method to quantify the number of DSs. Verifying the number of DSs for a finite window of arrival of samples, we propose a method to obtain the maximum number of DSs.

4.1.2 Method

We consider a situation in which a control application is running on a processor with a TDMA schedule. We investigate the operation of a processor in a time interval which includes the arrival times of *k* consecutive samples. Therefore, we define a relative time based on a specific time wheel in which the first sample out of *K* consecutive samples arrive. Let us consider the start time of the first time wheel as time *t=0*. Then, the relative arrival time of all samples are obtainable, since the sample arrivals are separated by the sampling period *h*. Figure illustrates the first time wheel in several repetitive executions of a TDMA time wheel.

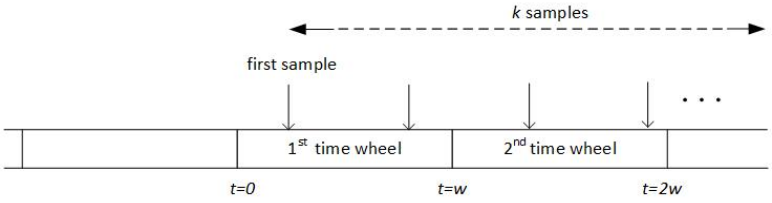


Figure 6 Relative position of the first time wheel and the first control sample in a repetitive execution of a TDMA-scheduled processor assigned to a control application

From the explanation above it is concluded that any sample arrived to the processor has a deadline equal to the sampling period *h*. Therefore, available resource for the control application is assessed in a time interval between *t* and *t+h*. We represent this by *Resource Availability Function* (RAF), *g(t)* such that

$$g(t) = \int_t^{t+h} f(\tau) d\tau$$

where *f(t)* is Allocated-Time Function (ATF) which takes 1 if the processor is allocated to the application at time *t* and zero otherwise. Figure a illustrates ATF of a control application with a sampling period of *h=700µs* and execution time of 270µs, for which 10 consecutive sample are considered to verify the maximum possible number of DSs. This application is assumed to be run under a TDMA schedule with a time wheel size of *w=550µs*. In any time wheel the slices (110µs -210µs) and (330µs -430µs) are allocated to the application. RAF for this application is shown on Figure 7b.

A sample, arrived at time t_j, misses the computational deadline if $e>g(t_j)$ where *e* is the execution time of the application. The horizontal line on Figure 7b shows the execution time of the application. Then a TDMA time wheel can be split into two types of intervals: 1) *miss zone* intervals in which $e \geq g(t_j)$ and any sample arriving at this interval well miss the computational deadline; 2) *hit zone* intervals in which $e<g(t_j)$ and any sample arriving at this interval will meet the computational deadline. These two intervals are specified by Miss Zone Function (MZF) *z(t)* such that

$$z(t) = \begin{cases} 1 & e \geq g(t) \\ 0 & e < g(t) \end{cases}$$

Let us consider a function that represents each control sample by a Dirac delta function *δ(t)* such that the first sample arrives at time *t=0*. We name the function as Control Sample Distribution Function (CSDF) such that

$$r(t) = \sum_{n=0}^{k-1} \delta(t - nh)$$

In view of Equation above, the number of DSs is represented by function *s(t)* such that

$$s(t) = \int_0^{\infty} z(\alpha) r(\alpha - t) d\alpha$$

where *t* is the arrival time of the first sample out of k consecutive samples. Figure 7c and Figure 7d show MZF and number of DSs for the example above. It can be shown the *s(t)* is periodic with a period of time wheel size *w*. Then the absolute maximum number of the DSs is obtained by verification of one period of *s(t)*. That is $s_{\max} = \max_{0 \leq t < w} (s(t))$. It can also be shown that any increase in the value of *s(t)* happens when at least one of the samples arrives at the start time of a miss-zone. Therefore if we obtain the position of the first sample in the first time wheel for

all the times that a sample arrives at the beginning of a miss zone we can guarantee to verify the

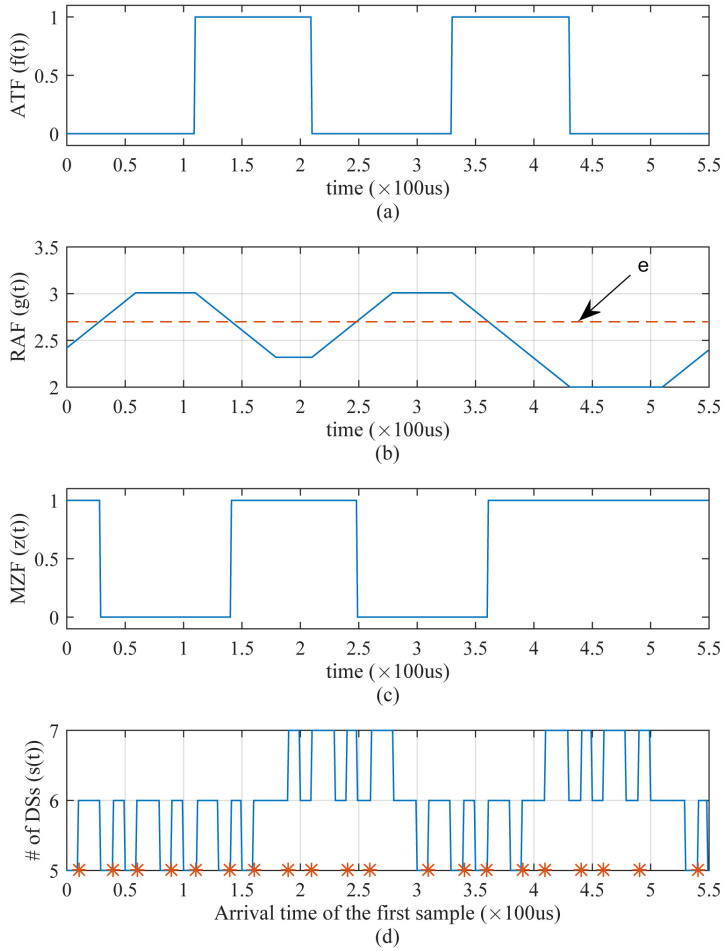


Figure 7 A TDMA time wheel with two slices allocated to a control application.

maximum possible number of DSs. In Figure 7d these points are shown by the red stars. The figure confirm over method by showing that all stars are located in the points that there are increase in the value of *s(t)*.

Then we conclude that for DSs quantification of a given control application mapped on a TDMA-scheduled processor we need to first find *s(t)* as explain above and then verify *s(t)* for finite points of time which are obtained by

$$t_{inc_i} = \begin{cases} t_{mf_i} - \mathrm{mod}(n \times h, w) & t_{mf_i} \geq \mathrm{mod}(n \times h, w) \\ w + t_{mf_i} - \mathrm{mod}(n \times h, w) & t_{mf_i} < \mathrm{mod}(n \times h, w) \end{cases}$$

where $n \in \{0,1,2,\ldots k-1\}$, t_{inci} is the start time of a miss zone in the first time wheel and

$$\mathrm{mod}(x, y) = y - x \times \left\lfloor \frac{y}{x} \right\rfloor.$$

4.1.3 Evaluation

In this section, we explain our experimental results of applying the proposed method on a realistic case. For illustration of the applicability of our proposed methods, we consider a control application with sampling period of 2ms. We consider a window of k = 125 consecutive samples in our experiments. Based on the specifications of the platform on which the application is running, execution time of the control task is determined. We took different sets of platform-related settings to verify the *(m,k)*-firmness properties of each. The FP method was implemented in MATLAB and compiled in a computer with a quad-core processor and a clock frequency of 2.6GHz. The same system was used to run the UPPAAL model.

Table 2 shows the settings and results of our experiments. In this table *w* indicates the size of the TDMA time wheel, *e* denotes execution time of the control application, and t_{ver} shows the verification time. The last column of Table 2 shows the maximum number of DSs for each case.

Table 2 Different sets of platform settings and verification results using Finite-point method for k=125 consecutive

W	Allocation interval (us)	e	t_{ver}	Max # of DSs
1.3ms	[0,80], [440,520], [870,950]	400us	225us	58
1.3ms	[0,175], [870,1000]	400us	327us	10
700us	[0,250]	600us	332us	125
700us	[0,250]	500us	352us	54

Figure 8 depicts maximum number of the DSs against sampling period for the first set of settings in Table 2. From classical response time analysis it can be verified that a sampling period shorter than 1.77ms will result in all DSs while a sampling period longer than 2.135ms is enough to meet all the deadlines. The range of sampling periods between the above values gives different number of DSs as shown in Figure 8 . For a given platform settings, this analysis can be used to choose a suitable sampling period to meet a *(m,k)*-firmness bound (hence, to meet the QoC requirement). That is, considering *(m,k)*-firmness properties, we can reduce the sampling period to a value less than 2.135ms without allocating more resource to the application. Besides, we can reduce the allocated resource instead of changing the sampling period to have a recourse efficient allocation. In the first set of settings in Table 2, for example, considering a sampling period of 2.135ms which results no DSs, we can reduce 11% of the length of each allocated slice, i.e. 33% less resource allocated, to have 25 DSs.

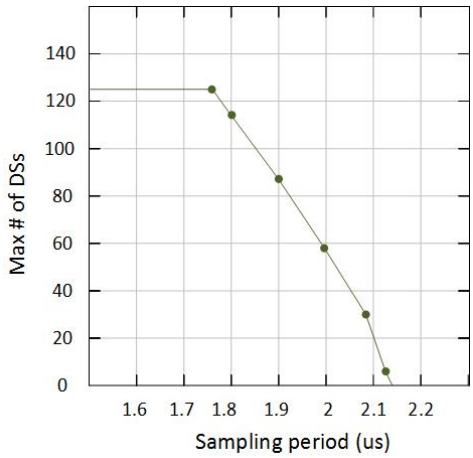


Figure 8 Maximum number of the DSs against sampling period for the case in the first row of Table 2

4.2 Tighter temporal bounds for dataflow applications mapped onto shared resources

4.2.1 Motivation and Objectives

This section focuses on modelling, analysis and visualization of mapping problem of a streaming application (application-level analysis) onto multi-processors. The presented method deals with single design point analysis and evaluation. Generally, embedded streaming applications such as video or image processing algorithms can be realized on shared platforms for cost and power reasons. These applications have real-time constraints regarding latency or throughput. One of the most important steps in DSE of embedded applications on shared platforms is allocating enough resources to these applications to *guarantee* their real-time constraints.

Often resource allocation strategies have an iterative process in which they initially allocate resources, they analyse the temporal behaviour of the system and then they adjust resource allocation parameters based on the analysis results [5]. The temporal analysis is one of the core parts of such algorithms and since it is a part of an iterative process, it should be fast enough to make the whole allocation process practical. Sharing resources introduces uncertainties (non-determinism) to the temporal behaviour of the applications depending on the scheduling policy. For example when sharing a resource by a Time Division Multiple Access (TDMA), clock drifts cause uncertainties in the relative position of the allocated time slots which in turn causes uncertainties in the response times of the tasks. To guarantee that the allocated resources make an application meet its constraints, we need to obtain conservative, but tight, temporal bounds on the worst case behaviour of the system (taking into account the uncertainties) in a reasonable time. We need the bounds to be tight in order to avoid over-allocation of resources.

One of the popular methods for the temporal analysis of applications is the Synchronous Dataflow Graph (SDFG) [6] model of computation (an example is shown in **Error! Reference source not found.** 9). This model represents the application by a graph in which the nodes (*actors*) represent the tasks within the application and the directed edges (*channels*) model the dependencies between them. The tasks start their execution, i.e. the actors *fire* whenever they have enough data in their input channels, then take a certain time to execute and produce data in their output channels. The

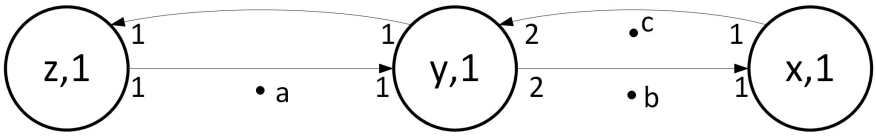


Figure 9 An example SDFG

presence of data on channels is represented by *tokens*. By firing of an actor, a fixed amount of data i.e. a fixed number of tokens is produced on and consumed from each of the output and input channels respectively, which is determined by the channel *rates*. An actor is said to be enabled if the number of tokens on each of its input channels is not smaller than the consumption rate of the channel. The least non-empty set of actor firings that returns the graph to its initial token placement is called an *iteration*.

According to [7], the timing behaviour of an SDFG can be captured by finding the *time differences* between the production times of tokens at the end of the iteration and the availability time of the initial tokens. This is done by *symbolically simulating* the application graph. Symbolic simulation considers the *symbolic time stamps* of produced tokens rather than only times; this way it captures the time differences between the production times of the tokens and each of the initial tokens.

In this work, we assume the resource is shared by budget schedulers, which allows us to determine independent time bounds for applications. A budget scheduler guarantees the application a minimum amount of budget (processing time) over a periodic time frame called the replenishment interval. The challenge is that in this case the exact response times of tasks cannot be determined because the precise state of the scheduler is not known when the task is able to start its execution. For example, the start of a task might be at a time instance where the whole budget allocated to the application is used for the current scheduling period, and the task has to wait for the next replenishment interval (worst case), or the task might immediately start working because it has arrived at the start of the allocated budget (best case). The actual response time can be anywhere between the best and the worst case. Although it is possible to obtain conservative bounds using the worst case response times in the symbolic time stamps, the obtained bounds are too pessimistic. In this work we present an analysis method to provide tighter temporal bounds for applications modelled by Synchronous Data Flow Graphs and mapped onto shared resources.

4.2.2 Method

We exploit the fact that the worst case response time assumption can be avoided for sequences of consecutive task executions on the same resource. We propose a new method to better detect consecutive executions. Then we use WCRCs to find the accumulated worst case response times of the consecutive tasks, which is less pessimistic.

Following [8], a budget scheduler can be abstracted by a Worst Case Resource Curve (WCRC). This curve specifics the minimum amount of service allocated to the application in any time interval of time. Using this curve we can extract Worst Case Response Time (WCRT) of firings. **Error! Reference source not found.** 10 shows a TDMA scheduler and its corresponding WCRC ζ. The WCRC considers the worst case positioning of firing start times with respect to the allocated slots. For example, let tuple (p, k) indicate the kth firing on processor p. Assume this firing corresponds to actor x with execution time of 1 time unit. The worst case positioning for

the start of firing of actor x on a processor shared by the example TDMA, is shown in **Error! Reference source not found.** 10. In this situation, the actor has to wait 2 time units to start processing at the next allocated slot; hence it is complete within 3 time units. Therefore the WCRT of this actor firing is 3. Now assume the next firing i.e. $(p, k+1)$ corresponds to actor y with execution time of 1 time unit. If we know that $(p, k+1)$ will be able to start no later than (p, k) completes, we can use the accumulated worst case response times i.e. we can make sure that the completion of both firings will not take more than 4 time units as shown in the same figure. When this observation is not exploited, the completion of both firings is estimated to take 6 time units in the worst case which is too pessimistic.

Next, we provide a method to identify consecutive task executions during the execution of the application. We can find the consecutive task executions, if we find all dependencies between firings for the firings involved in one iteration of the application graph. This enables us to separately capture all possible dependency paths that connect the completion time of firings to all initial dependencies. Then for each dependency path we can separately decide which firings

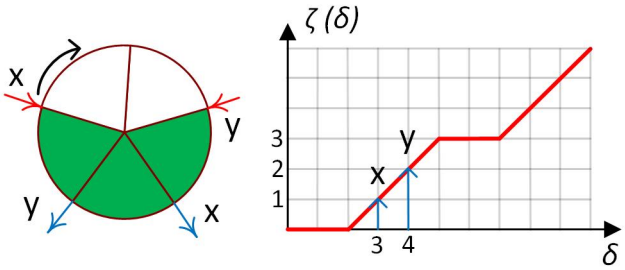


Figure 10 An example TDMA and its WCRC

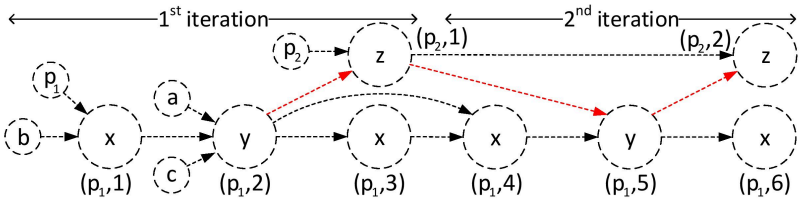


Figure 11 The dependency graph of example SDFG

are consecutive in it.

During symbolic simulation, for each token that is produced by firings, in addition to the symbolic time stamp, we add extra information regarding the firing that produced them. By keeping track of the tokens produced and consumed by firings, we can extract the dependency graph of the firings. Figure 11 shows the dependency graph associated with the execution of the example SDFG for two iterations. It is obtained by simulating the graph and finding the firing dependencies of each firing during the simulation. In this graph, the nodes indicate the firings. A directed edge from (p', k') to (p, k) indicates that (p, k) is dependent on (p', k'). The black edges indicate the firing dependencies on the same processor and the red edges indicate the dependencies on different processors. Using this graph we can track the dependencies of each firing back to the initial dependencies and separately compute the time difference between them. The key point is that if two nodes are connected only by black edges,

then the time difference between them is equal to the accumulated worst case response times of all firings

between them including the last node. Note that if there is more than one path between two nodes, then the time difference is equal to the maximum of time differences in all paths. We have implemented an algorithm that builds the graph and finds all consecutive requests during the symbolic simulation of the application. This algorithm first constructs the dependency graph. Starting from node (p,k) , it connects all nodes representing the firings in the dependency set of (p,k) to this node. Then, the same action is taken for each node in the dependency set of (p,k) only if it represents a firing on the same processor. This process continues until all source nodes of the graph (the ones without input edges) are either representing initial dependencies or firings on other processors. Then, the symbolic completion time of the firing is obtained by adding the maximum time difference of all paths that connect the source nodes to (p,k) i.e. the accumulated response time of all firings in the path, to the symbolic completion time of firings represented by source nodes and taking maximum of all.

4.2.3 Evaluation

We have implemented our temporal analysis method in the SDF3 tool [9]. We compared the throughput (1/cycle time) lower bound obtained by our approach with the state of the art analysis of [8] for three real life applications: H.263 encoder, H.263 decoder and sample rate converter, all available in the SDF3 tool. For each application, we used SDF3 to map it to a

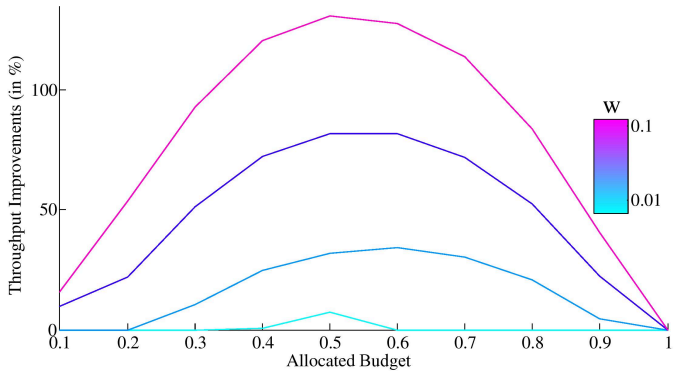


Figure 12 Lower-bound improvements for throughput

multiprocessor platform with four processors such that the total work load is evenly distributed between processors as much as possible. We limit the replenishment intervals to $0.01\times C_f \leq w \leq 0.1\times C_f$ where C_f is the cycle time of the application when all processors are fully allocated to the application. Large replenishment intervals cause huge delays in execution of the application, which is not desired; small replenishment intervals are less useful because of the context switch overhead. Figure 12 shows the average relative improvements in the throughput lower bound of applications for different replenishment intervals and allocated budgets. As shown in the figure, the improvement ratio decreases when the application gets smaller or larger processor shares. In these cases using the accumulated worst case response times does not have much improvements over WCRCs. The average analysis run-time for the mentioned applications in a standard computer is 320 milliseconds which is 17% longer compared to [8]. It is still in the practical range.

4.3 Trace based analysis

4.3.1 Motivation and Objectives

A wide range of the component, application and multi-applications level analysis methods can be further evaluated and validated using time-stamped execution traces, e.g., analysis presented in Section 4.1 and 4.2. Execution traces are sequences of time-stamped start and end events of system activities and form a generic way to represent dynamic system behavior. ALMARVI deliverable D1.3 has introduced the TRACE tool [10] for visualization and analysis of execution traces. In this section we report on an extension of the TRACE analysis capabilities, namely the capability to check specifications in the form of temporal logic formulas.

It is our observation that in practice many interpretations of performance-related metrics and terms such as “latency”, “throughput”, “jitter”, “pipeline depth” etcetera exist. The exact meaning of requirements such as “the throughput must be at least 25 images per second with a jitter of 50 milliseconds” is therefore not completely clear and may vary, even within a domain. Formalisms for property specification with a well-defined syntax and semantics can alleviate this problem. Metric Temporal Logic (MTL) [11] enables the specification of a wide variety of quantitative real-time properties for time-stamped event sequences such as execution traces.

4.3.1 Metric temporal logic

We assume the context of a set of states S, a set of atomic propositions $\boldsymbol{AP}$, and a labeling function $l : S \to 2^{\boldsymbol{AP}}$ that assigns to a state $s \in S$ the atomic propositions that are true in that state. MTL formulas are interpreted over *timed traces* which are possibly infinite time-stamped event sequences. These consist of state-time tuples, i.e., $(s_0, t_0), (s_1, t_1), (s_2, t_2), \cdots$, where $s_i \in S$ is a state and $t_i \in \mathbb{R}$ is a time stamp. These definitions give use the means to define the syntax and semantics of MTL formulas. The syntax is inductively defined as follows:

$$\phi = true \mid p \mid \phi \wedge \phi \mid \neg\phi \mid \phi \, \boldsymbol{U}_I \phi$$

where $p \in \boldsymbol{AP}$ and $I \subseteq [0, \infty)$ is a convex interval (open, closed or half open) on $\mathbb{R}$. The semantics is inductively defined as follows. Let $\rho = (s_0, t_0), (s_1, t_1), (s_2, t_2), \cdots$ be an infinite timed trace, and let $\rho_i = (s_i, t_i)$. Then:

- $\rho_i \vDash true$
- $\rho_i \vDash p$ if $p \in l(s_i)$
- $\rho_i \vDash \phi_1 \wedge \phi_2$ if $\rho_i \vDash \phi_1$and $\rho_i \vDash \phi_2$
- $\rho_i \vDash \neg\phi$ if $\rho_i \nvDash \phi$
- $\rho_i \vDash \phi_1 \boldsymbol{U}_I \phi_2$ if some $j \geq i$ exists such that $\rho_j \vDash \phi_2$ and $t_j - t_i \in I$ and $\rho_k \vDash \phi_1$for all $i \leq k < j$.

We say that ρ satisfies an MTL formula ϕ, denoted by $\rho \vDash \phi$, if $\rho_0 \vDash \phi$. Some useful abbreviates are:

- Finally: $\boldsymbol{F}_I \phi \triangleq true \boldsymbol{U}_I \phi$
- Globally: $\boldsymbol{G}_I \phi \triangleq \neg \boldsymbol{F}_I \neg\phi$

We omit the trivial interval $[0, \infty)$from our notation. The semantics can be defined for finite traces by restricting the scope of the existential quantifier in case of the until operator to the length of the trace.

4.3.2 Examples

We consider a pipelined processing system consisting of seven tasks, A – G, that work on a stream of input objects. Through the use of the OctoSim discrete-event simulator [12] we have access to finite timed traces of this system. For instance, Figure 13 shows a Gantt-chart representation, using the Trace tool, for processing of 10 input objects. The x-axis shows the time, and the rows on the y-axis shows the different activities. The color indicates the object that is processed. Atomic propositions in this setting are of the form *N*, *N(v)* or *N(v,i)* , where *N* is the name of the task, *v* is either *s* or *e* and indicates whether it is the start event of the task or the end event, and *i* is the object number. For instance, *G(e,0)* specifies the end of task G for the first object in the stream. A number of useful MTL properties that can be used to analyze timed traces of the system for, e.g., 1000 input objects, are shown below.

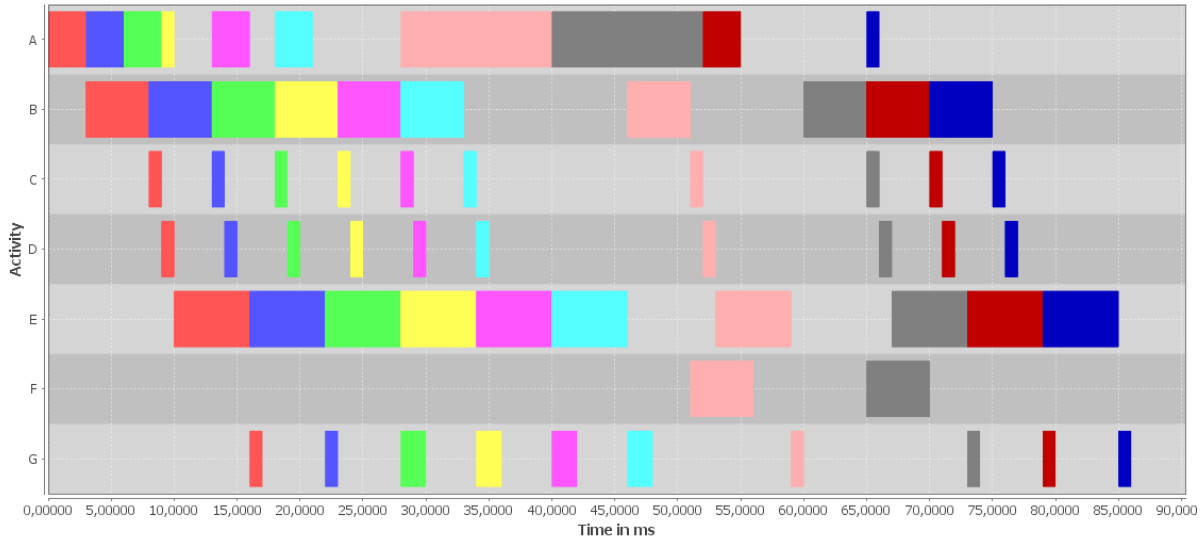


Figure 13: A TRACE view of the example system in which 10 objects (indicated by color) are processed.

1. The first property formalizes that the first object (with id 0) has been completely processed within 25 time units: $\boldsymbol{F}_{[0,25]}G(e,0)$.
2. The second property formalizes that the total execution time is at most 6500 time units: $\boldsymbol{F}_{[0,6500]}G(e,999)$.
3. The third property formalizes that the per object processing time is at most 70 time units: $\bigwedge_{i=0}^{999}\boldsymbol{G}\,(A(s,i) \Rightarrow \boldsymbol{F}_{[0,70]}G(e,i))$.
4. The fourth property formalizes that the throughput is at least 10/65 in every window of 10 consecutive end events of task *G*: $\bigwedge_{i=0}^{989}\boldsymbol{G}\,(G(e,i) \Rightarrow \boldsymbol{F}_{[0,65]}G(e,i+10))$.
5. The fifth property formalizes that the throughput equals 1/10 objects per time unit with a jitter of 5 time units: $\bigwedge_{i=0}^{999}\boldsymbol{G}\,(G(e,0) \Rightarrow \boldsymbol{F}_{[i\cdot 10-2.5, i\cdot 10+2.5\,]}G(e,i))$.
6. The sixth property formalizes that after any end event of task *G*, another end event of task G happens within 3 and 15 time units: $\boldsymbol{G}(G(e) \Rightarrow \boldsymbol{F}_{[3,15]}G(e))$.

These examples illustrate the flexibility and expressive power of MTL. The formalism allows us to define what we exactly mean with, e.g., pipeline depth, buffer occupancy, latency and throughput.

4.3.3 Good, neutral, bad and informative prefixes

We often have access to finite execution traces of some system. These traces can be obtained from a real system, but also, for instance, from a discrete-event simulation model. We distinguish two situations: (i) the trace represents the full execution of some process, or (ii) the trace is a prefix of some ongoing, possibly infinite, process. An example of the first situation is the execution trace of an image processing pipeline that processes 10 images and then is done. In this case, we can apply the MTL semantics for finite traces. An example of the second situation is a part of an execution obtained from a running web server. In this case, however, application of the finite MTL semantics is not appropriate, because there is an unknown extension of the trace that can affect the truth value of the property. For this situation, we have adopted the notion of informative prefixes [13].

Consider a finite prefix ρ of some timed trace and an MTL formula ϕ. Then we say that ρ is a bad prefix if and only if every extension of ρ dissatisfies ϕ. Dually, ρ is a good prefix if and only if every extension of ρ satisfies ϕ. A neutral prefix is neither good nor bad. Intuitively, an informative prefix tells the whole story about the (dis)satisfaction of an MTL formula [13]. For instance, the prefix *(p,0),(p,1),(p,2),(q,3)* is bad for $\boldsymbol{G}p$ and it is also informative. The prefix is also bad for $\boldsymbol{F}p \wedge \neg p$, but not informative because the dissatisfaction for any extension depends on the unsatisfiability of $p \wedge \neg p$. This information is not to be found in the prefix itself.

We have followed the approach of [14] to define strong and weak satisfaction relations for MTL formulas and timed traces, and have devised a recursive memoization algorithm that can check whether a prefix is informative good, informative bad or neither of those. The algorithm scales to large traces and can generate concise explanations of the truth value of the given MTL formula. For details of our approach we refer to [15].

4.3.4 Implementation in the TRACE tool

Figure 14 shows the Eclipse IDE with the TRACE plugin installed. The window has (1) a project explorer view of the files in the workspace, (2) a number of Trace toolbar items, (3) the main Gantt-chart view, (4) the MTL explanation view, and (5) a concrete explanation of the property being analyzed overlayed on the Gantt-chart view. In this case, the Gantt chart visualizes a (part of a) run of the system from the example above for 1000 objects while the 6th example property is being analyzed. The project explorer associates files with an mtl extension with the MTL dialog. Double clicking an mtl file when a trace is open, opens the MTL dialog with the contents of the mtl file. The MTL dialog has several configuration options: (i) whether to apply it to the set of filtered claims or to the whole set of claims, (ii) whether to interpret the trace as a prefix or not, (iii) whether to generate explanations of computed values. If the OK button of the MTL dialog is pressed, the MTL specification is checked against the current trace.

We generate explanations in two forms. First, the claims that are relevant for the truth value of the formula can be highlighted. This is a rather straightforward visualization based on a marking of states and their claims during the run of the algorithm. Nevertheless, it is often very useful and allows us to zoom into relevant parts of the trace quickly for diagnosis. The second form consists of an annotation of (part of) the time axis with the truth values of all subformulas of the formula that is checked. Also this annotation is constructed on-the-fly during the run of the algorithm. This annotation allows the user to trace the result according to the semantics. Figure 14 shows the user interface after checking the 6th example property and after visualizing the second type of explanation. Below the time axis are the three subformulas of the implication. A

red bar means that the property is not satisfied in any state in that time interval, and a green bar means that it is satisfied. A blue bar indicates that the property may or may not be satisfied by an arbitrary extension of this prefix. For this property, the key is that the implication $G(e) \Rightarrow \boldsymbol{F}_{[3,15]}G(e)$ holds for every end event of G but the last one. However, an extension of the trace could have more end events of task G within the indicated interval. Therefore, $\boldsymbol{F}_{[3,15]}G(e)$ may or may not be satisfied by the last state in the prefix, hence the blue marking.

Figure 14: A screenshot of the TRACE tooling in the Eclipse IDE.

4.4 Conclusions

This chapter presented a number of single design evaluation and visualization methods based high-level abstraction. On the one hand, such high-level models and analysis provide a solid basis for evaluating design points. On the other hand, the essence of these models and analysis results depends on the further refinement using implementation numbers coming from a specific target platform and corresponding implementation. This necessitates need for source code level analysis – Chapter 5. Ideally, the implementation numbers should be fed back to the models for their refinement and the development evolves iteratively following the V-model illustrated in Figure 1.

5 Source code level analysis

This chapter deals with source code targeting a specific platform for further analysis and DSE. The analysis numbers obtained at this stage are further used in refining the higher level models and closing the gap between models and implementation. Interaction/iteration over model and source code level analysis follows the V-model process as illustrated in Figure 1.

- **Section 5.1 (Pareon for design-point evaluation and trace visualization support) –** This section focuses on source code level analysis of implementation of compute intensive image/video processing algorithms targeting multi-core architectures. The presented work mainly deal with single design point evaluation and visualization at the application level. The focus on the enhancement of the existing tooling support (e.g., Pareon) for analyzing an implementation. These numbers may be used for model level analysis techniques and visualization (e.g., methods reported Chapter 4) and further, they relevant for settings with shared resources.
- **Section 5.2 (Floating-point to fixed-point Design Report C++ to FPGA conversion) –** This section describes the analysis and implementation method for healthcare images onto fixed point FPGAs. The challenge is to C++ code generated from Matlab uses floating-point while target FPGAs implementations using various HDLs uses fixed-point. Analyzing the efficiency and correctness of the above conversion is mainly performed by state-of-the-art methods and tool support. This is a representative of today's industrial DSE dealing with source code level modeling and analysis.

With respect to overall development process introduced in Chapter 3, the focus of the presented works is shown in Figure 15.

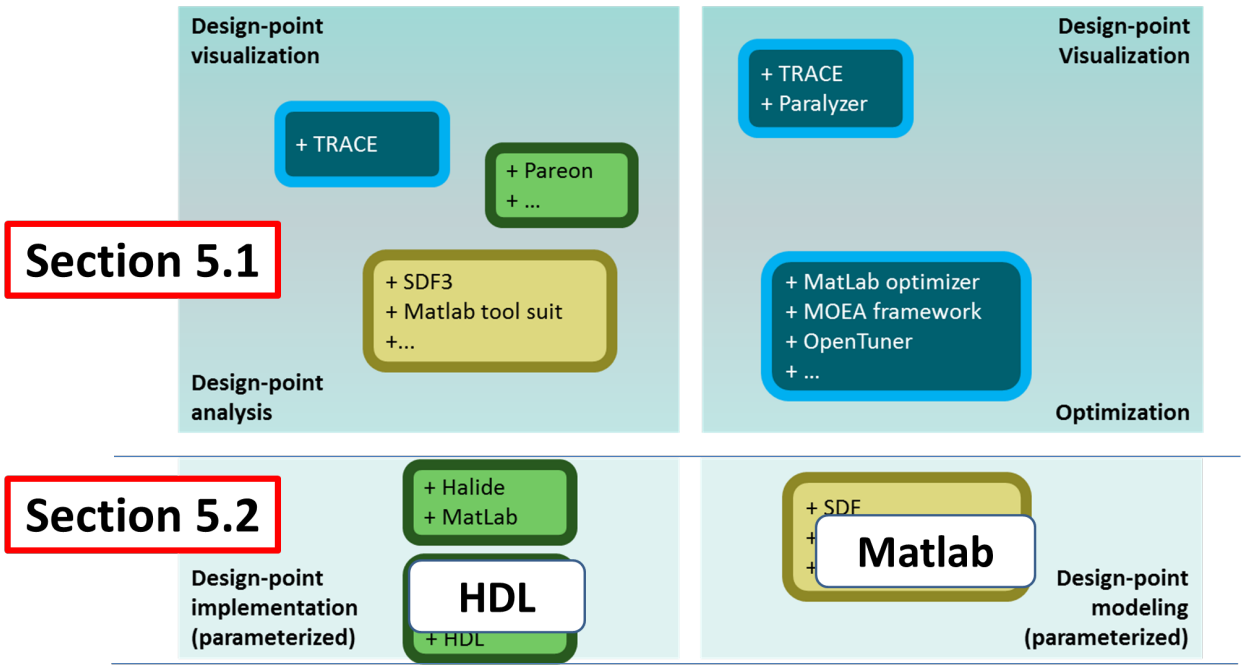


Figure 15: Section 5 overview

5.1 Pareon for design-point evaluation and trace visualization support

Vector Fabrics is developing the Pareon tooling for evaluating application software in the embedded system. This tooling is developed to address specifically compute-intensive applications (like image and video processing) on modern multi-core embedded platforms. This tool-supported application evaluation helps to analyse and review the application run-time behaviour regarding aspects such as:

- Detect performance issues
- Obtain hints on performance improvements especially related to multi-core behavior
- Obtain feedback on software defects that –among others- would lead to non-deterministic or undefined behavior.

To allow run-time analysis of application on embedded devices, extensive instrumentation tooling has been developed in Pareon. This instrumentation allows to extract execution traces on program run-time behaviour from the embedded platform, to be analysed on a host development system. This is depicted in the figure below:

From the application software development point-of-view, the Pareon report feedback focusses on multi-core usage. This is implemented through semantical analysis of the trace with respect to application multi-threading through the Posix and/or C++11 libraries as are being used on todays embedded systems. Such semantical analysis leads to messages on data-races, inconsistent locking, use of objects beyond their lifetime, etc.

For improved analysis of the ALMARVI applications, specific support is also being added to analyse for correct use of OpenCL in terms of concurrent processing and inter-core data sharing and synchronization. These specific OpenCL developments are beyond the scope of this deliverable, and instead reported in D4.1 which focusses more on OpenCL system aspects.

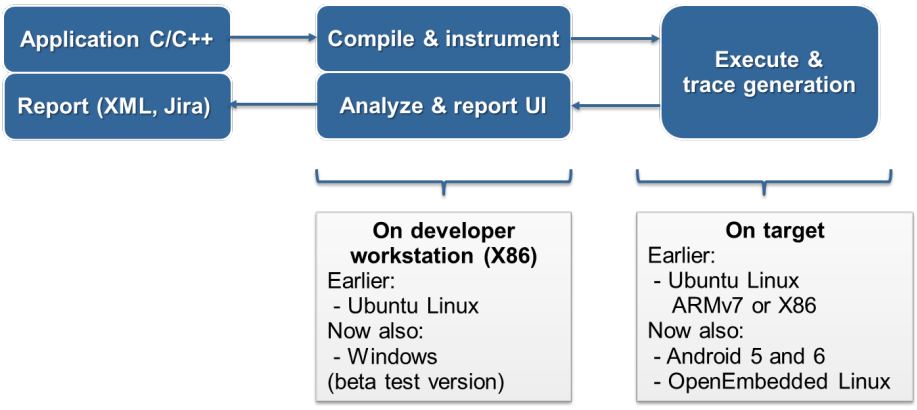
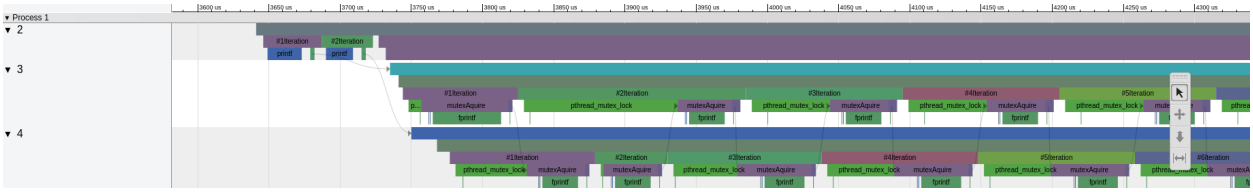


Figure 16: Pareon tool overview

Trace visualization support through Pareon

To further support application developers with their DSE process, the textual reporting is not very satisfactory. A closer cooperation between Vector Fabrics and the TUE shall lead to a visualization of the application trace analysis, which allows a more convenient feedback to the application designer regarding performance aspects. In particular, it depicts run-time aspects like contention on software locks and extensive stall-time of some threads in a concurrent

computing set-up. An initial screenshot taken from these current developments is shown below:



This picture has a horizontal time axis, and represents a zoomed-in fragment of a larger application run-time trace. It shows how two new threads were spawned from a main process. Both for the main and the two child threads, the function call-stack is displayed over the time axis. It shows inter-thread dependencies (through curved arrows, probably depicted too small in above print) that serialize thread behaviour by forcing run-time synchronization and sequentialization. Such events occur around locked mutexes, semaphores, barriers, and thread spawn and join operations. In combination, this provides insight in potential disappointing application performance.

This displaying of the application runtime behaviour is just an initial step. Further research and development will address the relations with application scheduling, and depicting results of deeper application behavioural analysis. One of the bottlenecks to address in this analysis and display tooling is sufficiently fast analysis and display of huge amounts of raw trace data, because even the compressed traces easily reach into the terabyte size range.

5.2 Floating-point to fixed-point Design Report C++ to FPGA conversion

Within the IXR department at Philips Medical, application analysis and profiling plays an important role in optimizing the applications to meet operational requirements. This section details the design of the Fixed-point Analyzer and Scaler Tool (FAST), which is used to analyze the range and precision error of floating-point to fixed-point conversions, as well as scale the bit width and decimal point of the fixed-point values.

5.2.1 Goal

At the Research and Development department of IXR at Philips, image processing chains need to be implemented in X-ray machines to provide clear pictures to the examining physician. One such image filter in the chain was recently converted from a Matlab model to C++. This C++ code uses floating-point values.

Floating-points can be a major hurdle to FAST performance due to complicated arithmetic. As a continuous throughput at high speeds is required, the final FPGA implementation needs to use fixed-point values instead of these floating-point values. The other implementation will be on the rVex, a dynamically reconfigurable VLIW processor. In this case, only bit-widths of fixed sizes are available.

This FPGA implementation will be programmed using two different technologies. One is the Vivado HLS, which converts C/C++ code to VHDL. With this, fixed-point values with variable, but limited, bit-widths are available.

Because custom bit-widths can vary between hardware blocks in the same FPGA design, every floating-point variable can be converted to a custom fixed-point design. The optimal solution would be to provide each variable with enough bits to adhere to a user-defined error precision, but save as much bits as possible to allow fast data transfer.

The goal of the FAST then is to provide insight into the C++ code floating-point values, comparing them to fixed-point values to maintain correct code. From this information, characteristics of the fixed-point values can be determined. Besides this analysis, the fixed-point bit-width should be able to be dynamically scaled by the user and fed back to the original C++ code, allowing for new analysis of the fixed-point values.

5.2.2 FAST requirements

The FAST code should adhere to certain hard, and some other softer requirements. Hard requirements in this case refer to requirements which **must** be fulfilled; soft requirements should be aimed for **as much as possible**. These requirements are listed beneath in their separate categories.

Hard requirements

- The range of the floating-point variables need to be determined for intermediate breakpoints in the code, preferably after every reassignment. If this is not possible due to high complexity, breakpoints should be defined at appropriate intervals. Breakpoints are positions in the code in which you poll data from defined variables.
- Generated code by FAST needs to be implementable on a FPGA device.
- The UI should be developed separately from the back-end, relying only on text or binary output files produced by running different implementations. This will allow for portability across different software platforms, from Matlab to C++ for example.
- Floating-point and fixed-point values should be compared at every defined breakpoint and at the output. At this comparison the absolute error and the relative error should be determined, as well as the range of the new value. Only measuring output errors does not provide enough insight to determine any useful characteristics of the variables.
- The user should be able to input a precision error, after which at every breakpoint it should be determined if the error between the floating-point and the fixed-point value is within this precision.
- A new fixed-point bit-width and the placement of the decimal point should be able to be adjusted dynamically in the UI. This new configuration should be written back into a header file, creating a new C++ implementation.
- It should be possible to combine and compare outputs of different implementations.
- A form of functional testing should be applied to code.

Soft Requirements

- Where possible, documentation should be provided in some structural form, like using Doxygen to generate documentation.
- A clean coding style is preferred, allowing for easy readability.
- The GUI should be portable across different platforms.
- The GUI should be easy to understand. Simplicity is key to this design.
- The FAST should be able to read both .txt files and binary files.
- The analysis of the different input files should not take too much time; the performance needs to be optimized wherever possible.

5.2.3 FAST Design

The FAST is segmented into two different parts. The first part, the back-end, consists of reading the input files and comparing the fixed-point and floating-point values. The second part, the front-end, is the visualization of the results presented in a GUI. This section is divided into three sub sections: 1) detailing the back-end design, 2) detailing the front-end design, and 3) describing the overall system design and how the back-end and front-end are combined.

Back-end

The design of the back-end of the FAST can be further divided into two parts: input construction and error comparison.

Input Construction

The input files that need to be compared are supplied by different implementations. The original Matlab model outputs the reference values, our so-called **Golden Standard**.

The first C++floating-point implementation should contain exactly the same values as the **Golden Standard**. However, because of the abstract implementation in Matlab, the program flow in the converted C++ code will differ from the original Matlab code. As a result, not all of the intermediate values in the Matlab code will be recreated in the converted C++ code, and not all the values can be compared directly. This is a first indication that certain breakpoints in-between functions need to be defined, of which can be made certain that every implementation will have these same values.

The C++ implementation that implements fixed-point values outputs files which contain approximately the same values as the **Golden Standard**, due to rounding errors. This is the first implementation where the back-end is needed to compare these files to the files outputted by the floating-point C++ conversion. Different implementations, with different bit widths in the fixed-point configurations and decimal points, may be created to compare. But by losing bits, accuracy is lost, so a balance needs to be found.

The Vivado HLS implementation also outputs files, much in the same way as the C++ code.

All these output files need to share certain characteristics.

- They should have the same structure, making it able for the back-end to read these values in automatically.

- The files should be in binary format. The conversion from .txt to binary takes some time, but reading binary files has more than a tenfold performance increase, increasing the performance dramatically.
- Variable data should be polled at the same stage of execution. For example, a noise reduction filter may have three phases: filter application, fine-tuning, and output. Every implementation should then output the values of all floating-point/fixed-point between these phases, and at the output.

After these output files are created and adhere to these characteristics, they can be compared.

Error Comparison

For every breakpoint, there exist different output files for every implementation. Looking purely at these values, the error between our Golden Standard and the implementation output is considered as the **Absolute Error**. This can be measured by subtracting these values.

A **Relative Error** is found when dividing the absolute error over the expected value. This will give an approximation of how serious the error is.

If one breakpoint has errors for an implementation, and the next breakpoint contains errors as well, these error values will stack and may obscure any hidden behavior. For this, the absolute error is not sufficient. It may be more useful to also check how much the error differs in a breakpoint from its previous breakpoint; this is considered the **Per-phase Error**.

The range of a variable can differ per testing image because different images can have different pixel values. If the range for a fixed-point variable is not sufficient, overflow can occur and can have disastrous consequences in the output. To this reason, every variable needs to be monitored and the **Range** needs to be calculated over the entire variable's runtime: this consists of the minimum and maximum value a variable grows and slinks.

These three properties of variables need to be calculated in every breakpoint using the files generated by the different implementations. These will be gathered by the back-end and will be made available for the front-end to display in a user-friendly way.

Back-end Diagram

To illustrate the different components of the FAST back-end the following diagram is presented. In this diagram, an example image filter is implemented, which results in different intermediate breakpoints as described earlier.

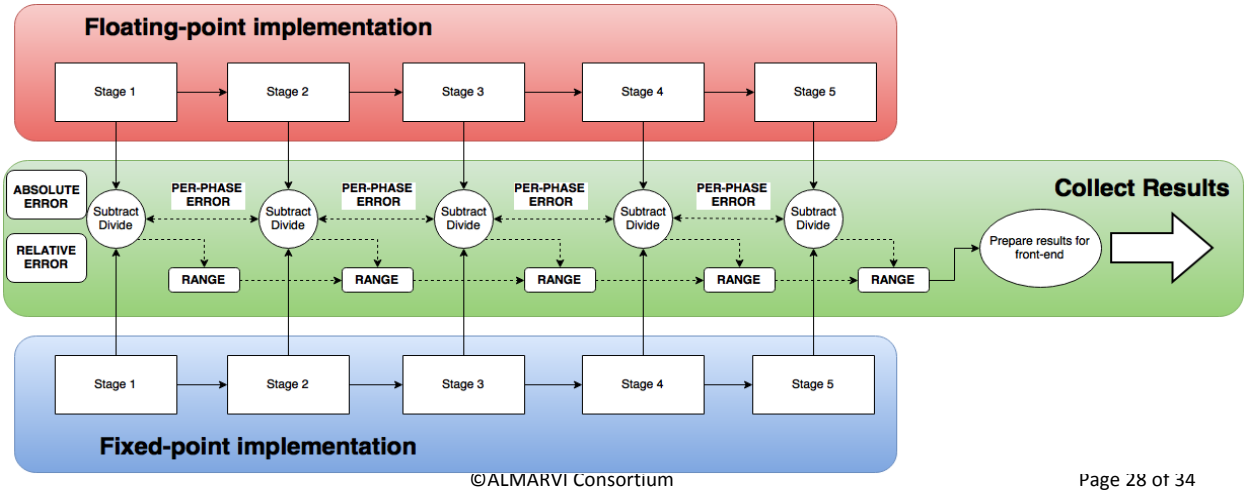


Figure 17: FAST back-end diagram

Front-end

The front-end implementation consists of three different components; the file comparison, the dynamic feedback and the actual GUI.

The back-end is developed in C++. Because of the requirement that the front-end and the back-end should be developed separately, the front-end needs not to be programmed in the same language. After all, the front-end only needs to read and display .txt or binary files, and output other text files. These functions can be performed in just about any programming language.

For the front-end development Java was chosen as an implementation language, due to different reasons.

1. Java is multi-platform, being installed on virtually every device supporting it. This makes distributing the FAST very easy as it will be stand-alone application.
2. Java has many dependencies which are suitable for fast prototyping of the GUI, for example JavaFX. As there is limited development time, fast prototyping is an important constraint.
3. Necessary experience with Java already exists. Because of the short development time, there is not much room to get acquainted with a new programming environment.

Another programming environment taken into consideration was Python. But after considering the different GUI libraries, the conclusion was that in the end almost none of the produced GUI's would be stand-alone, not only needing to install Python but several other libraries as well. This was considered sub-optimal and Java was to be preferred.

File Comparison

Before deciding on a particular configuration for fixed-point values, many different aspects have to be considered. For this reason, only looking at the results of a single implementation won't be sufficient. Several implementations, each with their own set of results collected by the back-end, need to be considered side-by-side to find the perfect solution.

Because of this, the front-end needs to be able to collect all these different results. New implementation results should be able to be added easily. The GUI should support these actions too.

In Java, reading and displaying files is readily supported and can be implemented with no trouble.

Dynamic Feedback

In case a fixed-point configuration does not fulfill the needs of the user, the bit width and the placement of the decimal point needs to be adjusted. This can be done using a slider in the GUI. After this is adjusted, the new header file, used in the underlying C++ implementation, should be generated by the front-end, as well as copying all the previous implementation files to create a whole new implementation.

This feedback is instantaneous and easy to use, creating new insight into the behavior of the code with a click of a button. Two disadvantages exist however. For one, the C++ implementation should be configured in such a way that by changing only one header file, the fixed-point variable configuration can be changed for the whole implementation, which may not be portable to other programming models. A second disadvantage is that the front-end won't be able to run the new implementation and collect the results. Support for running the new implementation is too time-consuming to implement in the GUI and rather easily done in one's own programming environment, which is why it isn't included.

Generation of these new header files is able in Java as well.

GUI interaction

The GUI will be written using JavaFX, using Netbeans as an IDE. This GUI will be quite simplistic, supporting only the few necessary functions for displaying files and values, as well as implementing the feedback in an intuitive way. Simplicity is key for fast prototyping.

A diagram of the front-end design is shown in Figure 18. Note that the back-end will provide the collected data.

It can be clearly seen that in this diagram, the GUI consists of two separate functions. A logical conclusion is to divide the GUI into two different displays between which the user can switch at the press of a button. Figure 19 shows two mock-ups of the first GUI prototype for analyzing and generating code, respectively.

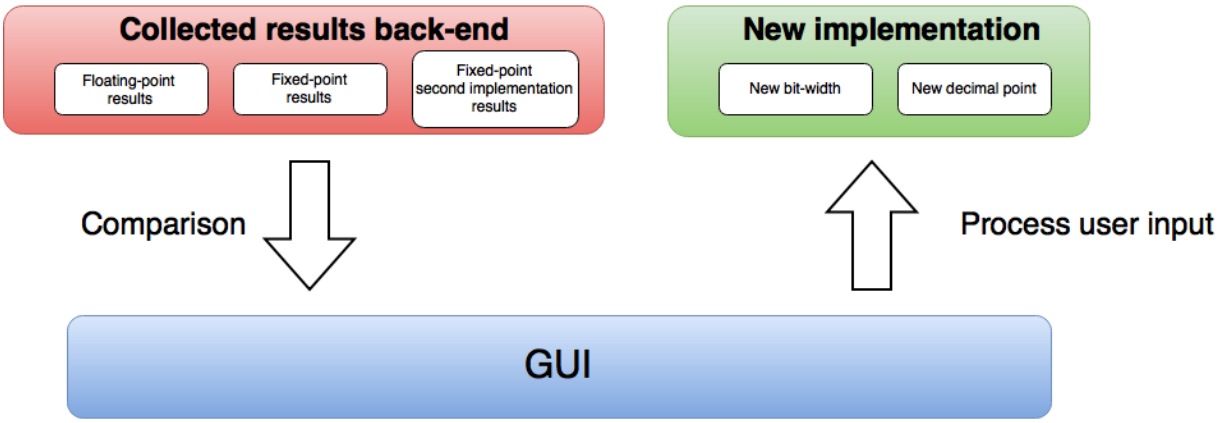
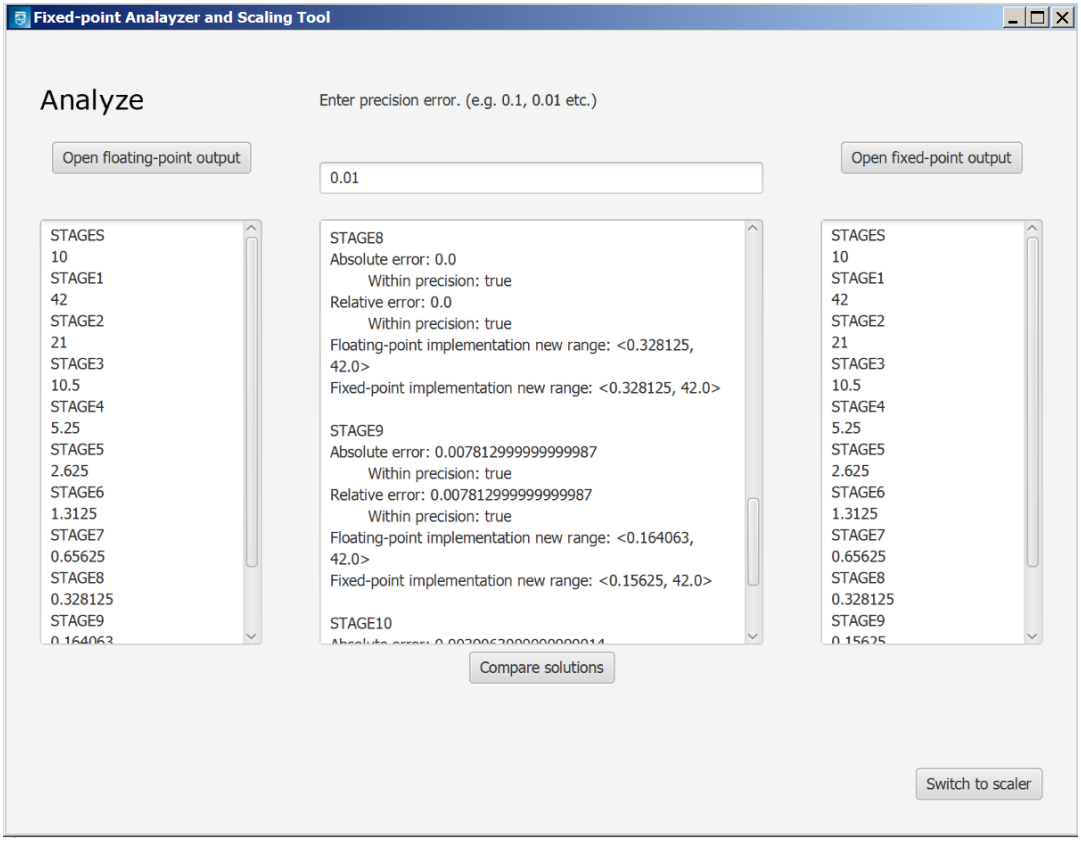


Figure 18: FAST front-end diagram



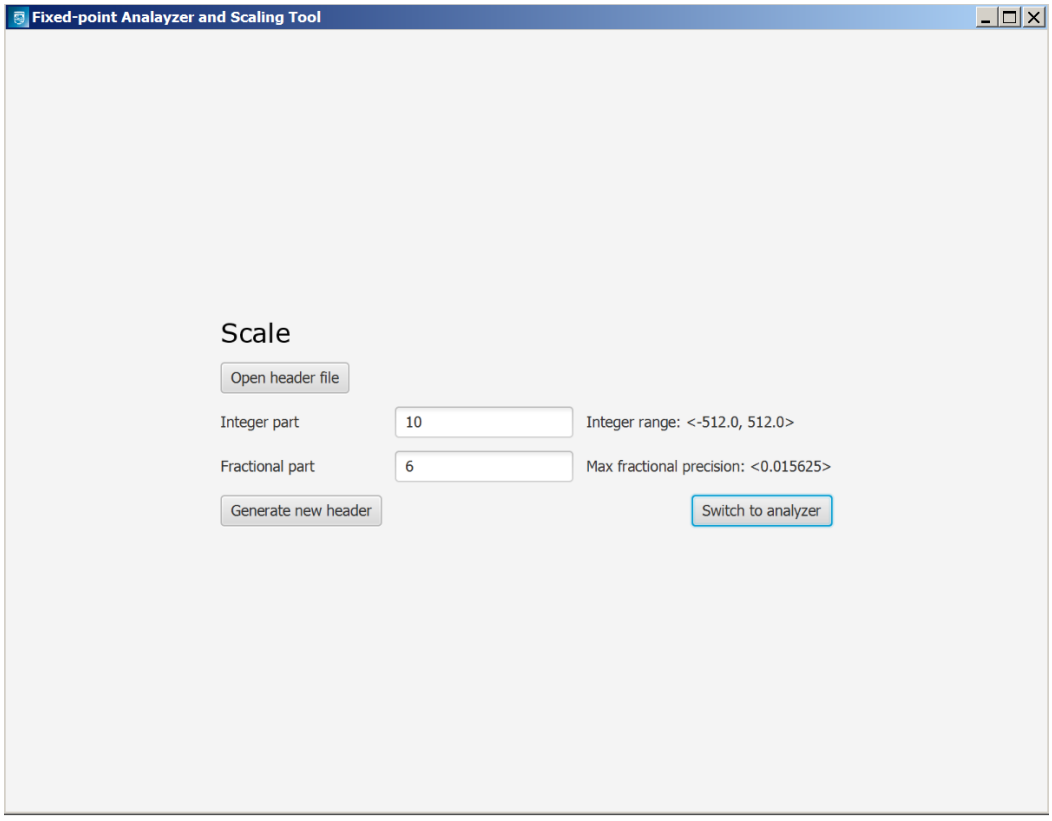


Figure 19: Two mock-ups of the first GUI prototype for analyzing and generating code

5.3 Conclusions

This chapter reported the source code level analysis and implementation for single design point for a streaming (image processing) application targeting specific platforms. Implementation level numbers from such analysis may be used at the model-level methods (Chapter 4) to further refine the models. The interaction and iteration over the model- and source code-level analysis and exploration of models as shown in V-model (Figure 1) allow for obtaining a closer-to-reality model, optimizing an implementation with respect to specific objective, target platform and perform trade-off analysis.

6 Conclusions

Overall, D4.3 presented modelling, analysis, evaluation, and implementation of single design point targeting multi-processors both at the model level and at the source code level. There are a number of tools extended and used in this context by ALMARVI partners. Various results are presented showing the improvement in terms of resource utilizing the models at different levels – component, application and multi-applications. A number of works are planned/ongoing which will be a part of the follow up deliverable D4.2 (Tool support for static application partitioning and mapping) due in D30.

©ALMARVI Consortium

7 References

[1] G. C. Buttazzo, Hard real-time computing systems: predictable scheduling algorithms and applications, Springer Science, 2011.

[2] B. Akesson, A. Minaeva, P. Sucha, A. Nelson and Z. Hanzalek, "An efficient configuration methodology for time-division multiplexed single resources," in *Proc. of Real-Time and Embedded Technology and Applications Symposium*, 2015.

[3] A. Behrouzian, D. Goswami,T. Basten, M. Geilen and H. Alizadeh, "Multi-Constraint Multi-Processor Resource Allocation," in *Proc. of Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), International Conference on*, 2015.

[4] M. Hamdaoui, and P. Ramanathan, "A dynamic priority assignment technique for streams with (m, k)-firm deadlines," *Computers, IEEE Transactions on,* vol. 44, no. 12, pp. 1443-1451, 1995.

[5] S. Stuijk, Basten, T. a. Geilen, MCW and H. Corporaal, "Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs," in *Proc. of the 44th annual Design Automation Conference*, 2007.

[6] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," in *Proc. of the IEEE*, 1987.

[7] M. Geilen and S. Stuijk, "Worst-case performance analysis of synchronous dataflow scenarios," in *Proc. of the 8th ACM international conference on Hardware/software codesign and system synthesis*, 2010.

[8] F. Siyoum, M. Geilen and H. Corporaal, "Symbolic Analysis of Dataflow Applications Mapped onto Shared Heterogeneous Resources," in *Proc. of the 51st annual design automation conference*, 2014.

[9] S. Stuijk, M. Geilen and T. Basten, "SDF For Free," in *Proc. of 6th International Conference on Application of Concurrency to System Design*, 2006.

[10] "Trace website," [Online]. Available: http://trace.esi.nl/.

[11] R. Alur and T. Henzinger, "Real-time logics: complexity and expressiveness," *Information and Computation,* vol. 104, pp. 390-401, 1993.

[12] M. Hendriks, T. Basten, J. Verriet, M. Brassé, L. Somers, "A Blueprint for System-Level Performance Modeling of Software-Intensive Embedded Systems," *International Journal on Software Tools for Technology Transfer (STTT), Vol. 18, No. 1, p. 21-40, 2016.*

[13] O. Kupferman and M. Vardi, "Model checking of safety properties," *Formal Methods in System Design,* vol. 19, no. 3, 2001.

[14] H. Ho, J. Ouaknine, and J. Worrell, "Online monitoring of metric temporal logic," *in Runtime Verification, Lecture Notes in Computer Science. Springer,* vol. 8734, 2014.

[15] M. H. e. al., "Checking Metric Temporal Logic with TRACE," Accepted for publication in ACSD 2016.

[16] M. Hendriks, J. Verriet, T. Basten, M. Brasse, R. Dankers, R. Laan, A. Lint, H. Moneva, L. Somers, M. Willekens. "Performance Engineering for Industrial Embedded Data-Processing Systems," *Workshop on Processes, Methods and Tools for Engineering Embedded Systems*, PROMOTE 2015.