ASAM : Automatic Architecture Synthesis and Application Mapping; dl. 2: Final design methodology, flow, and tool requirements

Published: 01/01/2011

Document Version
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the author's version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

Citation for published version (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.
• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 27. Dec. 2018
Grant agreement no. 100265
Artemis Project

ASAM
Automatic Architecture Synthesis and Application Mapping

D1.2: Final Design Methodology, Flow, and Tool Requirements

Due Date of Deliverable: 30th April, 2011
Completion Date of Deliverable: 30th April, 2011
Start Date of Project: 1st May, 2010 – Duration 36 Months

Lead partner for Deliverable: SH
Author(s): M. Lindwer (SH), M. Cocco (SH), R. Corvino (TUE), R. Jordans (TUE), L. Jozwiak (TUE), J. Madsen (DTU), P. Meloni (UNICA), L. Raffo (UNICA), G. Notarangelo (STM), B. Kienhuis (CMPN)

Revision: v1.0

Project co-funded by the Artemis Joint Undertaking Call 2009

<table>
<thead>
<tr>
<th>Dissemination Level</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PU</td>
<td>Public</td>
</tr>
<tr>
<td>PU/COA</td>
<td>Public with confidential appendices</td>
</tr>
<tr>
<td>PP</td>
<td>Restricted to other program participants (including Commission Services)</td>
</tr>
<tr>
<td>RE</td>
<td>Restricted to a group specified by the consortium (including Commission Services)</td>
</tr>
<tr>
<td>CO</td>
<td>Confidential, only for members of the consortium (including Commission Services)</td>
</tr>
</tbody>
</table>

Revision: v1.0
<table>
<thead>
<tr>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1</strong> Introduction</td>
</tr>
<tr>
<td><strong>2</strong> Scope</td>
</tr>
<tr>
<td><strong>3</strong> Users, tasks, and use cases</td>
</tr>
<tr>
<td>3.1 The design team</td>
</tr>
<tr>
<td>3.2 Tasks on which the ASAM flow will support the design team</td>
</tr>
<tr>
<td>3.3 Default use case for novice users</td>
</tr>
<tr>
<td>3.3.1 Default use case for expert users</td>
</tr>
<tr>
<td>3.3.2 Optimization use case for expert users</td>
</tr>
<tr>
<td><strong>4</strong> Specification of generic software quality</td>
</tr>
<tr>
<td>4.1 Quality Model in terms of Characteristics</td>
</tr>
<tr>
<td>4.1.1 Internal and External Quality Characteristics</td>
</tr>
<tr>
<td>4.1.2 Quality in use Characteristics</td>
</tr>
<tr>
<td>4.2 ISO 9126 Metrics</td>
</tr>
<tr>
<td><strong>5</strong> The ASAM project</td>
</tr>
<tr>
<td>5.1 The current Silicon Hive design flow and ASAM aims</td>
</tr>
<tr>
<td>5.2 General method</td>
</tr>
<tr>
<td>5.3 Design space exploration for System-architecture synthesis</td>
</tr>
<tr>
<td>5.4 Design space exploration for ASIP-architecture synthesis</td>
</tr>
<tr>
<td>5.5 Prototyping</td>
</tr>
<tr>
<td><strong>6</strong> Multi-processor subsystem IP, design flow, and code compilation</td>
</tr>
<tr>
<td><strong>7</strong> Compaan Design</td>
</tr>
<tr>
<td>7.1 Compaan Technology</td>
</tr>
<tr>
<td>7.2 Compaan Product</td>
</tr>
<tr>
<td>7.3 Example</td>
</tr>
<tr>
<td>7.3.1 The Code Example</td>
</tr>
<tr>
<td>7.3.2 How the code works</td>
</tr>
<tr>
<td><strong>8</strong> Design Space Exploration</td>
</tr>
<tr>
<td>8.1 Overview of the Framework</td>
</tr>
<tr>
<td>8.1.1 An example of framework modeling: the Service Model</td>
</tr>
<tr>
<td>8.1.2 Application Modeling</td>
</tr>
<tr>
<td>8.1.3 Architecture Modeling</td>
</tr>
<tr>
<td>8.1.4 System Modeling</td>
</tr>
<tr>
<td><strong>9</strong> ASIP Customization</td>
</tr>
<tr>
<td><strong>10</strong> The prototyping/implementation infrastructure</td>
</tr>
<tr>
<td>10.1 Evaluation results</td>
</tr>
<tr>
<td><strong>11</strong> References</td>
</tr>
<tr>
<td><strong>12</strong> Glossary and Terminology</td>
</tr>
<tr>
<td>Appendix A - SiliconHive technology overview (confidential)</td>
</tr>
</tbody>
</table>
1 Introduction

The ASAM (Automatic Architecture Synthesis and Application Mapping) project aims to develop a system-level methodology and design flow that will facilitate the design of complex embedded hardware-software systems. Such systems consist of optimized heterogeneous multi-processor hardware systems, running efficiently mapped and highly parallel software systems.

The top-level design methodology, as specified in this document, is leading the further development of particular methods and tools, in terms of defining the overall design flow and tool collaboration. From the specified methodology and flow follow the input/output relations between the tools, which in turn determine the top-level requirements for the particular tools. This document summarizes the overall ASAM design methodology, flow, and tool requirements.

This document determines, among others, the input and output formalisms of the design stages, used in the project. For these, the project aims to re-use as much as possible existing industry-strength technologies. In addition, this document also specifies the top-level constraints and expected quality of output results.

Within the scope of this document we define with the term ASIP processor a VLIW processor tailored to a particular application. We define a system as composed by 1 or more ASIP processors, 1 or more host processors and 0 or more devices for storage and data communication. By definition a system may contain other systems. Within this document the term “macro” will be used interchangeable with the term “system” and the term “micro” will be used interchangeable with the term “ASIP.”
2 Scope

This document is intended as a high-level requirement definition for the design methodology and design flow for complex embedded heterogeneous ASIP-based multi-processor systems. It provides an overview of the interoperability of existing and newly to be developed tools for automatic application-driven synthesis of hardware systems and application mappings. As such, it drives the definition of the ASAM design flow, particular tools, and the design of the interfaces between tools within this flow. Since a flow must have a defined starting point and a defined result, these are also specified in this document.

The document is not a requirement specification of any particular tool. The tools that will be developed within the ASAM project will all have their own specifications.

This document also is not a manual for the ASAM flow.

This document is meant to be a reference for the designers of the final design methodology and individual tools within the ASAM project, providing them with a specification of the expected interoperability, and I/O behavior of the tools to be developed.
3 Users, tasks, and use cases

3.1 The design team
ASAM focuses on the digital sub-system design of application-specific development trajectories. The intended users/stake holders of the ASAM flow are design teams consisting of:

- Application system designers: are to provide inputs/specifications. The project will not address their work per-se.
- Algorithm designers: are involved in order to modify algorithms such that they map efficiently on typical ASAM systems.
- Software and DSP engineers, who provide the implementation of the algorithm and drive optimizations of the algorithm and its encoding towards the multi-processor system.
- SoC system designers, who provide SoC (sub)system descriptions and drive the flow in terms of optimizations for those system descriptions. An SoC (sub)system generally consists of (multiple) processors and system-level devices, such as DMAs, memories, on-chip buses, etc.
- Processor designers, who drive the flow in terms of supplying descriptions of ASIPs and driving the flow's optimizations to those ASIPs.
- VLSI engineers: provide feedback to further optimize sub-system and processors. The project will not address their work per-se.

In particular the SoC/processor engineers have to take into account that their design will survive the current application, i.e. be generic enough to also be applicable to similar applications in the same domain.

3.2 Tasks on which the ASAM flow will support the design team
The ASAM flow guides and supports the team in the following tasks:

- application analysis
- restructuring and partitioning the algorithm’s sequential code for parallel systems,
- identifying optimization strategies for code partitions. Broad optimization strategies are: vectorization, task-level parallelism and super pipelining, instruction-level parallelism (VLIW), custom operations.
- vectorization for those partitions for which this is applicable,
- proposing the number, kinds and parameters of processors and other devices
Public with confidential appendices

for the SoC sub-system

- distribution of the application’s computation processes among different processors of the multi-processor system
- distribution of datasets to support partitions running on different processors and/or different VLIW data paths,
- mapping specific parts of algorithms on custom operations
- modifying application code to take advantage of loop optimization
- instantiation of processors in the SoC description according to the identified partitions
- optimization of the processors in the SoC according to the identified optimization strategies for each of the partitions
- evaluation of the resource requirements (e.g. area) for target silicon and FPGA technologies,
- evaluation of performance and power consumption of the application running on the resulting multi-processor system.

3.3 Default use case for novice users

This use case is meant to assist novice users in generating an initial multi-processor system and obtaining prototype quality level results. The ASAM flow is expected to perform the required tasks almost fully automatically.

The design team provides a standard single-processor SoC system and an initial application implementation which consists purely of sequential ANSI-C code. In this case, the ASAM methodology will particularly focus on the tasks of partitioning the code and identifying optimization strategies for the code. The ASAM flow will result in:

- a proposal for a parallel implementation of the application,
- a new multi-processor SoC design,
- evolving processors within the SoC design to fit the parallel application partitions
- a mapping of the application on the multi-processor SoC design,
- estimated performance, area, and power numbers for the application running on the multi-processor SoC design,
- PPA numbers of the solution are expected to be sufficient as proofs of concept or on par with 2010 state-of-the-art.

3.3.1 Default use case for expert users

This use case is meant to supply expert users with a methodology to achieve
Public with confidential appendices

optimized solutions in a very short time frame. The entry points are the same as with the previous case. The difference with the previous use case is that the methodology allows expert users drive the flow to achieve certain results, of which the user knows that they are achievable.

The ASAM flow provides expert users with the option to supply starting point processors that are already tuned to the required or a similar application domain. The tools will automatically select these more optimized processors to fit the properties of the application code. However, the users may also assign code sequences to certain processors. Users may provide steering in terms of restricting or widening the scope of automatic generation of custom operations (to the point where the expert users may explicitly select certain customer operations).

The starting point SoC system may contain hardwired accelerators, which are specified such that the flow will be able to match their functionality with parts of the application code.

In this way, the flow provides for a semi-automatic methodology that will allow expert users to very quickly converge to a solution that is on par with 2013 state-of-the-art. The ASAM flow will have accelerated the process of achieving this convergence.

3.3.2 Optimization use case for expert users

This use case is meant to provide expert users with a methodology to improve on the 2013 state-of-the-art. The input consists of already optimized code and heterogeneous multi-processor subsystems. The design space exploration tools will be used to iterate through small modifications to the subsystem, which will gradually improve the PPA numbers for the proposed solution beyond the 2013 state-of-the-art.
4 Specification of generic software quality

An adequate evaluation of an automatic design process and related tools, as ASAM project aims to propose, is important for both the project’s evaluators and its stakeholders. It will be necessary to evaluate the design methodology in the field of system-level heterogeneous SoC design, as well as, the resulting tools.

The main difficulty of evaluating quality in this Engineering field is to identify the fundamental process characteristics and map them into the final framework for a shareable understanding. Since the classification of framework entities has been pointed out (resources implemented, process, and whatever an empirical consideration of the tools could suggest), some quality factors and/or potential quality model has to be associated to each entity and the relative metric parameters have to be decided and evaluated: the result is a balanced quality assurance procedure of the entire project results. In this respect, it has to be taken into account that the goal of this project is not to produce an industrial-strength system of tools, but rather to study certain design tasks and problems, propose, and research appropriate solution strategies and methodologies. In the following paragraphs, when proposing the quality characteristics and metrics these constraints are taken into account.

In order to maximize the results of the design tool evaluation a common property framework shall be identified. The framework, chosen and proposed in the ASAM Technical Annex [1], is based on the international standard BS ISO/IEC 9126-1:2001 [2] (and associated parts 2 [3], 3 [4] and 4 [5]). This will be used as a basis for identifying the key attributes that are to be used to evaluate and compare tools to support the hardware and software development. A quality attribute is a property of a process or product that can have some qualitative or quantitative value and can be measured or observed. The ISO 9126 family addresses the characteristics of a software product that may be measured and used for evaluation. It should be noted that different quality attributes and metrics are differently important for deliverables of different kinds. Also several metrics may not directly apply to early software prototypes. Nevertheless, ISO 9126 is a good reference based on which the framework for ASAM quality evaluation can be proposed.

The Figure 1 is derived from the ISO/IEC 9126-1:2001 quality assessment context: it summarizes the flow throughout entity types, their own quality factors and measured attributes. The main advantage is that one or more quality models can model each quality factor and the output will be analysed and inferred by numbers.
Resources and Process entities are standardized into ISO/IEC 14598 series, while Product and System in Use into ISO/IEC 9126 ones. As outlined before, ASAM evaluation strategy will be developed based on standard ISO/IEC 9126 rules.

The ISO/IEC 9126 family of standards defines a Quality Model Framework with four different parts:

- Quality model: classifies software quality in a structured set of characteristics and sub-characteristics as described in Section 4.1
- External metrics: apply to the running software
- Internal metrics: static measures which do not rely on the software execution
- Quality in use metrics: available when the final product is used in real condition

Figure 2, derived from ISO/IEC 9126-1:2001, shows the relationships among these four factors and their own relative quality requirements.

---

**Figure 1 – Quality Assessment Context.**

**Figure 2 – ISO/IEC 9126 Quality Model Framework.**
As described in detail:

- **External quality**: the quality of the product as seen during testing or evaluation. External attributes can be measured only after the product is created and have to be externally visible (i.e., usability and/or reliability that assures the software works just the user expects). “External attributes of products can only be measured with respect to how the product relates to its environment” [3]. These quality metrics will be mainly specified in terms of the performance aspects of the resulting systems, as specified in the use cases of Deliverable D6.1, when applying the usage scenarios of Section 3.

- **Internal quality**: the quality of the product coming from the software’s internal structure (attributes like standard code compliance, size or coupling, complexity and modularity) and by examining artifacts (documentation) associated with the product. In other words, an internal attribute can be measured in terms of the product, process or resource on its own. Since the artifacts generated by the project mainly consist of reports, prototype tools, and demonstrators, the internal quality requirements are specified in terms of guidelines, rather than strict procedures.

- **Quality in use**: the quality of the product as seen during execution of the product by an end user who is using the product in its normal, intended mode of operation. The “Quality in use” intent is measuring the product’s ability to give effectiveness and efficiency to the user’s work, compared to a safe use and satisfaction in doing use the product. These quality metrics will be mainly specified in terms of design effort and learning curve for different groups of users, as specified in the usage scenarios of Section 3 and Deliverable D6.1.

### 4.1 Quality Model in terms of Characteristics

#### 4.1.1 Internal and External Quality Characteristics.

*Internal and External Quality* consists of 6 characteristics as reported in Figure 3:

![Internal and External Quality Characteristics](image)

*Figure 3 - Internal and External Quality Characteristics.*

Each characteristic is here briefly described in standard terms and related simpler meaning:

- **Functionality**: a set of attributes that bear on the existence of a set of functions and their specified properties. The deliverable is able to provide capabilities which meet stated and implied needs when it is used under specified conditions.
Public with confidential appendices

- **Reliability**: a set of attributes that bear on the capability of deliverables to maintain their level of performance under stated conditions for a stated period of time. The deliverables will be robust under external stated inputs.

- **Usability**: a set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users. The deliverables results are easy-to-use, easy to learn, and attractive (i.e.: GUI support).

- **Efficiency**: a set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions. The tools and methodology use the available resources appropriately, executing the right trade-off between performance and amount of resources available.

- **Maintainability**: a set of attributes that bear on the effort needed to make specified modifications. The capability of the deliverables to be corrected improved and adapted to possible environment changes, with a contained commitment.

- **Portability**: a set of attributes that bear on the ability of deliverables to be transferred from one environment to another.

Each *Internal and External Quality* characteristic provides a set of sub-characteristics listed below:

- **Functionality sub-characteristics**:
  - **Suitability**: the capability of the deliverables to provide an appropriate set of functions for specified tasks and user objectives.
  - **Accuracy**: the capability of the deliverables to provide the right or agreed results or effects with the needed degree of precision. Since the software deliverables are prototypes, they are expected to provide required accuracy on a set of pre-defined applications. However, for prototype tools, this is not expected in the context of completely new applications.
  - **Interoperability**: the capability of the deliverables to interact with one or more specified systems. In the context of the ASAM project, this is an important requirement, as the methodology needs to work in coordination with (EDA) tools from different partners and different vendors. However, for prototype deliverables, the set of interfaces to EDA tools will be restricted to just those that are needed to demonstrate the suitability of the methodology on the ASAM project’s sample applications.
  - **Security**: the capability of the deliverables to protect information and data so that unauthorised persons or systems cannot read or modify them and authorised persons or systems are not denied access to them. Data integrity and security is not a primary objective of the prototype tools. However, the security of sensitive competitive information, if available on the systems on which the tools operate can not be compromised.
  - **Functionality Compliance**: the capability of the deliverables to adhere to standards, conventions or regulations in laws and similar prescriptions relating to functionality.

- **Reliability sub-characteristics**: 

Maturity: the capability of the deliverables to avoid failure as a result of faults in the software. The prototype tools will operate without failure on ASAM’s sample applications. Being prototypes, this can not be generically guaranteed.

Fault Tolerance: the capability of the deliverables to maintain a specified level of performance in cases of software faults or of infringement of its specified interface.

Recoverability: the capability of the deliverables to re-establish a specified level of performance and recover the data directly affected in the case of a failure. Since the prototype tools are not expected to be put to operation on tasks that have a direct critical or other implication, this characteristic deemed not applicable for the ASAM project.

Reliability Compliance: the capability of the deliverables to adhere to standards, conventions or regulations relating to reliability. The deliverables will operate reliably on ASAM’s sample applications.

Usability sub-characteristics:

Understandability: the capability of the deliverables to enable the user to understand whether they are suitable, and how they can be used for particular tasks and conditions of use. This is an important aspect for ASAM, as ASAM’s results are specifically intended to increase design efficiency.

Learnability: the capability of the deliverables to enable the user to learn their application. As with the Understandability characteristic, this one is also important to increase the efficiency of the design phase. However, it is not the objective of the ASAM project to generate product-quality documentation with each of the tool deliverables.

Operability: the capability of the deliverables to enable the user to operate and control it.

Attractiveness: the capability of the deliverables to be attractive to the user. For the prototype tools, this is not a main concern.

Usability Compliance: the capability of the deliverables to adhere to standards, conventions, style guides or regulations relating to usability.

Efficiency sub-characteristics (for the prototype tools, this is not a main concern):

Time Behaviour: the capability of the prototype tools to provide appropriate response and processing times and throughput rates when performing their function, under stated conditions.

Resource Utilisation: the capability of the prototype tools to use appropriate amounts and types of resources when they perform their function under stated conditions.

Efficiency Compliance: the capability of the prototype tools to adhere to standards or conventions relating to efficiency.

Maintainability sub-characteristics, for deliverables such as prototype tools, demonstrators, reports, and documentation:

Analysability: the capability of the prototype tools to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified.
Public with confidential appendices

- **Changeability**: the capability of the prototype tools to enable a specified modification to be implemented. The deliverables will have clear specification- and design documents and will be built according to these documents. Reasonably skilled software engineers will be able to convert modifications to the specification- and design documents into modifications to the deliverables.

- **Stability**: the capability of the prototype tools to avoid unexpected effects from modifications.

- **Testability**: the capability of the prototype tools to enable modified software to be validated.

- **Maintainability Compliance**: the capability of the deliverables to adhere to standards or conventions relating to maintainability.

- **Portability sub-characteristics**:
  - **Adaptability**: the capability of the deliverables to be adapted for different specified environments without applying actions or means other than those provided for this purpose for the software considered. The ASAM project will choose an operating platform. The deliverables are expected to operate on that platform. Portability to other platforms is not an objective of this project.
  - **Installability**: the capability of the tool deliverables to be installed in a specified environment. All tools in the flow must be installable on a Red Hat Enterprise Linux 4 or equivalent operating system.
  - **Co-existence**: the capability of the tool deliverables to co-exist with other independent software in a common environment sharing common resources.
  - **Replaceability**: the capability of the tool deliverables to be replaced by or to replace tools with purposes in the same environment. As ASAM is a research project, this characteristic is not deemed practical for ASAM’s prototype tools. This project will result in tools for which no replacement exists, yet.
  - **Portability Compliance**: the capability of the deliverables to adhere to standards or conventions relating to portability.

Figure 4 summarizes all Internal and External Quality characteristics and sub-characteristics [2].
4.1.2 Quality in use Characteristics.
The Quality in use incorporates the rules of measuring of the user’s perspective on the deliverables. It standardizes how evaluate the ability of deliverables to support specified users goals in stated contexts of use.

Quality in use may be assessed using the following characteristics:

- **Effectiveness**: the capability of the deliverable to enable users to achieve specified goals with accuracy and completeness in a specified context of use.
- **Productivity**: the capability of the deliverable to enable users to expend appropriate amounts of resources in relation to the effectiveness achieved in a specified context of use.
- **Safety**: the capability of the deliverable to achieve acceptable levels of risk of harm to people, business, software, property or the environment in a specified context of use.
- **Satisfaction**: the capability of the deliverable to satisfy users in a specified context of use.

Figure 5 depicts the Quality in use characteristics.

![Quality in use characteristics](image)

4.2 ISO 9126 Metrics.
ISO 9126 provides 3 different quality metrics:

- External Metrics (ISO/IEC TR 9126-2:2003(E));
- Internal Metrics (ISO/IEC TR 9126-3:2003(E));
- Quality Metrics (ISO/IEC TR 9126-4:2004(E)).

The metrics allow a quantitative assessment of the software quality based on the specific characteristics of the product developed.

The assessment criteria, and its own relative metrics, depend on the quality requirement. To do this, the standard ISO 9126 proposes a scheme of the evaluation process that aims to evaluate the different components of the software product.
developed and to measure such quality characteristics.

The metrics for *external quality* evaluation are set up and applied during the validation process of the software product. They are, essentially, the quantitative elements those represent and validate the external requirements of a software product against the characteristics and sub-characteristics of the ISO 9126 framework. A metric for external quality has to be able to predict the level of *quality in use* of the software product, throughout the quantification of the level of compliance of the product compared to the explicit and implicit user requirements.

The metrics for *internal quality* evaluation are defined and applied during the verification phase of the software development. They measure and evaluate the quality of a product against the characteristics and sub-characteristics of the ISO 9126 framework, only in the interim work-in-progress processes and in the final but not executable stage. A metric for internal quality has to be able to predict the external level quality and prevent the potential faults in advance.

The metrics for *quality in use* evaluation are mainly able to verify the ability of a product to meet user requirements in a given usage scenario, in relation to specific goals. They are defined and applied during the operative phase by the final user and eventually re-defined depending of the user feedback reports. In general, they are a common mixture of elementary attributes related to the interaction between the product and the user.

It’s important to underline that these three quality metrics aren’t compartmentalized: there are several relationships and interdependencies among them and each of them should measure the same characteristic defined into the standard ISO 9126. For example the reliability can be externally measured by detecting the number of errors during the execution of the product in a given period of observation, and internally by inspecting the source code to verify the level of fault tolerance it has.

In conclusion the quality of the product has to be measured considering each characteristic from these all three dimension: external metrics, internal metrics and quality in use metrics.

Figure 6 summarizes a generic ASAM Evaluation Plan related to the ASAM Framework Implementation time-line.
Here it should be noted that, although the ISO 9126 standard implies that input from Final Users is taken into account in the verification of the results, the use scope of the project results is actually limited to the ASAM Partners.

It should be noted that Amdahl’s law plays a role: the ASAM flow/results necessarily only impact a part of the final system. Thus, beyond a certain level of optimization, when the influence of the digital part of the design is small, the ASAM optimizations on that digital part no longer influence the metrics on the full system. When determining attributes and metrics of the output of the ASAM flow, they are described in terms of those aspects of the target applications (as described in D6.1) which are influenced by the ASAM flow, i.e. the digital SoC sub-system (DSS).

The ASAM-specific quality attributes of the DSS are defined as follows:

- Design throughput time
- Design effort, in terms of man-months
- Power consumption (of DSS)
- Area (of DSS)
- Performance
5 The ASAM project

ASAM targets the heterogeneous multiprocessor systems, and specifically multiprocessor systems on chip (MPSoCs), based on adaptable application-specific instruction set processors (ASIPs) customizable to a particular application through instantiation and extension. Only a few companies in the world possess and license such a heterogeneous MPSoCs technology based on adaptable ASIPs. One of these companies is Silicon Hive being a partner of this project. Heterogeneous MPSoC involves several different ASIPs, each customized for a particular part of a complex application, interconnected with global memories and other sub-systems using a configurable hierarchical interconnection network, and implemented on one chip together with possible hardware accelerators and other digital or analog sub-systems.

The heterogeneous MPSoC technology based on adaptable ASIPs is very powerful. For example, several such powerful ASIPs with approximately 100 issue slots in total, each for 64-way vector processing, can be placed on a single chip implemented in 22 nm CMOS technology. When operated at 400-600MHz, these ASIPs can deliver more than 1 Tops/s, with power consumption far below the upper limit of mobile devices.

The customizable for specific applications heterogeneous massively-parallel MPSoC based on the configurable and extensible ASIPs enables efficient application-specific exploitation of various kinds of parallelism:

1. The MPSoC’s multiple processors serve coarse-grain parallelism exploitation at the task or sub-program level.
2. The ASIP’s parallel issue slots and custom instructions serve the fine-grained acceleration.

The MPSoC design technology based on adaptable ASIPs and hardware accelerators addresses fundamental development challenges of electronic systems for modern highly-demanding embedded and pervasive applications. It is able to deliver a high performance and low energy consumption close to hardwired ASICs, and high flexibility of programmable processors at the same time. MPSoCs based on this technology can be built at substantially lower costs and with much shorter times to market than hardwired ASICs or programmable MPSoCs based on custom processors built from scratch.

Moreover, this ASIP-based technology provides the flexibility required for engineering of robust, context-aware and adaptive systems for many application domains. It is relevant for a very broad range of application domains, e.g.: telecommunications, networking, multimedia, consumer electronics, medical instrumentation, advanced machinery, wireless sensor systems, military applications, automotive applications, etc., and for the cross-domain convergence products (e.g. mobile phones equipped with medical monitoring), and it is applicable to several implementation technologies, e.g.: SOC or ASIC, structured ASIC, and FPGA.

The main problem of efficient exploitation of this technology is that, with the current state of the art, the architecture, software, and hardware of an ASIP-based system
have to be designed by expert users having deep knowledge of application analysis and restructuring, the target technology, and of the mapping and compilation process. Moreover, even for an expert user application analysis and construction of related high-quality software and hardware system structures, and corresponding specifications is a very complex and error-prone task, which, because of its complexity, dramatically reduces the current possibility for high-quality systematic exploration of the system hardware and software design spaces, and results in low productivity or decreased design quality.

The automatic ASAM design methodology aims to much increase the productivity while not compromising on quality, in order to reduce the design cost and design time, while at the same time ensuring a high-quality of the designed systems.

5.1 The current Silicon Hive design flow and ASAM aims

The ASAM design flow builds on the design flow of Silicon Hive outlined in . The Silicon Hive design flow is based on hierarchies of IP blocks and has a plug-in mechanism to accept external IP. The ASAM IP block hierarchy is extensively described in subsequent sections. In this section we present an overview of the basic design steps.

![Silicon Hive design flow](image)

**Figure 7: Silicon Hive design flow**

The current design flow using the Silicon Hive IP and tools can be briefly described as follows:
1. The application code is manually split into a number of parallel sub-programs that may communicate.

2. The system consists of a set of processors (ASIPs) each specifically designed to efficiently support a particular part of the application. ASIPs are described as hierarchical collections of underlying IP blocks, using the TIM language and a mixture of pre-designed IP blocks and possible new designs. The system is composed of the processors and other IP blocks, such as memories and interfaces, and described by the use of the HSD language.

3. The mapping of the subprograms to the processors is performed manually, and it is an iterative process where the details of each processor are refined to efficiently support the mapped sub-program. This refinement may include identification and implementation of additional instructions to speed up the code execution and reduce its memory footprint. In order to evaluate the quality of a mapping, the sub-program is compiled and executed (through simulation) on the processor model. This may lead to changes in the code or processor design (inner loop).

4. When all sub-programs have been mapped to the optimized processors of the system. The whole system can be evaluated through simulation. This may lead to changes in either or both of the application code and system (outer loop).

5. Finally, the optimized system definition is semi-automatically converted into an RTL hardware description, such as VHDL, for further processing.

The European research project ASAM (Automatic Architecture Synthesis and Application Mapping for Customizable MPSoCs) of the ARTEMIS Program aims to propose a coherent automated design methodology for the MPSoC architecture synthesis and application mapping of the MPSoCs based on customizable ASIPs, and related prototype EDA-tool system. The tools for the ASAM design flow will involve new own and existing third-party EDA-tools, and have to enable an experimental research and evaluation of the ASAM methodology and design flow. The new automated coherent design environment should enable the system designers to quickly develop high-quality designs through rapid system-level design exploration, and an efficient and automatic final system synthesis.

The ultimate objective of the ASAM project is to much enhance the MPSoC design efficiency, while not compromising on improving the result quality. This will be realized through an adequate automatic system and sub-system level architecture exploration and synthesis, and uniform automatic HW/SW compilation of the application specifications into their highly optimized application-specific heterogeneous multi-ASIP implementations. The uniform HW/SW design flow will provide efficient exploration of promising parallel structures of the application’s computation processes, promising system and sub-system level architectures, and application mapping alternatives and trade-offs.

The project aims to develop a prototype of a coherent architecture synthesis and prototyping environment for customizable MPSoCs, involving; application analysis and computation process restructuring, multi-objective design space exploration for customizable heterogeneous multi-ASIP SoCs and construction of the application-tailored and technology-aware system-level architectures, multi-objective architecture customization of particular generic ASIPs, communication resources and memory.
structures, adequate hardware synthesis of the so designed hardware system, software compiler re-targeting and (semi-)automatic software restructuring and implementation on the resulting multi-processor system, feedback creation for both architecture design levels through simulation and FPGA emulation.

The project builds on the system-based design of heterogeneous multi-processor embedded systems, and specifically Silicon Hive customizable MPSoC. The quality-driven model-based system architecture exploration and synthesis approach earlier developed by TU/e. The design modeling, exploration, synthesis, analysis, emulation and estimation concepts developed or to be developed by the project partners (mainly: TUD, TU/e, UNICA), and the hardware and software analysis, parallelization and compilation concepts and techniques developed or to be developed by all the project partners and some external parties. The analysis, synthesis and prototyping flow will involve several different kinds of models, many various tools operating on them, and complex collaboration among the models and tools. The success of ASAM heavily depends on the compatibility and coherent collaboration of all the models and tools.

### 5.2 General method

The SiliconHive flow presented in Figure 7 involves some manual and some automated processes. In order to explain the ASAM design flow, we will first introduce a simplified view of the Silicon Hive design flow as shown in Figure 8.

In general, with the Silicon Hive design flow the ASIP-based system architecture generation is automated, the analysis of the system architecture is partially automated (i.e. simulation-based analysis), but the actual design of the ASIP-based system architecture and mapping of the software onto the processors in the system are not automated.

![Simplified Silicon Hive design flow. Items in dark blue are part of the HiveLogic tools and flow. Items in light blue are application dependent.](image_url)
The objective of the ASAM design flow is to extend the capabilities of the Silicon Hive design flow by automating some of the currently manual processes and introducing some new automated processes. Figure 9 shows a simplified view of the proposed ASAM design flow.

![Figure 9: Simplified ASAM design flow. Items in red are the new ASAM processes.](image)

The ASAM design flow consists of the following processes marked in red and coarsely described in Figure 9:

1. **Design space exploration (DSE) for a multi-processor system and its particular ASIPs** [TUE/DTU]: The manual application analysis and restructuring, computing system architecture design and application mapping on the system is substituted by two exploration processes; one for the inner loop ASIP design space exploration (ASIP or micro-level DSE), and one for the outer loop of system level design space exploration (system or macro-level DSE). Each of the DSE processes involves its own application analysis and restructuring, and computing system synthesis, because they have to be performed at each architecture level with different precision and (partially) account for different aspects.

2. **Application analysis** [TUE/DTU/SiliconHive/Compaan]: The application code written in C is translated into an equivalent KPN or other graph-based model exploring the parallelism of the application. Each DSE, at the multi-ASIP system level or at the single ASIP level, has its own application analysis and uses the appropriate graph model to perform the associated parallelization explorations, mainly the task level parallelization for the system level DSE, and the data, instruction and operation level parallelization for the ASIP level.
3. **Rapid prototyping** [UNICA/TUBS]: The simulation is complemented with a prototyping FPGA system that allows for a fast analysis of system execution characteristics. It will be used as feedback to the design space exploration processes, making it possible to explore the design trade-offs with hardware in the loop.

The general task of the ASAM design flow takes, as input, the system requirements for a given application in the form of:

- the original application behavior specification as a sequential C code, and
- the decision model composed of parametric constraints, objectives and trade-off information

and it produces, as output, the best possible to construct (in relation to the system requirements) application-specific parallel MPSoC architecture instance composed of:

- the best possible instance of the top-level MPSoC
- the best possible instances of particular ASIPs
- the corresponding parallel versions of the (parts of the) application behavior scheduled and mapped on the best possible MPSoC architecture instance
- information on the physical parameter estimates (area, delay/throughput, power, etc.) and margins in relation to constraints and objective values

The original application behavior specification is sequential. The resultant application-specific MPSoC architecture instance has to be highly parallel, while efficiently exploiting all needed kinds of parallelism: task or sub-program level, instruction-level and data parallelism. The general task of producing the best possible application-specific parallel MPSoC architecture instance can be decomposed into the following two sub-tasks performed at two different architecture levels:

- find the best possible multi-ASIP system-level architecture instance
- find the best possible architecture of particular ASIP-based sub-systems

At each of the two architecture levels, the *task of finding the best possible architecture instance* decomposes into the following two sub-tasks:

- find the best possible parallel version of (the part of) the application
- find the best possible computing system architecture instance to implement this best possible parallel version of (the part of) the application

Figure 10 gives a more detailed overview of the ASAM design flow and clearly shows the macro- and micro-level DSE within the ASAM design flow. The micro-level architecture exploration of particular ASIPs is tightly coupled with the macro-level architecture exploration of the multi-ASIP system architecture. The macro-level architecture exploration performs the initial application analysis and uses the information from that analysis for partitioning of the application into several parts and assigning each part on a separate ASIP processor of a given multi-processor
system template.

It also usually decides the type of a process of this system to which a particular partial code is assigned. The decision process requires information on various processor features in relation to the (partial) application code when executed on the processor, like execution time, silicon area, and power consumption. It is relatively easy to obtain such information when a processing element is selected in the form of an off-the-shelf specific processor or hardware accelerator from a library, but very difficult when this processing element is an adaptable ASIP because of the large variation in ASIP architectures and their features.

Figure 10. A more detailed overview of the ASAM design flow.

For ASIPs this information can be provided by the micro-level architecture exploration in the form of (best-case, worst-case, or other) execution time estimates, and if possible, other parameter estimates (e.g. area, energy, etc.) for a given (partial) code and type of ASIP. This information on the most relevant metrics could be delivered in the form of a probability density function which would much benefit the macro-level architecture exploration, but methods for obtaining the information require further research. In the initial synthesis steps the information provided by the micro-level architecture exploration represents a rough estimation and is not to be interpreted as a definite figure.

With the progress of the ASIP micro-level architecture exploration more accurate estimates can be given upon request from the macro-level architecture exploration.
for a specific code and ASIP type. This requires a more detailed model of the ASIP which, in turn, depends on a more advanced and detailed ASIP design. At these stages it also becomes possible for the micro-level architecture exploration to give a feedback on the expected area and power for the selected ASIP type. This new information makes it possible for the macro-level architecture exploration to reiterate and improve the application partitioning and processing elements selection.

The highest level of detail is only available when the complete micro-level architecture exploration and ASIP synthesis is performed for the application parts and corresponding processors decided by the macro-level architecture exploration. This last step of the micro-level architecture exploration is very complex and time consuming and should only be performed for a small set of the most promising partitioning cases and the most promising parallel versions of the (partial) application code assigned to a single processor. The result of this last part of the micro-level architecture exploration and synthesis will be an executable model of the application and synthesizable model of the corresponding ASIP, which are ready for both the detailed simulation and synthesis into the corresponding actual hardware.

Summing up, the macro-level architecture exploration will exploit the micro-level synthesis for providing specific exploration, synthesis and estimation services related to a given pair composed of a (partial) application code and an ASIP processor on which the code is supposed to be executed. To ask for the services, the macro-architecture exploration will provide the micro-level exploration and synthesis the input composed of the C-code of the application part, with a set of stimuli for profiling and verification of the application on the ASIP architecture.

The micro-level exploration will perform the required exploration, synthesis and estimation services, and deliver the relevant software and hardware structures and their performance estimations to the macro-level exploration. The micro-level architecture exploration will use the ASAM prototyping environment and the SiliconHive design flow for obtaining the final performance estimations through simulation and/or emulation of the application part on the completed ASIP micro-architecture using a set of stimuli for the application part provided by the macro-level architecture exploration.

The micro-level architecture exploration will output performance estimates (e.g. throughput, area, power) with increasing precision together with the parameters required for the micro-level architecture exploration to reproduce the obtained results during subsequent macro-level architecture exploration design space iterations. At the highest level of detail, the micro-level architecture exploration will also produce a description of the completed ASIP micro-architecture, together with a restructured version of the application part, suitable for synthesis with the ASAM prototyping environment and the SiliconHive design flow, as well as, the related parameter estimates and their distances from the target values.

5.3 Design space exploration for System-architecture synthesis

The System-Architecture (macro-level) design space exploration begins with the analysis of the application code which will show
Public with confidential appendices

- all the possibilities for parallel processing within the application
- the performance requirements of each parallel task (i.e. required operation throughput)
- the communication throughput and latency requirements towards all other tasks.
- indicate whether, and what granularity, synchronization between tasks is required.

The system-architecture DSE phase will propose a set of (partially designed) processors (ASIPs) in a multiprocessor system based on the clustering of tasks that resulted from the application analysis phase. It will make sure that enough processors are proposed to cope with the required performance, as indicated in the application analysis phase.

The application analysis phase also determines how much bandwidth is required between the processing tasks. Thus, when mapping the application on the instantiated processors, the System-architecture DSE phase also instantiates enough communication bandwidth to cope with the requirements. Initially, this can be achieved by instantiating a generic version of the SHMPI Network-on-Chip and specifying the required amount of bandwidth for each connection. In a later phase of the design, these interconnect specifications can also be translated into a balanced set of on-chip buses or point-to-point (streaming) interconnects, as provided with the HiveLogic IP base.

The combination of communication, performance, and synchronization requirements may require that DMA units are selected from the HiveLogic IP base and instantiated on the interconnect fabric.

Summarizing, the System DSE phase has the following interfaces:

1. Inputs:
   User specified:
   - ANSI-C code [Required], with [Optional] HiveCC specific code annotation (vector operations, HiveCC pragmas, etc.)
   - (Partial and optional) Processor descriptions (TIM) specialized and optimized for the given application [can also be an input from the micro-level design space exploration]
   - Stimuli for profiling and running the application (part) and obtaining performance estimates from the prototyping environment [Required]

   From the micro-level design space exploration:
   - Performance estimates (area, power, throughput) on pairs of parts of application code and single ASIP with precision depending on the type of analysis
   - Exploration parameters for the current micro-architecture design exploration (useful as pre-defined exploration parameters when incrementally improving ASIP architectures for the same application part)
Public with confidential appendices

• Restructured and instrumented C code and Processor descriptions (TIM) specialized and optimized for the given portion of the application [Only complete for final analysis]

• Optimized portion of the application [Only for final analysis], also used as output to the Hive logic

From Hive Logic:
- System level IP blocks from the HiveLogic libraries

From prototyping:
- Metrics measurements

2. Outputs:
To the prototyping phase:
- Hive System descriptions
- Restructured and instrumented C-code of all the application partitions
- TIM description of all the ASIPs composing the system

To the micro-architecture exploration:
- ASIP type
- Application partitioning: Partial ANSI-C code [Required], with [Optional] HiveCC specific code annotation (vector operations, HiveCC pragmas, etc.)
- Stimuli for profiling and running the application (part) and obtaining performance estimates from the prototyping environment [Required]
- Pre-defined exploration parameters and constraints for limiting the search space (such as ASIP type, vector size, etc.) [Optional]
- Type of analysis to perform, i.e. required service [Required]

3. Tools
- HiveCC
- Compaan/ACE
- Vector Fabrics
- University Prototype Tools

5.4 Design space exploration for ASIP-architecture synthesis
We decided to sub-divide the ASIP-architecture (micro-level) design space exploration and synthesis into three phases. These phases explore the micro-level architecture at different levels of detail and can be served with different speed, when using different application and/or computing system models. This allows for a fast, but coarse, feedback for the macro-level architecture exploration from the first phase, and slower, but more precise, feedback from the later phases. An early
feedback on the proposed designs is critical for limiting the exploration effort and time of the ASAM flow when working with complex highly-demanding applications. The three phases of the micro-level architecture exploration are the following

1. Phase 1: Application bottleneck and parallelization potential analysis.

2. Phase 2: Application parallelism and initial ASIP architecture exploration.


Dividing the micro-architecture exploration into the three phases of which the first one is coarse, but very fast, the second much more precise, but slower, and the third one the most precise, but relatively time consuming, enables to avoid the repeated execution of the complex and time consuming computations scheduled in the later phases before the macro-architecture and the coarse micro-architecture are actually decided, while still providing a reasonable feedback to the macro-level architecture exploration.

We also took care for the three phases of the micro-level architecture exploration to well correspond to the three general design phases observed in the currently exploited state of the art ASIP design methods.

This is a big advantage. When exploiting our proposed automatized micro-architecture exploration and synthesis flow the designers will not be required to dramatically change the general design methodology, and the patterns of their design activities that they are used to. In contrary, the automatic flow will only much support them in their way of the micro-architecture exploration and synthesis.

The analysis and synthesis processes of the three micro-architecture synthesis phases are available as services to the macro-level architecture exploration, either as single services enabling a fine-grain control by the user of the flow, or as a combination of multiple phases, services, simplifying the process flow (e.g. to a novice user). A micro-level architecture exploration service control layer is added to facilitate the service exploitation. The macro-level architecture exploration communicates with the micro-level architecture exploration service control layer to select the operations to be performed by the micro-level architecture exploration and provide the inputs and constraints. Figure 11 shows the interface between the macro-level architecture exploration and the micro-level exploration service control layer together with a graphical representation of our proposed micro-architecture exploration and synthesis method, subdivided into the three phases of the more and more precise micro-architecture exploration.
Figure 11. Micro-architecture design exploration flow.

The micro-level architecture exploration takes as input pairs of application parts and stimuli. It also takes requirements to lead the exploration, an optional processor type as starting point, and the type of services to be provided to the macro-level. Each exploration phase has its own decision model and adds new precision to the processor design by setting values for some parameters of a generic processor template. The micro-level architecture exploration gives as output partial or completed micro-architectures and additional information on architecture performance and efficiency.

Summarizing the above discussion, we can identify the following interfaces between ASIP-architecture design space exploration and the other stages in the flow:

1. **Inputs:**

   From macro-level design space exploration:

   - ASIP type
   - Application partitioning: Partial ANSI-C code [Required], with [Optional] HiveCC specific code annotation (vector operations, HiveCC pragmas, etc.)
   - Stimuli for profiling and running the application (part) and obtaining performance estimates from the prototyping environment [Required]
Public with confidential appendices

- Pre-defined exploration parameters and constraints for limiting the search space (such as ASIP type, vector size, etc.) [Optional]
- Type of analysis to perform, i.e. required service [Required]

From Hive logic:

- HiveLogic Processor design libraries for individual processor blocks (register files, issue slots, function units, memory devices, etc.) [Required]
- Compilation and simulation results [Required]

From prototyping:

- Metrics measurements

2. Outputs:

To macro-level design exploration:

- Performance estimates (area, power, throughput) with precision depending on the type of analysis
- Exploration parameters for the current design (useful as pre-defined exploration parameters when incrementally improving ASIP architectures for the same application part)

To macro-level design exploration and prototyping:

- (Partial) Processor descriptions (TIM) specialized and optimized for the given portion of the application [Only complete for final analysis], also used as output to the Hive logic
- Optimized portion of the application [Only for final analysis], also used as output to the Hive logic

3. Tools

- HiveCC
- Static analysis tool from TU/e
- ...

5.5 Prototyping

The ASAM tool-flow will include a technology-dependent hardware for generation and prototyping. The main aim of such framework will be to provide support for all the levels of the architectural analysis, reducing the gap between the estimation of the performances considered during the early steps of the design flow and those really measurable after the implementation. The framework will be capable of creating an FPGA implementation of a candidate system configuration. The target application will
be executed on it for emulation, in order to obtain switching activity figures and performance metrics to be considered during the ASIP- and System-architectural optimization processes.

Summarizing, the Prototyping phase has the following interfaces:

- **Inputs:**
  - System and ASIP description (in HSD and TIM languages)
  - Optimized and partitioned application

- **Outputs:**
  - Metrics for System and ASIP architecture DSE

- **Tools**
  - HiveCC
  - RTL Compiler
  - FPGA Synthesis
6 Multi-processor subsystem IP, design flow, and code compilation

This technology is mainly handled by Silicon Hive and is commonly referred to as ‘HiveLogic’. HiveLogic is a system-level set of IP and tools that are configured into ANSI-C-programmable SoCs for different application domains. This section provides a high-level overview of the current state of this flow and the extensions needed for the ASAM project. This section also describes the components of HiveLogic and the way it will be applied. As such, it also describes the current state-of-the-art in automation of system-level design capabilities for complex application-specific heterogeneous multi-processor SoCs.

HiveLogic provides a set of IP blocks based on which users construct their own Electronic System-Level (ESL) designs. Such designs would consist of combinations of application-specific processors (ASIPs or programmable accelerators) and other ESL IP. In order to achieve required performance levels at small silicon area and low power dissipation, HiveLogic-based multi-processor subsystems are typically focused at computational efficiency\(^1\).

The actual design of the processors, multi-processor systems, and application mapping is a manual task. The ASAM design space exploration tools will – to a large extent – take over these manual tasks.

Summarizing the HiveLogic phase in the ASAM flow, the following interfaces are identified:

- **Inputs:**
  1. ASIP-architecture DSE $\rightarrow$ TIM descriptions of processors
  2. System-architecture DSE $\rightarrow$ Hive System Descriptions
  3. System-architecture DSE $\rightarrow$ Mapped application code
  4. NoC design flow $\rightarrow$ NoC system-level block

- **Outputs:**
  1. System-level IP blocks $\rightarrow$ System architecture DSE
  2. Processor Cell RTL $\rightarrow$ Prototyping
  3. Processor Cell RTL $\rightarrow$ VLSI
  4. System RTL $\rightarrow$ Prototyping
  5. System RTL $\rightarrow$ Prototyping
  6. Executable code for processor Cells $\rightarrow$ Prototyping
  7. Executable code for processor Cells $\rightarrow$ VLSI

Please refer to Appendix A on page 61 for further in-depth information on the SiliconHive flow.

\(^1\) Background information on how the combination of all HiveLogic features results in high computational efficiency is assembled in section 12.
7 Compaan Design

7.1 Compaan Technology

The problem Compaan Design address is the process of mapping C-code onto an FPGA, as Figure 12 illustrates. Writing the C-code is quite straightforward; it is reasonably easy to validate that this is the correct code. Should problems occur, good debuggers exist to help locate the problems. Mapping this C-code onto an hardware is difficult. C code is sequential, with a single thread of execution and shared memory in mind. The references to variables y(i) and z(i) implicitly refer to variables that reside in a large, global memory space. To take full advantage of the parallelism inherent to hardware, one should exploit concurrency. This translates into to the possibility of identifying threads of control. In addition, typical hardware architectures do not have singular large memory pools, but rather scattered strips of distributed memory. The conclusion to be drawn is that applications are unlikely to map well onto ASAM’s architectures due to the assumption of global memory in the semantics of the C language and the dominance of the meme of single threaded code and coding.

Instead of writing C, a designer could also have written a parallel application representation. In Figure 12, right hand side, a network of processes exchanging data using FIFOs (First In First Out buffers) is shown. This is called a Kahn Process Network (KPN). Once a designer has such a partitioned parallel application representation, the mapping onto a hardware or heterogeneous architecture is more straight forward. A Kahn process becomes either a microprocessor or a hardware accelerator. A FIFO connection is mapped to the distributed memories provided by...
native FIFO support. The downside is that writing an application in the KPN form is not simple. For a designer it is difficult to debug such an application representation, as things happen in parallel. In addition, deciding how to partition an application over threads is not easy. The Compaan compiler allows the designer to express an application in the format he knows well; a C application that is subsequently converted into an equivalent KPN representation - as shown in Figure 13. Next, this KPN representation can be mapped onto an FPGA in a straightforward manner.

An advantage of the Compaan Technology is that C code is translated into an equivalent KPN representation using mathematical techniques based on the polytope model. This model is a geometric representation of the original computation. For this model, very powerful techniques exist implemented as open source libraries that Compaan Design exploits to perform the translation. Using these packages, Compaan Design taps into one of the world’s best research in polyhedral mathematics and integer linear program solvers. The use of the polytope model limits the translation to a specific class of programs called Parametric Nested Loop Programs. Any Matrix-Matrix or Matrix-Vector program can be expressed in this way. Most signal processing applications fit the Parametric Nested Loop Programs and oftentimes we can rewrite dynamic code to Parametric Nested Loop Programs.

Compaan focuses on C-code as in the embedded space, C is still the dominant programming language (as opposed to Java for example). Most reference code for new standards in communications or media fields, are written in C. Embedded programmers like C-code as this means they can specify at a very low level what needs to happen on a microprocessor and conversely, one can inspect at low level what happens in the processor.

Since the translation is mathematical without the use of heuristics, there is a one-to-one correspondence between the C-code and the resulting KPN. Compaan developed
a technique called tracing, which is used to prove that both programs are indeed equivalent.

Parallelism comes in different forms: data-parallelism, instruction-level parallelism and task-level parallelism. Compaan Design focuses on task-level parallelism that can further be sub-divided into Fork/Join parallelism and pipeline parallelism - as shown in Figure 14. Fork/Join parallelism is among the simplest design techniques for obtaining parallel performance. This type of parallelism is more frequently seen in, for example, High Performance Computing, where OpenMP has found fertile soil. It tends to be applied to systems that contain many homogeneous cores where its effect is to optimize for latency.

![Fork-Join Parallelism and Pipeline Parallelism](image)

*Figure 14 Fork/Join Parallelism versus Pipeline Parallelism*

Compaan, on the other hand, generates pipeline parallelism. This form of parallelism is crucial when processing large volumes of data where throughput is the key metric. The best results are obtained when low overhead core-to-core communication is available, which is the case with the fast FIFOs found on today’s high-end FPGAs. The networks that Compaan delivers are ideally suited to take advantage of this form of communication.

It looks as if Compaan compilers can only generate a single KPN. However, an instance of a Compaan compiler can generate an indefinite number of different KPNs with the same input-output behavior. Each KPN will have different characteristics i.e. numbers of channels and processes. One can view this as expressing the C code (the functional description) in different degrees of parallelism.

Figure 15 depicts a Compaan compiler generating alternative KPNs for a single piece of C code. Suppose that the middle network is the default network obtained when running the Compaan compiler. Then, by using a *merging* technique, one can reduce the number of processes and thereby reducing the level of parallelism. The reverse is also possible, by *splitting* a process in new processes thereby increasing parallelism. By repeating this process in both directions, the full range can be covered, from no-parallelism to full parallelism, for a single piece of code.
Figure 15 Playing with parallelism is the game

The KPN network which Compaan derives is a model rather than a specific implementation. It captures the essence of a hidden computation structure within the C-code in terms of its parallelism. The next step in a design process is a hardware implementation or mapping onto heterogeneous multi-processor ASIP. The designer now makes choices how to express processes, i.e. in hardware or software, using a mixture of hardcores, softcores\(^2\), or custom hardware accelerators.

Figure 16 A Process is a model; a designer can chose between a software implementation on a microprocessor of hardware implementation on a hardware accelerator

Figure 16 shows how a process from a KPN can either be mapped onto a microprocessor (e.g., a Xilinx MicroBlaze), onto a hardware accelerator or onto an

\(^2\) A softcore is a microprocessor expressed in the reconfigurable fabric of an FPGA. A hardcore is true microprocessor embed in the FPGA.
ASIP. In case of the microprocessor or ASIP, the process is expressed as a piece of software (i.e., a software thread) that is executed on the microprocessor. By implementing a small OS with multi-threading support, multiple processes can be mapped onto microprocessors. For ASIPs, this is not the case. In case of the hardware accelerator, a wrapper is provided that consists of a Read/Execute/Write unit. The Read and Write unit are needed to get the right data at the right moment. The read data is provided to an IP core that is typically pipelined. Once data comes out of the Execute Unit, the Write Unit picks up the data and sends it to the next processor in the network. The Read/Execute/Write structure fits naturally with the pipeline behavior of the overall KPN network. This is illustrated to the right of Figure 16. It shows how a 3-stage IP core will behave in the hardware accelerator. Consequently, the hardware accelerator accepts a single token at every clock cycle, making the design high-performance. A unique feature of the hardware accelerators and ASIPs is that the read and the write unit operate independently of one another. Therefore, they can very easily handle deeply pipelined IP cores and naturally allows for bubble compression in the pipeline. Bubble compression means that some 'E' blocks will be empty showing up as holes or bubbles. Scheduled IP cores have problems with these holes as they disrupt the global schedule state space. As a result, the IP core has to stall as soon as a bubble appears leading to delay. The Compaan hardware accelerators and the ASIPs do not have this problem as the Read and Write Unit operate independently of each other. The IP core can continue to operate when a bubble arrives, as the Write state does not affect the Read state.

The FIFO in the KPN model is also a model. In the mapping process, we can choose to implement it as a FIFO. However, it is also possible to map the FIFO onto a shared bus structure, in shared memory or by using a Network on a Chip (NoC) structure.

### 7.2 Compaan Product

Since September 2007, Compaan Design has been assembling a design flow that implements the presented Compaan technology. The design flow analyses an application written in ANSI-C code and converts it automatically into a streaming, pipelined architecture that can subsequently be mapped onto FPGAs. Figure 17 shows this flow that consists of a number of tools. It shows that the Compaan compiler converts an application written in C-code into a KPN model (expressed in XML for ease of use). KpnMapper accepts this KPN model, which is the back-end technology of Compaan Design. KpnMapper requires a system and a mapping specification. In the system specification, a designer indicates the resources to be used on an FPGA. The designer is able to use a mapping specification to indicate which process of a KPN is mapped to which resource in an FPGA. A resource is either a Microprocessor (i.e., a Microblaze or a PowerPC) or hardware IP Core. KpnMapper generates all the files needed to create a KPN on a Xilinx FPGA using the EDK (Embedded Development Kit) tool. EDK is the Multiprocessor design environment of Xilinx. KpnMapper instantiates, integrates, and programs all the microprocessors and hardware accelerators by generating EDK specific files. Using standard synthesis and place&route tools of Xilinx, a bit stream is created that programs an FPGA.
The complete Compaan design flow shown in Figure 17, is integrated in a graphical design environment to make the tool flow easily accessible. This graphical environment is in fact the Compaan product; it provides editors, graph visualization tools, next to an integrated help, and update mechanism. The update mechanism makes it easy to update the Compaan product to the latest version over the internet.

The Compaan Compiler makes use of CoSy, which is ACE’s compiler development system. The choice for this piece of infrastructure was made for two reasons; faith and strategic nature: designers need to have faith that the compiler they use produces correct code. CoSy is well established in the market and is a product with industrial strength. The use of a commercially supported infrastructure means that Compaan can focus its engineering resources on its product and does not need to develop and maintain a large and complex infrastructure. Compaan Design receives all updates and improvements to CoSy and can incorporate these into its own product. This contributes to keeping the Compaan product up to date and stable. Also, due to the module structure of CoSy, Compaan Design can work on its own engines integrating techniques it needs without having to worry about the stability of the other components as this is done by ACE.

An additional factor, that is becoming increasingly apparent with the advent of multi-core systems, is the strategic nature of compiler technology. This is not only a matter of make or buy, but of structure, engineering model, and capabilities. In the path from application to realization, the executable specification in C is often the only moment in the flow that all information about an application is known and available. All this information is available during compilation in the form of particular
data structures. Having access to these data structures and manipulating them in a consistent way is essential to solve any multi-core problem. CoSy provides a modular framework in which software engineers can access the data structures and manipulate them in a consistent way.

7.3 Example

To give a feeling what you can express with the Nested Loop Affine language the Compaan tool accepts, we now show an example that shows how to express code that is converted in a task-parallel description. This example demonstrates how to describe a situation where a data stream is split in multiple sub streams that are processed in parallel and then merged again in a single stream.

![Figure 18 data parallelization](Image)

7.3.1 The Code Example

```c
#define MAX_I 100
#pragma compaan_procedure parallelization_example
void parallelization_example(int data_in[MAX_I], int data_out[MAX_I]) {

    int i;
    int j;

    const int P = 4; //parallelization factor
    const int MAX_IP = MAX_I / P;

    int a[MAX_IP][P];
    int b[MAX_IP][P];

    for (i = 0; i < MAX_IP; i = i + 1) {
        for (j = 0; j < P; j = j + 1) {
            a[i][j] = data_in[P * i + j];
        }
    }

    for (i = 0; i < MAX_IP; i = i + 1) {
        for (j = 0; j < P; j = j + 1) {
            b[i][j] = Transform(a[i][j]);
        }
    }

    for (i = 0; i < MAX_IP; i = i + 1) {
        for (j = 0; j < P; j = j + 1) {
            data_out[P * i + j] = b[i][j];
        }
    }
}
```

ASAM Deliverable_1.2_v1.0.doc Page 39 of 101
### 7.3.2 How the code works

The code starts by defining the symbol MAX_I which indicates how many tokens an array contains. Then the network function parallelization_example is created which provides the external data arrays data_in[MAX_I] and data_out[MAX_I]. After initializing some variables, the variable P=4 is defined. This variable is used to define the ‘parallelization’ factor. The parallelization factor defines how the MAX_I tokens are processed over P streams. The number of tokens per stream are defined by variable MAX_IP and is used to define the arrays a[i] and b[i].

Using the linearization function P*i+j, data from array data_in[] is put in four different arrays a[i][]. Then the function ‘transform’ is executed in parallel on the P streams. Since P=4, the transform function is unrolled 4 times. Array a[i][0] is the first stream, a[i][1] the second stream etc. The result of applying the function transform is stored in array b[i][] for each stream. Finally, the four different versions of b[i][] are merged back into the single 1-D array data_out[] that is streamed out. Again a linearization technique is used to moved a 2-D array into a 1-D using the function P*i+j.

The KPN graph we obtain for example parallelism is given in Figure 19.

*Figure 19 KPN Graph for Data Parallelism*
8 Design Space Exploration

In order to allow efficient system level design, a flexible framework for performance estimation providing fast and accurate estimates is required. Several methods have been presented in recent years allowing performance estimation through formal analysis or simulations of architectures at high levels of abstraction. Recently approaches that rely, at least partly, on formal methods of analysis in order to allow performance estimation have been presented. In theory, these approaches eliminate the need for simulations in order to predict performance. However, in most cases, the accuracy of these approaches only justifies their use in the very early stages of the system design phase where they can be used to reduce the number of potential candidate architectures and the detailed performance estimates are obtainable only through simulation in the later design stages.

The majority of the approaches based on fast simulations are using high speed instruction set simulators with high level modeling of data memories, caches, interconnect structures, etc. performing a number of abstractions and thereby trading accuracy for simulation speed. These approaches have their merit, especially in the early design stages, and often even allow software developers to start the target specific software development in parallel with the hardware developers - long before low level register transfer level descriptions of the system exist or the actual hardware prototype bring-up.

The high level models fulfill the needs for early software development and initial architectural exploration. However, in many cases one needs to be able to generate highly accurate performance estimates in order to reason about the actual performance of the system so as to verify architectural design choices. In order to do so, cycle accurate models are required implying that, currently, register transfer level descriptions of the architectural elements of the target system is often the only viable solution. The simulation of large-scale systems described at the register transfer level, however, suffers from tremendous slowdown in the simulation speed compared to the high level simulations.

Even worse, the development of such detailed descriptions is long and costly implying that when these are finally available, often at a very late stage of the development phase, changes of the architecture are very hard to incorporate resulting in limited possibilities for design space exploration.

Thus there exists a gap between the fast semi-accurate methods that are highly useful in modern design flows allowing the construction of high-level virtual systems, in which rough estimates of the performance of the system can be generated, and the detailed and very accurate estimates that can be produced through register transfer level simulations.

DTU has developed a compositional framework for system level performance estimation, for use in the design space exploration of heterogeneous embedded systems. The framework is simulation based and allows performance estimation to be carried out throughout all design phases ranging from early functional to cycle accurate and bit true descriptions of the system. The key strengths of the framework are the flexibility and refinement possibilities as well as the possibility of having components described at different levels of abstraction to co-exist and communicate within the same model instance. This is achieved by separating the specification of functionality, communication, cost and implementation and by using an interface-based approach combined with the use of abstract communication channels. The interface-based approach implies that component models can be seamlessly interchanged in order to investigate different implementations, possibly described at dif-
different levels of abstraction, constrained only by the requirement that the same interface must be implemented. Additionally, the use of component models allows the construction of component libraries with a high degree of re-usability as a result. The macro DSE will use this framework as a starting point, focusing on developing a DSE tool that will ease and speed-up the application and platform refinements, as well as the computation of a mapping able to meet the designer’s constraints.

8.1 Overview of the Framework
The framework presented, illustrated in Figure 20, is related to what is known as the Y-chart approach. Starting from the application and platform models (respectively described in Deliverable 2.2 and Deliverable 2.1), the mapping and schedule are defined. This result is then analyzed and/or simulated and the mapping, the application and platform model can be refined according to it. The main purpose of the application model is to capture the task level parallelism; a refinement in the task graph model can correspond to:

- splitting one or more tasks in order to extract additional task-level parallelism that can be exploited at macro-level DSE
- combining two or more tasks in a unique task if the mapping and scheduling show that a higher level of granularity in the application representation can be useful

For what concern the platform model, this can require an update in the type/number of cores and the interconnection network among them. Any of these updates will require a new mapping and scheduling calculation. This refinement process should be repeated until reaching satisfactory results and moving to the implementation phase.

8.1.1 An example of framework modeling: the Service Model
A possible way to model the mapping of the application model onto the platform model is through the notion of services that plays an important role in decoupling the specification from functionality, communication, cost and implementation. A service is defined as a logical abstraction that represents a specific functionality or a set of functionalities offered by a component. In this way, services are used to abstract
away the implementation details of the functionality that is offered by the component. Thus, the service abstraction allows two different models to offer the same services, having the same functional behavior but with a different implementation, cost and/or latency associated and, so, allow different implementations of a model to be investigated easily.

The functionality of the target application is captured by a Task Graph model. Application models are composed of a number of tasks that can be represented as service requests. The tasks serve as a functional specification of the application and specify a partial order of service requests needed to capture the functionality and the level of parallelism of the application. No assumptions on who will provide the services required are made and, thus, separate the specification of functionality and implementation.

The target architecture is modeled by the PCU Model (deliverable D 2.1). Each Processing and Communication Unit provides a service that can satisfy one or more service requests by the application's tasks and the communication among them. The service models of a platform model can be described at arbitrary levels of abstraction implying that in one extreme they only associate a cost with the execution of a service request or, in the other, the service request is modeled in the platform model both cycle accurate and bit true. Costs can be associated with service requests whether computed dynamically or pre-computed. It is the cost of the execution of a service that differentiates different implementations of the particular service. The service model at the higher level of abstraction can well fit with the requirement of macro-level design space exploration and at the same time at a lower level can find application in the micro-DSE.

Quantitative performance estimation is performed at system level through the simulation/analysis of a system model. A system model is constructed through an explicit mapping of the components of an application model to the components of a platform model. The components of an application model, when executed, request the services (processing and communication) offered by the component onto which they are mapped, modeling the execution of the requested functionality, taking the implementation specific details and required resources into consideration and associates a cost with each service requested. In this way, it becomes possible to associate a quantitative measure with a given system model and, hence, it becomes possible to compare systems and select the best suited one from a well-defined criteria.

In the next paragraphs the application and the platform model used in the framework are described.

8.1.2 Application Modeling

Applications are represented by task graph models that are composed of an arbitrary number of parallel executable tasks (T) and message tasks (MT), as presented in deliverable Deliverable 2.2.

From a formal point of view, each task vertex represents a unit of work that can be performed, it means that different levels of granularity can be depicted (single instruction, set of instructions, etc.). In our case, each vertex represents a piece of sequential code that composes our application and that allow identifying the task level parallelism present in the application.

The message tasks (MT) represents instead the message passing between two tasks, i.e., communication can only be done through message tasks. The motivation for introducing this type of node in a task graph model is to ease the mapping process of the application on the PCU model representing the HW platform. When two communicating tasks are mapped onto different processing units of the platform, the data to be transferred, i.e. the message task, has to be mapped to the communication in-
framework of the platform. This may be a mapping to a single bus, a shared memory unit, or a more elaborate path including multiple buses/links and switches. The separation between computation and communication allows us to focus on the cost of the communication between tasks mapped on different processing units. On the other hand, once a mapping is established, the message tasks between tasks mapped on the same processor could be removed, considering the communication time to be zero (at least as a first approximation).

The edges between tasks impose a partial order of nodes execution and in particular characterize the data dependences (a task can start its execution only when the data provided by its input edges is available).

The information that is available for each task is the type of computation/operation performed (service requests in the service model) and the amount of data that a task can generate at its output edges.

The task graph model is built starting from the Kahn Process Network generated by the Compaan compiler: this can model the different levels of parallelism existing in an application defined as C code; an important step in the translation from KPN to task graph will be to identify the appropriate degree of parallelism for the macro-level DSE. As previously described the tuning of the application model can be done after the analysis/simulation of the mapping results or can be a decision taken statically according to the type of applications or the HW platform available. Moreover the task graph model can also be adjusted thanks to the information provided by the micro-level DSE.

In a KPN the minimum unit of communication between different nodes is a token. The number of tokens in the Compaan KPN can be different (depending on the parallelism). Moreover, they can have different sizes. One possible way to represent the tokens inside the task graph is to associate one or more of them to the message task.

The task graph model can be executed and used for verifying the functional behavior of the model; however, at this level of abstraction there is no notion of time, resources or other quantitative performance estimates. In order to obtain these, the tasks must be mapped to the processing and communication elements of the platform model.

8.1.3 Architecture Modeling

The goal of the platform model is to capture a specific implementation of the functionality offered by the target architecture at macro-level. As presented in deliverable D2.1, a hierarchical graph model called PCU Model is used. The PCU Model gives a representation of the components with data processing capability (processing units, PUs) and of interconnections (communication units, CUs).

For example ASIP processors, HW accelerators and memories are modeled as PUs and the interconnections among them can be modeled as a CU if a bus is used as interconnection network or as a combination of PUs and CUs if a more complex network is used.

The hierarchical representation of the PCU Model allows extending its use also at micro-level modeling.

The platform model, in particular every single processing and communication unit needs to declare the functionalities (services in the service model) that can offer. This can be done providing an estimation (in the form of probabilities) of the performances that each task can reach if mapped on a specific processing unit (or that a message task can achieve when mapped on PU/CU composing the interconnection.
network). These estimations can be obtained through a first analysis at micro-level and can provide information about the parameters that are involved in the design space exploration (e.g. power and execution times). Moreover also static information, not related with the mapping should be available (e.g. static power and area of a PU). A task could be mapped on more than one PU (according to the functionalities required/provided); the differentiating factor will then be the cost associated with the execution of the tasks on each of the candidate processing unit.

The platform model should be build starting from the Silicon Hive description of the ASIPs core, IP blocks and interconnection networks that compose the target architecture. Moreover also the platform constraints coming from the designer should be considered in the realization of the model.

### 8.1.4 System Modeling

A system model is constructed by mapping the tasks that compose the application onto the processing unit available in the platform model.

In case message tasks are included in the application model, these can be mapped on the interconnection network and can ease the mapping and scheduling DSE processes, enlightening also the costs/requirements of the communication.

If multiple tasks are mapped onto the same processing unit, a scheduler must be provided as part of the platform model.

As already mentioned the mapping design space exploration is based on:

- application model
- platform model
- estimation of performances that a specific task mapped on a certain processor can reach

This last information is available only as probabilities and should be provided by the micro-level analysis. The macro-level framework that is used in the realization of the system model works according to an iterative process and computes a mapping that match with the designer requirements. If these requirements cannot be met in simulation/analysis phase, then the framework iterates and recalculates a new mapping. This iterative process can involve not only a new mapping calculation, but also an update of the application and platform models: the task graph can be modified in order to capture different levels of parallelism, meanwhile the platform model can be changed (type and number of cores/interconnections) so that can better fit with the designer's constraints.

Additionally, an iteration not visible in the Y-chart previously presented involves the refinement of the performance estimations proceeding from the micro-level DSE. The purpose of this gradual refinement is to reduce initially the mapping design space at macro-level with estimation that can be rapidly computed and to allow the calculation of more precise performances only for a limited amount of (task, processor) pairs. Another reason for working with probabilities is to try to capture the different performances that can be provided by an ASIP core: in fact its micro-architecture (number of functional units, register files, issue slots, etc.) is tuned during the micro-level DSE and generates different ASIP's configurations with different performances. Having an estimation of the performance that can be potentially reached at micro-level will allow a better macro-level DSE.
9 ASIP Customization

This section briefly describes the high-level requirements on the ASIP customization process of the ASAM design methodology and design flow for complex embedded heterogeneous ASIP-based multi-processor systems (over-viewed in Section 5 and graphically represented in Figure 9 and Figure 10). The requirements presented in this section drive the definition of the ASIP Customization process, as well as, of its interfaces and collaboration with other processes within the ASAM and Silicon Hive flows. This section is not a requirement specification of any particular ASIP customization method or tool. The customization methods and tools that will be developed within the ASAM project will all have their own requirement specifications developed in the successive months of the project.

Based on an analysis of the application communication bottlenecks, parallelism exposed through the KPN-based application restructuring, and other application features and requirements, as well as, possible computation micro-architectures, the macro-level DSE proposes the system-level multi-processor architecture through specifying the proposed ASIP processors, memories, their communication, and the distribution of the required application parts to particular ASIP processors. In result each of the ASIP processors proposed is assigned a part of the application’s C code. Information on the possible performance of the application part when mapped to an ASIP can greatly improve the quality of the exploration result. Therefore, the ASIP customization and implementation process provides different services to the macro-level DSE, each of the services provides feedback on the performance metrics of a proposed application part at different levels of precision. This allows coarse, but fast, estimations for early exploration and more precise, but more time consuming, estimations for the more advanced exploration stages.

The macro-level DSE provides the part of the application’s C code assigned to be executed on the ASIP, as well as, parametric requirements related to the (ASIP, partial C code) pair (e.g. RT computation speed, throughput, area, power) to the ASIP Customization and Implementation process and selects the service of the micro-level DSE according to the precision of the metrics required in the current point of the macro-level exploration. The micro-level DSE is performed separately (and in parallel) for each of the (ASIP, application part) pair.

Based on the parallelism characteristics of the application, the micro-architecture exploration will deliver performance predictions of a proposed (application part, ASIP architecture) pair, and a set of (partial) processor designs that satisfy the performance requirements of a certain application part.

In this micro-architecture exploration, it is essential to exploit the following types of parallelism:

1. Data parallelism (SIMD architecture): when repeated sequences of operations are performed on different data elements

2. Instruction-level parallelism (VLIW architecture): issuing multiple operations per instructions.
Public with confidential appendices

Operation-level parallelism: application domain specific custom operations increase computational efficiency by packing multiple simple operations, typically when they can run in parallel, into more complex operations.

Due to communication overhead, it is usually beneficial to search for the largest set of parallel tasks that can still be efficiently mapped onto a single processor.

Based on the application bottleneck and parallelization analysis the micro-architecture exploration should decide the proper instantiation of all the ASIP resources, and particularly the following:

- **Register files**: the most straightforward solution would be to instantiate a single register file with as many registers as needed to hold all local data, with as many read ports to provide parallel inputs to all issue slots, and as many write ports as there are outputs of issue slots. However, the register file implementation with too many ports can be too costly, and several smaller register files can result in a more economic addressing structure. Consequently, in general a processor needs to have multiple register files. In practice, it is likely that each issue slot can be connected one or more register files. Register files then typically have local interconnects to only the function units of certain specific issue slots. The ASIP architecture DSE step has to adequately decide the set of register files and interconnects between register files and issue slots. The system of interconnect, as specified between register file read ports and function unit inputs, is referred to as: Argument Select Network.

- **Local memories**: For memories, a similar reasoning applies as for register files. However, in the case of memories, the actual instantiations are less fine grained and more directly related to application requirements than register files are. The data flow graph of the application will indicate which arrays of data are accessed in parallel. From this kind of analysis, the number of local memories can be derived. When instantiating memories, the ASIP-architecture DSE phase needs to take into account that load/store function units (LSUs) are needed to access these memories. The LSU associated with a certain memory has to be placed near (in terms of register file access) to the function units that require access to this memory.

- **Result Select Network**: determines where the results from particular issue slots are routed to. Typically, they get at least routed into write ports of the local register files, associated with these issue slots. However, some output paths to some other register files need to be instantiated too if for instance, these are register files associated with issue slots performing some future operations on the produced data. Thorough DSE is needed to prevent the Result Select Network to grow into a fully connected network. To reduce the schedule length on a wide VLSI processor, it is preferred to have dedicated or less congested connections between the Load/Store functional unit Issue Slot outputs and Register file write ports.

The **ASIP Customization and Implementation** process, using this input, as well as, an additional input information from the Silicon Hive design flow on the processor templates, processor building blocks and their characteristics, and further more
precisely analysing the partial application’s C code assigned to the ASIP and its various possible parallel versions, proposes a (partial) instance of the ASIP customized to the partial application’s C code assigned to it, when addressing the parametric constraints and objectives. The level of this exploration, and the completeness of the ASIP instance depends on the service selected by the macro-level DSE. Three levels of exploration will be realized in the micro-level DSE: coarse (application bottleneck and panelization potential analysis), medium (application parallelism and initial ASIP architecture exploration), and fine (ASIP refinement). Partial customization proposals will be analysed using static analysis tools developed by TU/e and the complete customized ASIP instances are simulated and emulated using the prototyping environment which computes the related parametric characteristics and feeds them back (short feedback loop) to the ASIP Customization and Implementation process. Using the parametric characteristics, the ASIP Customization and Implementation process checks if the required constraints are fulfilled and checks to what degree the objectives are satisfied. If the result of the ASIP Customization process is not satisfactory, the (ASIP, partial C code) pair is further refined to more efficiently perform the mapped C sub-program, and the short feedback loop is used to get information on the characteristics of the refined processor. The ASIP Customization process of a given ASIP stops, if its result is satisfactory or it turns out to be impossible to fulfil the hard constraints or optimize the objectives to a satisfactory degree with the available resources.

At this moment, the ASIP Customization process feedbacks a report on the customization results to the system-level DSE process. The system-level DSE process may then decide to simulate and emulate the total system in the prototyping environment that computes the related parametric characteristics and feeds them back (long feedback loop) to the system-level DSE process (only in the case if the result of the ASIP customization process is satisfactory for all the (ASIP, partial C code) pairs involved), or alternatively, the system-level DSE may propose another system-level multi-processor architecture and ask the ASIP Customization process to perform its customization. The whole iterative MPSoC architecture design and application mapping process stops if its result is satisfactory or it turns out to be impossible to satisfy the application requirements to a satisfactory degree with the available resources.

From this brief description it should be clear that the ASIP Customization and Implementation process plays a crucial role in the whole design process of the ASIP-based heterogeneous MPSoCs, through transforming a coarse abstract MPSoC architecture and application mapping proposed by the system-level DSE into the actual hardware design of the MPSoC computation and actual design of software that will run on this system, but also through providing the macro-architecture level DSE with information on characteristics of particular (ASIP, partial C code) pairs what enables the system-level DSE to validate the earlier taken decisions or to take new better informed decisions and reiterate the architecture synthesis process.

The collaboration between the macro-level and micro-level architecture synthesis, as well as, the work of each of the DSE exploration processes of these two levels, will be thus a specific implementation of the quality driven design decision matching process as described in [6].
The **ASIP Customization and Implementation** process will use as its **input** the following information:

1. From the system-level DSE process:
   - the number of processors and type of each processor
   - part of the application’s C code assigned to be executed on each of these processors accompanied by stimuli for profiling and running this part
   - parametric requirements related to each (ASIP, partial C code) pair
   - the requested service (precision level) at which to perform the exploration
2. From the existing Silicon Hive design flow (particularly from its IP repository):
   - HiveLogic IP blocks, and particularly, processor templates and processor building blocks, including Base IP Blocks (processors, processor slices, specific function units, specific memory devices etc.), and their parametric characteristics
3. From the Prototyping Environment:
   - feedback on parametric characteristics (computation speed, area, power) of the partial customization proposals and the complete customized ASIP instances with related software.

The **ASIP Customization and Implementation** process will produce as its **output** the following information:

1. For the system-level DSE process:
   - feedback on the customization results obtained by the ASIP customization for the (ASIP, partial C code) pairs and their related parametric requirements of a the system-level multi-processor architecture proposed by the system-level DSE (this feedback will include references to the processor and corresponding parallel C code instances being the results of the ASIP customization process).
2. For the existing Silicon Hive design flow (particularly for its IP repository):
   - designs of the ASIP instances and their building blocks, as well as, the corresponding parallel C code instance, being the results of the ASIP customization process.
3. For the Prototyping Environment
   - partial customization proposals and complete customized ASIP instances with related software and stimuli for profiling and running.

Using the above specified input, the **ASIP Customization and Implementation** process will perform the ASIP customization in the following way.

The combined Silicon Hive/ASAM design flow is aimed to support its users in
construction of the ASIP-based application-specific heterogeneous MPSoC designs involving customized instances of programmable ASIPs with corresponding customized parallel C code instances running on them, as well as, customized memories, application-specific hardware accelerators and/or other (own or third-party) IP blocks communicating through application-specific interconnection structures. Being a crucial part of this flow, the ASIP Customization and Implementation process will support the users in performance of the following tasks:

1. application code analysis for the code parts assigned to single ASIP processors and restructuring of the code for application- and technology-specific parallel implementation;

2. optimization opportunities identification for different ASIPs and application code parts assigned to them, when accounting for the exploitation of the task-level, instruction-level (VLIW) and data parallelism, code vectorization, loop optimization, pipelining, custom operations and possibly some additional kinds of optimization opportunities;

3. the actual optimized customization of the particular ASIP processors and application code parts assigned to them through adequate instantiation of the particular processors, distribution of datasets to support partitions running on different processor VLIW data paths, mapping specific parts of algorithms on custom operations and other processor extensions and optimizations, as well as, corresponding modifications of the application code parts assigned to the processors, when using various optimization strategies, exploiting information on parallelism identified and optimization opportunities, as well as feedback on the performances;

4. iterative improvements of the custom (ASIP, partial C code) pairs through their (small) modifications;

5. the actual custom construction of the particular ASIP processor instances, their extensions and application code parts;

6. evaluation of the customization results in relation to the requirements imposed on each of the (ASIP, partial C code) pair related to the performance and resource (e.g. area, power) consumption requirements;

7. providing feedback on the customization results for the system-level DSE process based on the above evaluation.

The automatic support does not mean a complete elimination of the user intervention in the ASIP customization process. The users may (re-)assign some code parts to certain processors, suggest some application code restructuring and optimization opportunities, may decide some parameters for processor instantiation or control the scope of automatic generation of custom operations or other processor extensions, including an explicit select of certain custom operations. In this way, the ASIP customization and implementation process will deliver a semi-automatic design methodology that will allow users with varying expertise degrees to very quickly converge to satisfactory optimized custom ASIP instances and code part instances assigned to them.

HiveLogic provides four broad categories of IP blocks: processor templates,
processor building blocks, system-level building blocks, and debug infrastructure blocks (see Chapter 6). These IP blocks can be selected, configured, interconnected (also together with the user’s own or third-party IP blocks), simulated, emulated and evaluated to construct the multi-processor computing systems addressed in ASAM. (Parts of) the application C code can be mapped and implemented on such constructed computing systems or their single processors, correspondingly, and executed in the ASAM FPGA-based prototyping environment. Of these IP blocks, the processor templates and processor building blocks are of main interest to the ASIP Customization and Implementation process. A processor template is an IP block providing the top-level design of any HiveLogic processor. It has a specific fixed hierarchy and defines the interfaces between the constituent processor IP components. All processor building blocks and customer-designed blocks have to fit particular hierarchy levels within the processor template.

The Template Building Blocks are the generic components of a processor, such as sequencer or issue slot. HiveLogic contains amongst others the following Template Building Blocks: sequencer, instruction decoder, loop cache, register file, issue slot, function unit, logical memory, memory arbiter, address map decoder, memory device, and data routing network (for more information see Chapter 6). The Base IP Blocks are the default elements, used to implement basic processing functionality, such as arithmetic units or memory devices. Base IP Blocks are provided at several levels of granularity: base processors, processor slices, specific function units, and specific memory devices. Base processors are constructed through instantiating and interconnecting one or more base processor slices. A processor slice is a re-usable cluster comprising a complete vertical slice through the Processor Template (i.e. comprising register files, interconnect, issue slots, and logical memories). The base processor slices make use of Template Building Blocks. The base processor slices are especially built to be composable and extensible. The processor slices and all their building blocks (e.g. operations, memories) can be individually re-used within newly configured processors. The Base IP Blocks can be augmented with application-specific units. To describe processors in terms of blocks instantiated and interconnected within the constraints of the template, the TIM language is used. The Processor Template and Template Building Blocks discussed in Chapter 6 are the language elements of TIM. The TIM compiler compiles the textual processor descriptions into executable processor instances.

Processor Template (also referred to as a Cell) defines a processor. A Cell is composed of two parts: Core and CoreIO which form together a VLIW machine capable of executing parallel software with a single thread of control. The Core performs computations under software program control. It consists of a VLIW data path and a sequencer controlling the data path under software control. The data path contains a number of function units organized in a number of parallel issue slots, which are connected to registers organized in a number of register files via programmable interconnect. The function units perform computation operations on intermediate data stored in the register files. On each issue slot an operation can be started in every clock cycle. The CoreIO provides a memory and I/O subsystem allowing the core to be easily integrated in any system. The system in which a cell is integrated has access to the devices in CoreIO via slave interfaces. System access to the program memory, as well as, to the status and control registers of the cell is provided through a slave interfaces as well. These interfaces support a commonly
used standard protocol to provide easy integration of a processor in a wide variety of system architectures.

From the above brief description of the HiveLogic IP blocks it should be clear that ASIP IP cores considered in the ASAM project are configurable, but also extensible. Thus, ASIP Customization involves both ASIP instantiation and ASIP extension.

During the ASIP Customization, the processor instantiation decisions can be made regarding the following ASIP aspects:

1. number of issue slots
2. functional units (number, type, distribution)
3. register files (number, sizes, ports)
4. interconnect within issue slots and between issue slots
5. instruction set
6. application-specific instruction set
7. local memories (number, connection, ports)
8. interfaces: number, type, protocol.

For more details please see Section 12 in the confidential appendix.

ASIP IP cores considered in the ASAM project are configurable, but also extensible. This last feature means that custom operations can be added to basic processors together with the corresponding hardware implementing them, as well as, any other hardware extensions can be added to increase the processor effectiveness or efficiency. Also, the heterogeneous SoC system may contain hardwired accelerators, which are specified such that the flow will be able to match their functionality with parts of the application code. All these ASIP or SoC extensions are related to the ASIP-architecture and hardware design, and therefore, they will be constructed and selected by an autonomous sub-task in the scope of the ASIP-architecture customization task. In Figure 21 on page 54 this autonomous sub-task of the ASIP customization is represented as Extension Builder.

During the construction of the promising extensions and selection of the best of them, the proposed extensions have to be analyzed and evaluated in separation, before deciding to include them into the ASIP IP cores or SoCs, and before actually including them. The ASAM prototyping infrastructure (see Deliverable 4.1) has to enable the separate analysis and evaluation of the hardware extensions proposed. Fortunately, Genesys provides among others the capability of generating the HDL description of single function units implementing custom instructions. This capability will be used inside the prototyping infrastructure to evaluate the possibility of adding candidate custom instructions and related hardware to a particular ASIP. Using Genesys an HDL module will be generated for hardware of each candidate custom instruction, and used as input for the technology aware characterization. In a similar way all other ASIP or SoC hardware extensions will be analyzed and evaluated, i.e. their corresponding HDL modules will be generated and characterized subsequently.
After making all the proposed ASIP instantiation and extension decisions, and this way producing a promising customization alternative for a given (ASIP, partial C code) pair, the proposed customized instance of the ASIP has to be expressed in TIM. The TIM description can be compiled by the TIM compiler and simulated. Moreover, from the TIM description, a corresponding HDL ASIP instance description can be generated using Genesys and used together with the corresponding partial C code instance to emulation in the ASAM FPGA-based prototyping environment that will compute the related parametric characteristics and feeds them back (short feedback loop) to the ASIP Customization and Implementation process. Using the parametric characteristics, the ASIP Customization and Implementation process checks if the required constraints are fulfilled and checks to what degree the objectives are satisfied. If the result of the ASIP Customization process is not satisfactory, the (ASIP, partial C code) pair is further refined to more efficiently perform the mapped C sub-program, and the short feedback loop is used to get information on the characteristics of the refined processor. The ASIP Customization process of a given ASIP stops, if its result is satisfactory or it turns out to be impossible to fulfil the hard constraints or optimize the objectives to a satisfactory degree with the available resources. The whole ASIP Customization process stops, if it stops for all the (ASIP, partial C code) pairs transferred by the system-level DSE process to the ASIP Customization process. At this moment, the ASIP Customization process feedbacks a report on the customization results to the system-level DSE process.
10 The prototyping/implementation infrastructure

The ASAM tool-flow will include a technology-dependent hardware generation/prototyping/implementation/optimization infrastructure. This is the main object of WP4.

The main aim of such framework will be to provide support for all the levels of the architectural analysis, reducing the gap between the estimation of the performances considered during the early steps of the design flow and those really measurable after the implementation.

Firstly, the framework will be capable of creating an FPGA implementation of a candidate system configuration. The target application will be executed on it for emulation, in order to obtain switching activity figures and performance metrics to be considered during the ASIP- and System-architectural optimization processes.

Secondly, the infrastructure will be capable of creating all the necessary input for a prospective VLSI back-end implementation process of the candidate architecture.

Figure 21: Infrastructure general organization

Figure 21 depicts the general overview of the infrastructure. It is composed by several sub-tools that interact with each other and with commercial tools. The inputs for the framework are placed at the top of the flow-chart. All the inputs are needed...
to describe the system configuration under prototyping.

In particular, as envisioned by the general approach of the ASAM project, the description of the system is provided by means of a System-architectural specification (involving the system-level description), together with a set of ASIP-architecture specifications (involving the module-level description of the configurable modules to be instantiated in the system).

The **HDL/script generation** sub-flow will be in charge of converting the system- and module-level specifications into the inputs needed for the implementation of the candidate architectural configuration. This tool set will produce all the HDL code to be synthesized and all the scripts needed to run the implementation flow on commercial tools. Involved tools will be:

- A System-Level Builder, that will be capable of analyzing the system-level specification describing the System-architecture under prototyping/implementation and generating accordingly an HDL description of the upper hierarchy-level of the system.

- An ASIP HDL builder, a tool set that is part of the HiveLogic suite by Silicon Hive, that will allow the creation of the RTL description of a customized processor according to a specification of the ASIP-architecture features of the customized processor to be produced, expressed using the TIM language.

- The SHMPI builder, in charge of generating the complete HDL Network-on-Chip topology, creating, connecting and configuring a set of network component instances. The produced topology will be custom-tailored to the NoC ASIP-architecture specification file provided in input to the framework.

On one hand, the FPGA-tailored HDL code and the scripts produced by the previously mentioned tools will be taken as input by commercial tools, to perform all the different implementation steps needed to produce the a programming file for the target FPGA device.

On the other hand, the VLSI-tailored HDL code and the scripts produced by the previously mentioned tools will be taken as input by commercial tools to generate a pre-tape-out description of the system, suitable to for actual production or for characterization at the highest precision level.

The actual prototyping will consist in the **execution of the target application** on the complete system implemented on the target FPGA platform, or in a partial evaluation, such as the execution of a part of the application code on a given ASIP implemented on the FPGA platform, or the test of the hardware of a proposed custom extension. The application binaries, provided as input to the prototyping infrastructure by the software compilation tool chain will be loaded in the program memory of the processors included in the system.

Dedicated **hardware probes/counters** (mainly cycle counters to estimate latencies and event counters needed to estimate the activity of the modules) related with the metrics to evaluate, will be connected to the system signals, and will be made accessible by the processors in the system as memory mapped resources.
The performance numbers and the activity traces obtained in this way will be interpreted using adequate area/frequency/energy models, to obtain early power, energy, timing and area figures related to a prospective ASIC implementation, without the need to perform long-lasting cycle-accurate simulations.

One last feature of the framework will be related to the extension of the ASIPs: custom operations can be added to basic processors together with the corresponding hardware implementing them and hardwired accelerators can be added to heterogeneous SoC systems.

All the extensions will be constructed and evaluated by an autonomous sub-flow, represented in the figure as Extension Builder. Such flow will allow the prototyping infrastructure to enable the separate analysis and evaluation of the hardware extensions proposed.

Detailed information and descriptions of the prototyping infrastructure are reported in Deliverable 4.1, extensively discussing its general organization and interfaces.

### 10.1 Evaluation results

The main metrics that will be produced in output by the prototyping infrastructure, to be considered as objectives for the optimization of the system will be:

- Area occupation
- Power/Energy consumption
- Execution Time

Beside the objective functions to be optimized, several other metrics will be collected at each prototyping step, to drive the optimization algorithms in iteratively refining the architectural configuration.

Here follows a preliminary list, to be validated or extended/reduced during the project according to the results of the future research activities.

- Metrics related with ASIP ASIP-architecture:
  - Number of accesses to every issue slot
  - Number of accesses to every functional unit inside the issue slots
  - Number of accesses to every register file
  - Number of accesses to every internal bus from load-store-unit to memory
  - Number of accesses to every branch of the register selection network module
  - Number of stall cycles
Public with confidential appendices

- Metrics related with the interconnect structure
  - Number of flits sent by every NI and by every switch port
  - Number of nacks at each switch port due to congestion
  - Number of idle cycles
11 References

[1] ASAM Technical Annex


12 Glossary and Terminology

ACN  Argument Connection Network: the HiveLogic IP module instantiated within the processor’s datapath to provide a sparsely connected communications facility running from Register file outputs to Issue slot inputs.

AHB  AMBA High-performance Bus: bus fabric and interface introduced by ARM Ltd.

AMBA  Advanced Microcontroller Bus Architecture: bus fabric and interface introduced by ARM Ltd.

ANSI-C  American National Standards Institute version of the ‘C’ programming language.

API  Application Programmers Interface

ARM  ARM Ltd.: semiconductor intellectual property provider

ARU  Arithmetic Unit

ASIP  Application-specific Instruction-set Processor

AXI  Advanced eXtensible Interface: third generation of ARM Ltd’s bus interface specification

C++  Programming language, based on the ‘C’ language. SystemC is built on top of C++.

CIO  Silicon Hive’s proprietary MMIO interface protocol

CoreBrowser  HiveCC tool providing graphical view of HiveLogic, HiveFlex, and HiveGo processors

CoreIO  Input/output and memory module of HiveLogic processor

DDR  Double DataRate SDRAM

DLP  Data-Level Parallelism: refers to kind of processing where a single operation on a processor handles vectors of processors simultaneously

DMA  Direct Memory Access: In this document, DMA refers to a particular hardwired system block which can autonomously transfer (blocks of) data.

DSP  Digital Signal Processor: may either refer to the kind of software specifically targeted at processing digital signals, or to the type of processors specifically meant to run that software

EDA  Electronic Design Automation

ELF  Executable and Linkable Format for object files and binaries generated by HiveCC

ESL  Electronic System Level: term to qualify tools which deal with SoC-level issues

eVC  e Verification Component: referring to verification IP expressed in theCadence ‘e’ language

FU  Function Unit

GeneSys  HiveLogic tool for instantiating and interconnecting IP blocks (either HiveLogic or customer-supplied)

HDL  Hardware Description Language

HiveDBHiveCC debugger tool

HiveCC  Silicon Hive’s Software Development Kit, including ANSI-C compiler

HiveIDE  HiveCC Integrated Development Environment

HiveLogic  Silicon Hive’s configurable parallel processing flow

HiveRTHRT: Hive Run-Time, application programmers interface for driving and communicating with HiveLogic processors

HRT  HiveRT

HSD  Hive System Description: language for describing multi-core systems

IDE  Integrated Development Environment

ILP  Instruction-Level Parallelism: refers to the kind of instruction processing where a single instruction contains multiple operations. Also refers to the measure of the average number of operations executed in parallel on a VLIW machine, throughout (part of) and application.

IS  Issue Slot

ISP  Image Signal Processor: referring to the HiveFlex 2xxx family of processors

JTAG  Joint Test Action Group; typically describing a serial interface standard allowing test access to SoCs.

Kgate  Kilo-gate, measure of chip logic complexity and area

LSU  Load/Store Unit: Function unit specifically meant for exchanging data between VLIW datapath and memory
MAC  Multiply-Accumulate: the combined operation of multiplying and adding, also refers to the multiply-accumulate function unit within a processor

MIMD  Multiple Instruction Multiple Data: referring to multi-processors which can execute multiple independent parallel operation streams on multiple independent data streams

MMIO  Memory-mapped Input/Output

NCSim  Cadence Incisive toolsuite, meant for verification of ASICs, in particular containing the ncsim unified simulation engine for Verilog, VHDL, and SystemC

OCP  Open Core Protocol: bus interface introduced by the OCP-IP Association

OLP  Operation-Level Parallelism: single operations performing multiple tasks simultaneously which, on a RISC processor, would have taken multiple operations

PC  Program Counter

RPC  Remote Procedure Call

RISC  Reduced Instruction-Set Computer

RF  Register File

RSN  Result Select Network: the HiveLogic IP module instantiated within the processor’s datapath to provide a sparsely connected communications facility running from Issue slot outputs to Register file inputs

RTL  Register Transfer Level

SAD  Sum of Absolute Differences

SoC  System-on-Chip

SR  Status Register

SRAM  Static Random Access Memory

SW  Software

System Top-level collection of hardware components in HSD, often roughly corresponding to the functionality of an SoC. This specifically does not refer to application-level systems consisting of constellations of different processing kernels, such as often associated with MatLab descriptions and Kahn process networks, but rather the hardware on which such systems could be mapped.

Sub-system  Collection of hardware components in HSD, corresponding to some intermediary hierarchy level

SystemC  A set of C++ classes and macros which provide an event-driven simulation kernel

Tcl  Tool Command Language: scripting language used by many EDA tools to automate processes using those EDA tools

TIM  Silicon Hive’s proprietary processor description language

UART  Universal Asynchronous Receiver Transmitter

VCS  Synopsys’ multicore-enabled functional verification solution (HDL simulator)

Verilog  Hardware description language, used to model electronic systems

VHDL  VHSIC Hardware Description Language

VHSIC  Very High-Speed Integrated Circuit

VLIW  Very Long Instruction Word: generally used to refer to processors which can execute multiple independent operations in parallel.

VLSI  Very Large Scale Integration

VSP  Video Signal Processor: referring to the HiveFlex VSP2xxx family of processors

VSS  Video Sub System: referring to HiveGo VSS3xxx SoC sub-systems

SDK  Software Development Kit

TLP  Thread-Level Parallelism: In Silicon Hive terminology, this refers to the kind of processing where multiple processors within one SoC operate simultaneously.

X86  Abbreviation of the line of Intel processors (and associated instruction-set architectures) which started with the 8086