UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTE OF INFORMATICS
COMPUTER ENGINEERING

FELIPE AUGUSTO CHIES

Validation and Evaluation of the ASAM - Automatic Architecture Synthesis and Application Mapping - Flow

Final Report presented in partial fulfillment
of the requirements for the degree of
Computer Engineer.

Prof. Dr. Luigi Carro
Universidade Federal do Rio Grande do Sul
Advisor

Menno Lindwer
Intel Corporation
Coadvisor

Porto Alegre, June 2013

CIP – CATALOGING-IN-PUBLICATION

Chies, Felipe Augusto

Validation and Evaluation of the ASAM - Automatic Architecture Synthesis and Application Mapping - Flow / Felipe Augusto Chies. – Porto Alegre: COMGRAD ECP UFRGS, 2013.

40 f.: il.

Final Report (Master) – Universidade Federal do Rio Grande do Sul. Computer Engineering, Porto Alegre, BR–Brazil, 2013.
Universidade Federal do Rio Grande do Sul
Advisor: Luigi Carro; Intel Corporation
Coadvisor: Menno Lindwer.

1. Embedded systems. 2. Heterogeneous multi-processor system-on-chip (MPSoC). 3. Customizable ASIPs. 4. Architecture synthesis. 5. Design space exploration. I. Carro, Luigi. II. Lindwer, Menno. III. Título.

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programmers Interface
ARU	Arithmetic Unit
ASAM	Automatic Architecture Synthesis and Application Mapping. EU-funded project with the high-level goal of inferring suitable architectures and map software onto these, given unoptimized target-unaware input applications.
ASIP	Application-specific Instruction-set Processor
Bus	A TDM interconnect with one channel.
DSE	Design space exploration
EDA	Electronic Design Automation
FIFO	First-In First-Out. A queue.
FU	Function Unit
HiveCC	Intel-VIED's SDK for Intel-VIED-specific ASIPs.
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.
IPC	Inter-process communication
IS	Issue Slot
MAC	Multipy-Accumulate: the combined operation of multiplying and adding, also refers to the multiply-accumulate function unit within a processor.
MCG	Mobile and Communications Group, within Intel.
MIMD	Multiple Instruction Multiple Data: referring to multi-processors which can execute multiple independent parallel operation streams on multiple independent data streams
MoC	Model of computation
OLP	Operation-Level Parallelism: single operations performing multiple tasks simultaneously which, on a RISC processor, would have taken multiple operations

PC	Program Counter
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
SDK	Software development kit. Set of tools to support developing software.
SIMD	Single-Instruction Multiple-Data.
SoC	System-on-Chip
SR	Status Register
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
TIM	Intel-VIED's proprietary processor description language.
VIED	Video and Imaging Engineering Department within Intel-MCG.
VLIW	Very Long Instruction Word: generally used to refer to processors which can execute multiple independent operations in parallel.
VLSI	Very Large Scale Integration

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

High-quality MPSoCs can only be constructed exploiting more adequate concepts of computation, storage and communication, as well as usage of efficient design methods and electronic design automation (EDA) tools. This document discusses some subsystems of the Automatic Architecture Synthesis and Application Mapping (ASAM) flow. In addition, we propose a component based design for multi-ASIP platform and we will introduce and compare several strategies to connect different issue slots. Experimental results show that our approach give the best trade-off regarding application execution time and energy consumption between the different architectures analyzed.

Keywords: Embedded systems, heterogeneous multi-processor system-on-chip (MP-SoC), customizable ASIPs, architecture synthesis, design space exploration.

Validação e Avaliaçao de uma ferramenta automática de síntese de arquitetura e mapeamento de aplicação (ASAM flow).

RESUMO

A fim de construir MPSoCs de alta qualidade é necessário investigar e explorar conceitos mais satisfatórios de computação, armazenamento e comunicação, além de utilizar eficientes métodos de design e softwares de projeto de circuitos integrados. Este documento discute alguns subsistemas de uma ferramenta automática de síntese de arquitetura e mapeamento de aplicação (ASAM flow). Além disso, será proposta uma plataforma baseada em componentes para o design de sistemas com múltiplos ASIPs e algumas estratégias para conectar diferentes issue slots serão introduzidas e comparadas. Os resultados experimentais mostram que o método introduzido neste documento resulta nos melhores trade-offs com relação ao tempo de execução e ao consumo da aplicação entre as diferentes arquiteturas analisadas.

Palavras-chave: sistemas embarcados, sistema-em-um-chip com multi-processadores heterogêneos (MPSoC), ASIPs customizáveis, síntese de arquitetura, exploração espacial de designs.

1 INTRODUCTION

Modern embedded systems design technology is relevant for different fields of application (multimedia, telecom, medical, military, etc.). These applications require ultra high performance, but also flexibility and low Non Recurring Engineering (NRE) costs in order to be adjustable to shifts in the market.

The progress in semiconductor technologies has allowed Multiprocessor System on Chip (MPSoC) to reach the high performance needed by this kind of application. However, high-quality MPSoCs can only be constructed exploiting more adequate concepts of computation, storage and communication, as well as usage of efficient design methods and electronic design automation (EDA) tools.

The high performance is often achieved through highly specialized processors. Application specific instruction-set processors (ASIPs) are programmable, deliver high performance and they are energy efficient. Recent system-on-chip solutions ((Movidius, 2013), (Intel, 2013)) contain such components. We can improve the efficiency of the system by exploring the intrinsic parallelism of an application for instance (by effectively exploiting the target VLIW ASIP).

The focus of this document is to evaluate and validate the ASAM flow, in terms of e.g. lead time, amount of manual work, and quality of results. During the process of validation, some issues were found in the flow and the solutions implemented will be presented. In addition, a component based design for multi-ASIP platform was developed and will be introduced in this document.

This document extends the previous work that can be found in the Appendix G. This Appendix will be referred many times in this document as the project description. All the research and results presented here are related to a European project ASAM (Automatic Architecture Synthesis and Application Mapping) being currently executed in the framework of the ARTEMIS program. By the time that this document was written the project was still in progress. Thus, things could have changed and some issues introduced here could have been fixed.

This document is organized as follows. The next section introduces some related works. Section 3 discusses the ASAM flow, the integration challenges and issues found and the solutions proposed. Section 4 focuses in the component based design proposed and its relevance when generating multi-ASIP systems. Section 5 discusses the experimental results and Section 6 concludes this document and lists some perspectives.

2 RELATED WORKS

Many problems related to automatic ASIP design represent hot research topics. Retargetable compilers, such as Coware Processor Designer, Expression, Mescal, ASIP-Meister, Tensilica's compiler or HiveCC, are used to schedule and map a high-level application specification onto the optimized configurable ASIP platform. However, the major general challenge is the hardware and software co-design tuned for a specific application.

Many published research results (JOZWIAK, 2001), (JOZWIAK; ONG, 2008), (DENSMORE; PASSERONE, 2006) and system design frameworks, e.g. Metropolis (Metropolis, 2013), Daedalus etc., target heterogeneous MPSoC design. Some achievements in design automation have demonstrated that it is possible to automatically generate application-specific architectures that compete with handmade solutions. Examples of these successes are High-Level Synthesis (SCHREIBER; ADITYA; MAHLKE; KATHAIL; RAU; CRONQUIST; SIVARAMAN, 2002) for ASIC design and automatic Instruction Set Extension (MURRAY; FRANKE, 2012) for ASIP configuration.

Several industrial tool-flows, e.g. (FlexASP project, 2013) and (KATHAIL; ADITYA; SCHREIBER; RAU; CRONQUIST; SIVARAMAN, 2002), are available to specify, simulate, and/or synthesize VLIW architectures. (KATHAIL; ADITYA; SCHREIBER; RAU; CRONQUIST; SIVARAMAN, 2002) can perform an automatic exploration of these architectures, but it's focused on VLIW processors with a centralized register file. Terechko et al. (TERECHKO; THENAFF; GARG; EIJNDHOVEN; CORPORAAL, 2003) defined a taxonomy for different variations of clustered VLIW processor architectures. However, until now, clustered processor architecture exploration has been performed by hand.

Beside ASAM, no flows currently exist which automate both (multi-core) system design and design of the multiple ASIPs within those systems. The existing multi-core DSE flows (such as Metropolis and Daedalus) assume a given library of processors. Moreover, in existing flows, the types of processors are not as heterogeneous as those built by ASAM. The existing flows merely differentiate between a multitude of standard RISC cores and standard DSPs. The task of specifically having to restructure the code partitions, in order to target the differences between these cores is typically also not handled by other flows.

3 PROPOSED CHANGES IN THE ASAM FLOW

3.1 Overview

The main objective of the ASAM project is to develop a design methodology and a design flow that provides efficient application mapping and automatic architecture synthesis for multi-ASIP systems. Figure 3.1 gives an overview of the project's global objective. Briefly, given an input application (sequential C Code), the flow should propose a suitable target platform. In addition, the application's tasks should be partitioned and optimizations in communication, memory management, loop transformations and vectorization should be applied. This flow will be implemented by extending the traditional Y-Chart Work Flow (Figure 3.2). A more detailed explanation about the project can be found in this project description (Appendix G, Section 2) and in the literature ((JOZWIAK; LINDWER; CORVINO; MELONI; MICCONI; MADSEN; DIKEN; GANGADHARAN; JORDANS; POMATA; POP; TUVERI; RAFFO, 2012), (JOZWIAK; LINDWER, 2011), (JOZWIAK; LINDWER, 2010), (ASAM, 2013), (JOZWIAK; LINDWER; MADSEN, 2011)).

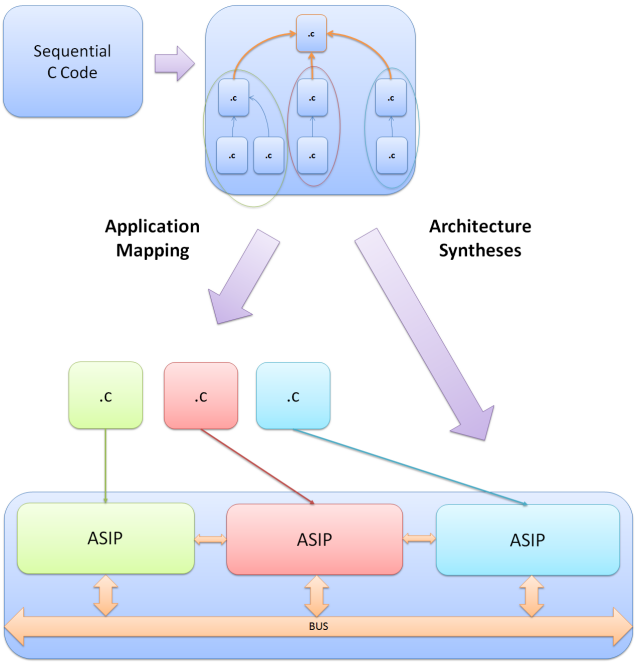
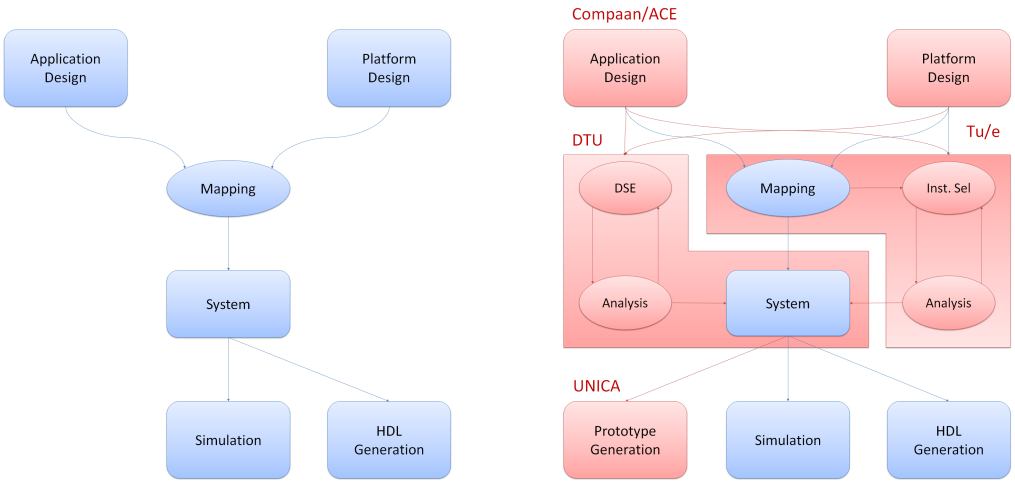


Figure 3.1: Global objective of the ASAM Project.



(a) Traditional Y-Chart Work Flow (b) Y-Chart Extention proposed

Figure 3.2: Work Flow's extension proposed by the ASAM Project.

3.2 Integration Challenges

The initial goal of this thesis project is to evaluate the ASAM Flow. However, in order to do it, the tools should be working properly and all the subsystems of the flow should be connected or, at least, they should have a standard of communication already defined in order to simulate and generate all the inputs/outputs correctly. Unfortunately, that was not the case. When this work started, most part of the tools were not delivered and the partners of the project had never tried to communicate the subsystems. In addition, the use cases chosen were not ported to be used on the flow and a lot of extra work was required.

One of the most important challenges of this kind of project is that its success heavily depends on the compatibility and coherent collaboration of all models and tools. Due to its complexity, the scope of this project covers many different research fields. The synthesis and prototyping flow involves several different kinds of models, many various tools operating on them and complex collaboration among models and tools (JOZWIAK; LINDWER; MADSEN, 2011). Keeping all this information in mind, it becomes evident that a significant effort should be made in order to integrate all the tools that compose the flow.

Another challenge usually faced in projects like this is the interaction between the different partners. Taking the ASAM project as an example, we have eight different partners (four universities and four companies) and most of them are spread around the world. This makes project meetings something difficult to happen, which can lead to a small interaction between the partners. Good project managers and leaders become very important in this case to ensure that the partners keep exchanging information and to make sure that the subsystems that need to integrate are communicating well. Wrong assumptions about what one subsystem can supply to the other can have seriously consequences on the whole project, sometimes even leading to the failure or restructuration of a whole subsystem.

Different priorities, ways of working and objectives are other points to be faced during the progress of the project, mainly between academia and companies. Universities partners for example are normally focused only in the research/novelty part of the problems. It can let the "non-research" part of the work (default libraries, use cases, global integration, templates, general documentation, interfaces, translators, among others) forgotten

without implementation, creating therefore gaps in the communication and avoiding the integration of the whole system. In general, these tasks are not implemented because they are seemed as a "waste of valuable time". However, knowing that these implementations could allow a whole automated flow to work, it can only be seen as beneficial in a long-term project. On the other hand, universities tend to be more flexible than companies, being more adaptable to little changes in the initial specification of the project.

Lastly, it is important to highlight that the integration process should start as soon as possible, even though the tools are not finished. Only by integrating we can find out what is missing in the flow and what was misunderstood between the different partners. A lot of time can be saved by doing it in earlier phases of the project.

3.3 Initial Integration and Evaluation

As mentioned before, some challenges were faced during the integration. In order to evaluate and integrate the flow, a work strategy was created and followed during the execution of this project. Basically, use cases and some extra examples were used with the intention of stressing the flow to find out gaps and communication problems. This strategy is illustrated on Figure 3.3.

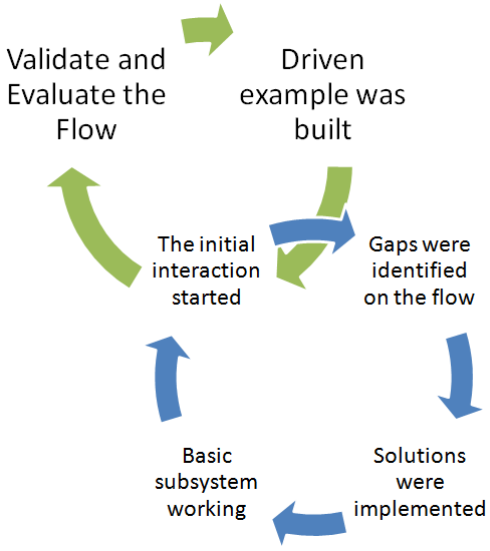


Figure 3.3: Meta-flow used to evaluate and integrate the ASAM Flow.

3.3.1 The Model of Computation Issue

The first task performed during this thesis project was to map a consumer-oriented application (in this case a Motion JPEG encoder) into a multi-ASIP system created specially and optimized for this application. Most of steps required to produce and map this system were performed by hand in order to generate a new reference to compare against the results generated automatically by the flow. This example is different from the initial use cases because we kept a track for each transformation applied on the code and on the system. Thus, we are able to analyze the result of each tool separately and not only the result of whole flow. An exhaustive description of these steps can be found on the project description (Appendix G, Section 4).

With the MJPEG example created and ready to be analyzed, we tried to run the automated ASAM flow with this example as input. By doing it, we noticed that the initial C Code of the input application needs to be changed. The reason of this requirement is that the flow needs an initial Model of Computation (MoC) to start the Design Space Exploration (DSE). The MoC chosen by the project is a Kahn Process Network (KPN). Due to its expressiveness, the KPN is hard to be inferred and it's difficult to generate an efficient implementation without some pragmas/hints embedded on the C Code. The tool used to generate the MoC is the Compaan Compiler and an example illustrating the modifications needed by the tool can be found on Appendix G, Section 4.1.2.

The ASAM project has three main use cases: an Electrocardiogram algorithm (ECG); a Digital Hearing Aid System; a MPEG4 Encoder Algorithm (further information on Appendix G, Section 3.1). These use cases were supplied by the partners of the project and they are coded in C. However, they were not compatible with the Compaan Compiler, generating thereby the first issue to be faced in order to evaluate the flow.

The Hearing Aid application was the first target use case to be ported. The initial code was relative small (about 750 lines of code), but much more effort than expected was required to adapt the application. Almost five working days were spent porting this use case and the final code was more than two times bigger than the original code (2000 lines of code). The final KNP generated by the tool can be found on Figure 3.4.

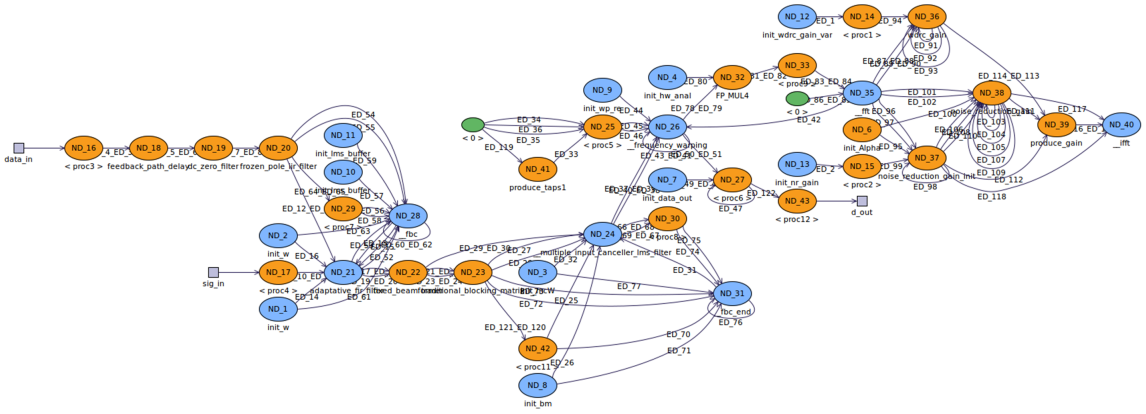


Figure 3.4: KPN generated analyzing the Hearing Aid use case.

A similar matter was observed with the MJPEG encoder. A simple coarse-grained KNP (only six nodes were generated from the six main functions of the application - Appendix G, Section 4.1.2) can be easily generated without many modifications on the code. However, the DSE of the flow needs a more fine-grained Model of Computation (almost a node for each loop on the code) in order to perform important analysis of the code and to infer significant code transformations. For the MJPEG encoder, a hierarchical and fine-grained KPN was generated after almost four working days. It is important to notice that these are quite small applications. More complex applications as the MPEG4 encoder would need much longer time in order to analyze the code and after to port it. For this reason, it was decided to use the MJPEG example as an alternative of the MPEG4.

In the opinion of the author of this document, the trade-off brought before concerning modifications in the input application is one of most important open discussion of this project. It needs to be carefully studied to decide how much effort should be allocated to change the code in order to generate a certain level of granularity in the Model of

Computation. This granularity will have a significant impact on the quality of the final result.

3.3.2 A Unified Communication Method

With the use cases ported and ready to run on the ASAM Flow a new evaluation attempt started. This time, many problems concerning the communication between different subsystems prevented the flow to work. In order to repair it, a large interaction with the different partners of the project started. After numerous meetings, a standard of communication was defined to be used for all the subsystems. It was decided to use three different XML files. The first one should specify the user constrains, the second one should be used to describe the communication between the different processors and memories on the systems. The third one is the most important and it will be used by almost all tools. It describes all the analysis performed in the application, simulations results regarding power, area and performance and the decisions taken concerning the mapping and the system generation. Examples of these three files can be found on Appendix A, B and C respectively. One important aspect of these files is that they keep track of the current status of the flow. Making use of this information, a designer controlling the flow could change some parameters of the tools helping the flow to converge to a more optimized solution.

In addition, as the different partners/tools started to interact more, concerns in the initial description of the flow were found and some modifications were needed. Figure 3.5 illustrates the updated version of the ASAM Flow. In comparison with the initial flow (introduced on the project description) we can have a much clearer idea about the different tools inside the flow and how they interact.

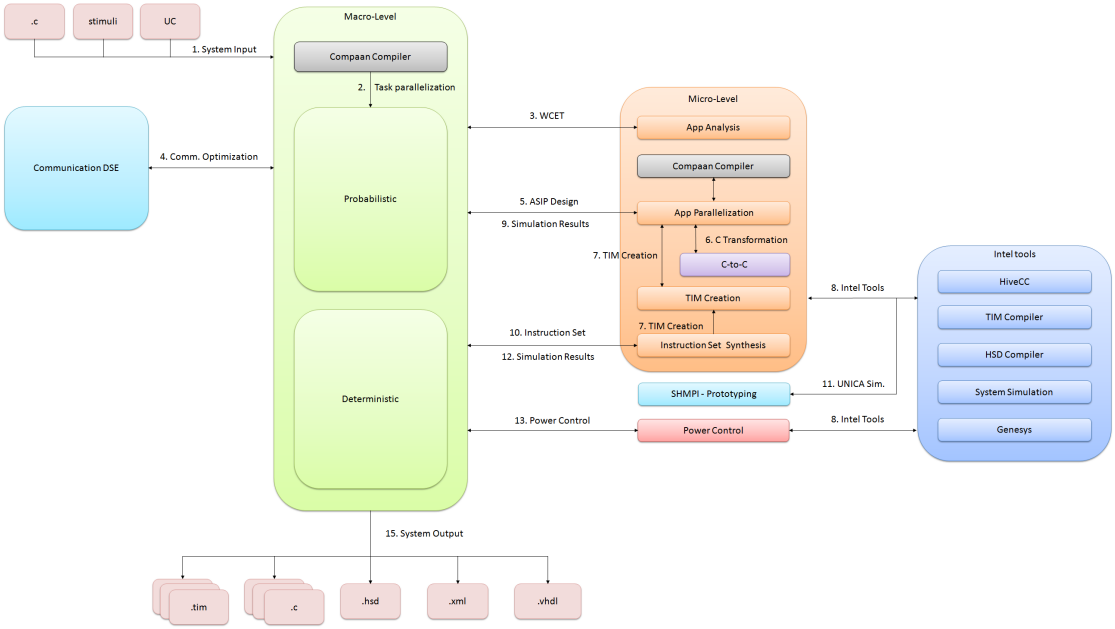


Figure 3.5: The ASAM Flow.

3.3.3 Selecting and Porting Extra Benchmarks

With the flow redefined and with the communication problem solved, we could start a new iteration of the working meta-flow using the Hearing Aid and the MJPEG applica-

tion. As it's generally expected, this first try of the flow failed and more issues appeared. Even though these use cases are not so complex, they have some complexity that can be explored in different ways by the DSE and it was quite hard to identify which part of the flow had some problem or did not perform well. Hence, some basic examples were needed in order to easily validate individual tools.

A solution proposed to this matter was the Polybench set of benchmarks (Polybench, 2013). In total, 25 benchmarks were ported to be used on the ASAM flow. These benchmarks were chosen because they are very small, they contain static control parts (that help the generation of the MoC, making it easy to port to the ASAM Flow) and they are largely used in Academia (providing us many results to compare against). Two more benchmarks were ported: a Low Pass Spatial Filter (LPSF) and a Down Sampling Filter.

By stressing these benchmarks on the flow problems were found and places with some gaps were easily identified. The solutions proposed can be found in the following section.

4 COMPONENT BASED DESIGN FOR MULTI-ASIP PLATFORM

4.1 Background Concepts

4.1.1 Architecture Template

The project targets clustered VLIW ASIP machines capable of executing parallel software with a single thread of control. Figure 4.1 shows a simplified view of the architecture template.

The VLIW datapath is controlled by a sequencer that executes the program stored in the local program memory. The datapath contain Functional Units (FUs) grouped into Issue Slots (IS) that are connected via a programmable interconnect network to Register Files (RF). The sequencer is a special IS containing branch instructions, a program memory, a status register (SR), a program counter (PC) and some registers. FIFOs can be added in order to manage external communication. They can be connected to any IS, but they are usually connected to the sequencer.

FUs within the same IS share hardware, in consequence they cannot be used at the same time. In addition, FU implements operations that can require pipelining. Thus, each IS can start at most one new operation per cycle.

Almost every component of within the processor architecture template (PAT) is freely scalable. It is possible to have as many RFs, ISs, FUs, memories and master interfaces as wanted. All these components can work both with scalar and vector elements of different sizes. Therefore, both single-instruction multiple-data (SIMD) and multiple-instruction multiple-data (MIMD) can be performed. The interconnection between all components is also customizable.

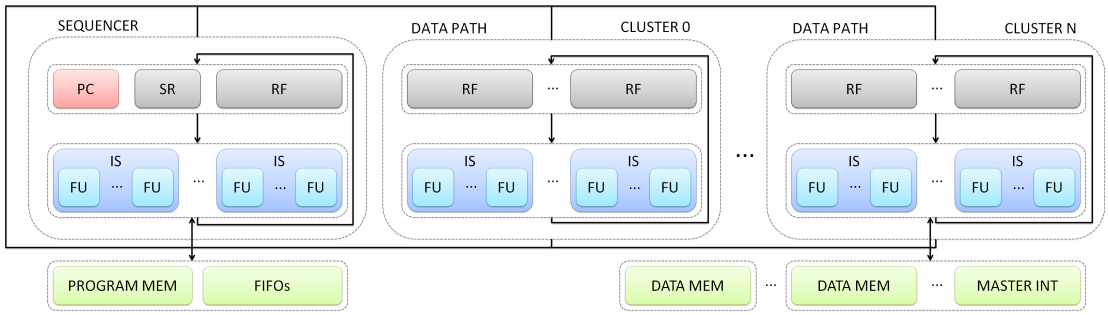


Figure 4.1: The Processor Architecture Template (PAT).

4.1.2 Application Model

In the Micro-Level DSE of the ASAM flow, the application is represented by a task-graph model. This task-graph is an Array Oriented Language (Array-OL) (GLITIA; DUMOND; BOULET, 2010). This model contains two abstraction levels: a higher level that specifies inter-task dependencies and a lower level that specifies intra-task (data-) dependencies.

4.1.3 Architecture Construction

The initial prototype of the ASIP's architecture is derived from an exploration of the application model which intent to find the best combination of several possible loop-transformations (e.g. loop fusion, tiling, and vectorization). In this way, we adapt the hardware and the software architectures to the requirements of the application with satisfactory results.

4.2 Configurable Library of Components

As explained before, the architecture construction is obtained from the application model. However, we can only infer the number and the kind of issue slots and memories that we need from the application model and the real architecture was not being generated. To fill this gap, a configurable library of components was created to work together with the DSE. This allowed to automatically generate the ASIP architecture. These components were developed in the TIM language VIED's proprietary processor description language.

Appendix D contains some examples of the components created to be part of a default library. They are configurable and the element size can be easily changed. By connecting these components together the ASIP can finally be created and simulated.

As the instruction-set architecture exploration is based on a shrinking method, these components have a default and extensive set of operations. This allows the exploration to investigate many different designs and to choose only the operations more relevant to the application being analyzed.

4.3 Inter Issue Slot Communication DSE

Another concern found inside the flow was the way that the different issue slots were being connected. Even though we could generate models of ASIPs and simulate their behavior in software, the implementation was not possible. It was happening because all issue slots were connected through a fully connected network. This is not feasible in practice due to the complexity of the routing process, the final area and power consumption of the circuit. In this section we will describe a solution proposed to reduce the size and the complexity of the interconnections between different issue slots.

The construction of the network starts from the hierarchical Array-OL model. Each task in the Array-OL description becomes a cluster in our model. The nested level of fusion identifies nested levels of clusters. It is important to observe that due to locality reasons, issue slots inside the same cluster tend to need a higher level of communication than issue slots of different clusters.

From the analysis of the data-flow graph, the maximum number of operations that can be executed in parallel is computed. This number defines the number of issue slots to be allocated and distributed inside the different clusters. Then, issue slots and memories are

evenly distributed in the clusters.

The issue slots within a cluster are fully connected and the clusters are connected to each other in a pipelined manner. Inside a cluster only one issue slot is chosen to take care of the external communication as illustrated in Figure 4.2. The input memory is connected to the sequencer and the output memory is connected to the latest issue slots on the pipeline chain. As the sequencer is connected to the input and to the default memory (where the stack is placed) the communication with this issue slot is usually higher. For this reason, we decided to connect all clusters in the highest hierarchical level directly to the sequencer. In this way, the maximum distance between these clusters become equal to two. This reduces the overall connectivity of the VLIW, although it maintains minimum connections as required by the application.

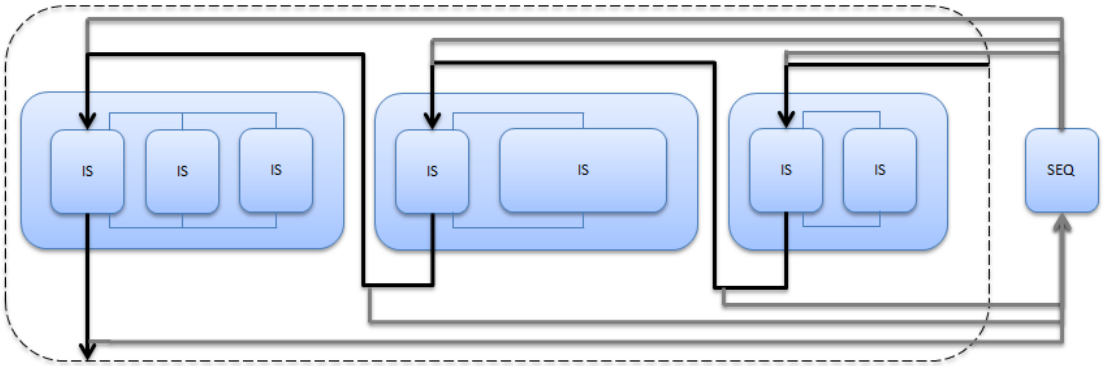


Figure 4.2: Example of clusters of issues slots interconnected.

4.4 Deriving the System Level Communication from a Task-Graph

The ASAM Project has already a tool to design and explore the system communication. However, during the period of this project, the Communication DSE was not integrated with the ASAM flow. A fast solution was needed in order to generate and simulate complete systems, allowing the partners to evaluate design choices.

We decided then to create a hierarchical bus-based network derived from a task-graph. Figure 4.3 shows this procedure. The task-graph is generated by the Macro-level system of the ASAM Flow. Each node of this graph is a cluster of different tasks that will be mapped on the same ASIP and each edge represents a channel of communication. In order to infer the connections we will generate new clusters. The clusters' generation works as follow. First, we select a consumer from the graph and a new cluster is created. Then, all nodes that are producers for this consumer will be included inside this cluster. This procedure will be repeated until each consumer of the graph has its own cluster. The next step is to generate a multi-layer bus for each cluster connecting all nodes inside the cluster. The number of layers in the bus is equal to $log_2(number\ of\ edges\ inside\ the\ cluster)$. As we want to have some flexibility in favor of possible future changes in the application, it is important to create a communication path between all ASIPs in the system. This is ensured by creating a last multi-layer bus that connects all clusters. The number of layers in this bus is equal to $log_2(number\ of\ cluster)$.

The model of communication proposed here can still be used in the future of the ASAM project. As this model showed to be efficient in the experiments performed, it can be used as an initial approach to generate first estimations.

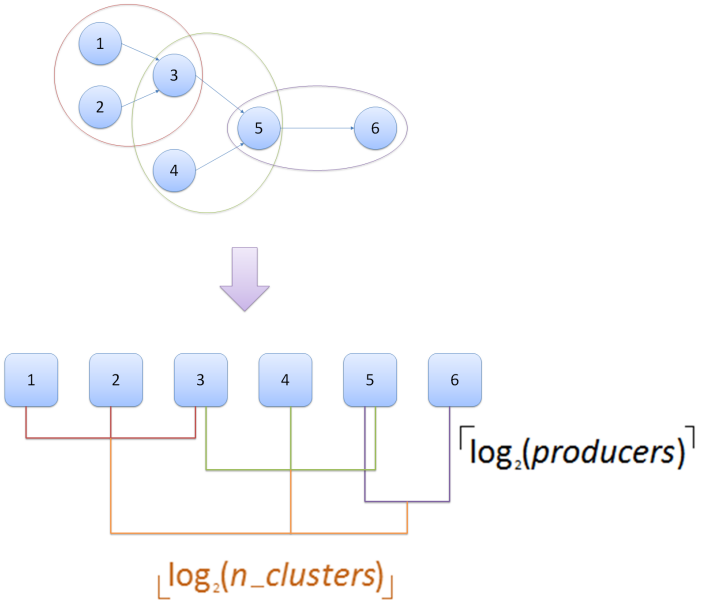


Figure 4.3: Deriving the system communication from a task-graph.

4.5 Generic API Definition for a User-Oriented Multi-ASIP System

The last gap found in the ASAM flow, but crucial for it success was that lack of some API to manage the system communication. As we aim to develop multi-ASIP systems, we need a system structure which ensures the synchronization between diverse tasks mapped on different ASIPs. In addition, it is important to have some kind of API to help to load and store data from the host processor to the system.

Based on a shared memory approach and making use of FIFOs to shake hands between the different processes, an API was developed to manage the system communication. The elements that compose this API are presented in the Figure 4.4. This API is generic, working both with scalar and vector elements. In addition, it was implemented in a way that more complex modes of operation like double-buffering are allowed.

Examples of the use of this API can be found in the Appendix E and F. The Appendix E represents the orange host square in the Figure 4.4 and it shows a host code interacting with an ASIP. It makes use of the functions *hrt_asam_indexed_store_signed_vector* and *hrt_asam_indexed_load_signed_vector* to load and store vectors in the local memory of the ASIP. To store scalar elements, the function *hrt_asam_mem_store_nooffset* is used.

The Appendix F is the green squares in the Figure 4.4. This is a default template that can be generated automatically from the analysis of the application. It is responsible for all the system communication.

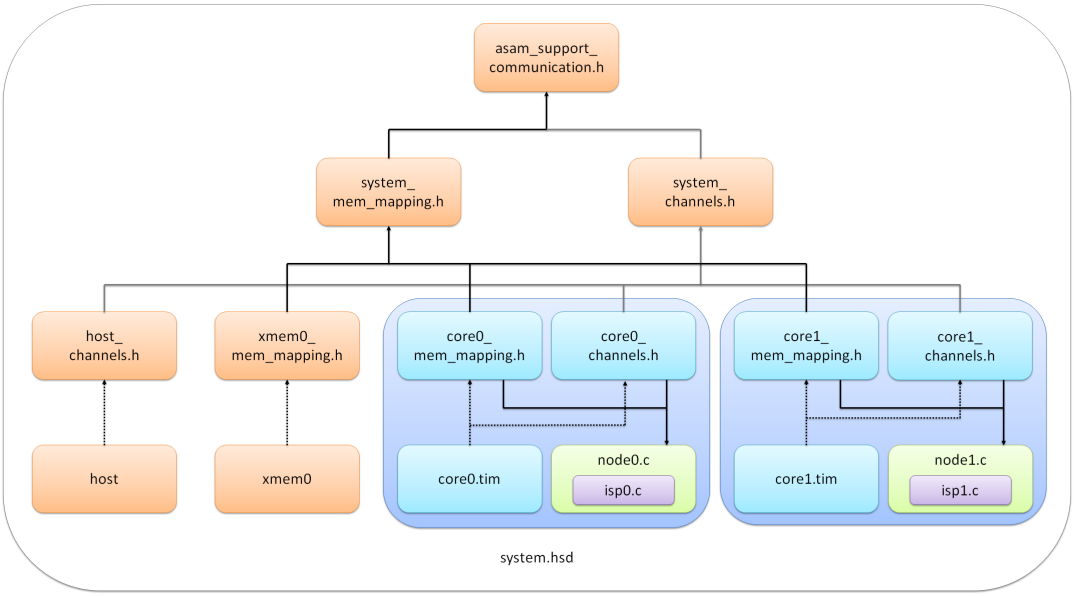


Figure 4.4: Template of the API proposed for a User-Oriented Multi-ASIP System.

5 EXPERIMENTAL RESULTS

To validate the methods briefly introduced before and test some subsystems of the ASAM flow we have selected seven of the benchmarks presented on Section 3.3.3. The applications chosen are listed in Table 5.

The only parts of the flow that we were able to test were the application analysis and the application parallelization of the Micro-Level in collaboration with Intel tools. Some steps were accomplished manually in order to supply the correct input information to these tools. However, these tools were sufficient to validate all the concepts presented before and we were capable to automatically: analyze the code, infer code transformations and generate customized systems for the application.

Each exploration of the application parallelization tool infers an ASIP and applies loop transformations on the input application, generating a Pareto front of solutions and selecting the best results. Taking into account that the longest exploration took eleven seconds, this exploration is very fast compared to a handmade DSE. On the other hand, at the time that this evaluation happened, this tool was able to explore only three code transformations: loop fusion, loop tiling and data partitioning.

As many different solutions were generated for each application we decided to choose the fastest one. The tool generates a graph with the transformations selected and a XML file describing the system (see 4.1.3). By combining the XML file and the configurable library of components (see 4.2) we were able to create the customized ASIP for the application. To validate the communication DSE proposed in this document we decided to generate three different customized ASIPs for each application. The first one (custom_1bus) connects all clusters through a single bus. This configuration extremely reduces the overall connectivity. The second one (custom_pipe) is the model proposed which connects the clusters in a pipelined way. The last one (custom_fully) connects all clusters together with a bus for each cluster's output connected to all clusters' input. It is important to emphasize

Name	**Description**
Lpsf	Low pass spatial filter
Down	Down sampling
2mm	Two matrix multiplication
3mm	Three matrix multiplication
Mvt	A matrix vector product and transpose
Syrk	Symmetric rank-k operations
Syr2k	Symmetric rank-2k operations

Table 5.1: Benchmarks

that the custom ASIPs are different for each application. In other words, the custom_pipe generated for 2mm will be different from the custom_pipe generated for 3mm.

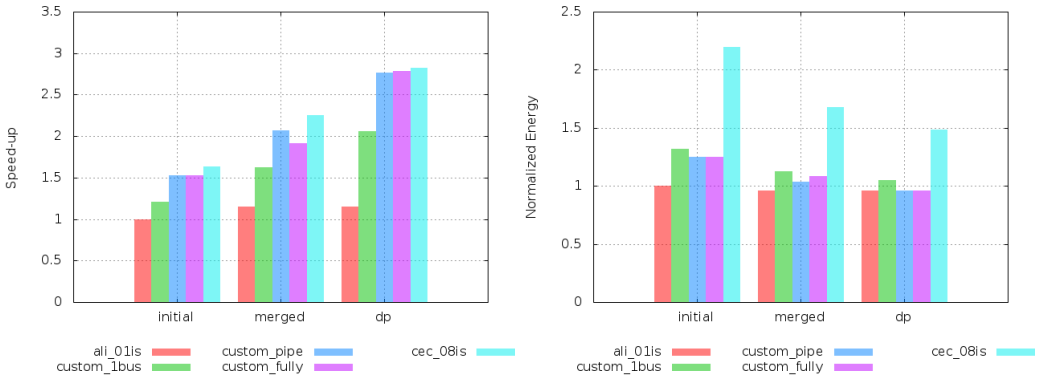
In order to evaluate the results we decided to compare our customized ASIPs against two different architectures. The first one (called ali_01is) is a scalar processor containing only one issue slot. This architecture aims to represent the least parallel architecture possible and it tends to take many cycles to finish, but using little energy. On the other hand, the second architecture (cec_08is) is a very large processor containing eight issue slots fully connected. We did not generate vector issue slot because the Application Parallelization tool is not able to automatically infer vectorization yet. We know that vectorization can decrease a lot the number of cycles and also reduce the power consumption. However, with these ASIPs we can still verify if the flow is implying a correct number of clusters and issue slots, taking advantage of the parallelization available in the code.

All the application versions were then compiled to all target ASIPs and then the systems were simulated to measure cycles, power and utilization. All results were normalized and the result for the initial version of the code mapped on the ali_01 ASIP was selected as reference. The system level communication and the generic API for a user-oriented multi-ASIP system proposed in this document were used to generate and simulated the system.

Figure 5.1 shows the speed-up and energy results for 2mm, 3mm and syr2k. In this figure "initial" means the initial code, "merged" means only fusion explored and "dp" means that fusion and data partitioning were explored. As we can observe, for all applications the transformations applied on the code speed-up the execution. In the 3mm we achieved a speed-up of almost 3.25 which is a good result for only three code transformations (without vectorization). In addition, we can observe that the number of cycles for the custom_pipe and for the custom_fully is very similar. Also, the pipelined way consumes less energy. This means that the pipelined connection is a good approach and a fully connected set of clusters is not needed. Another important observation is that in some cases the custom_pipe performed better than the custom_fully, which shouldn't happen. We trust that the explanation come from the compiler. As we were not using the compiler on the exhaustive mode (which search for the optimal schedule), we believe that the pipelined structured helps the compiler to keep some variables in the correct cluster (instead of passing to other clusters, which could happen more frequently in the fully connected case because it has less restrictions in the connectivity). The custom_1bus is always slower and uses more energy than the other customized ASIPs. This validates the inter issue slot communication approach proposed.

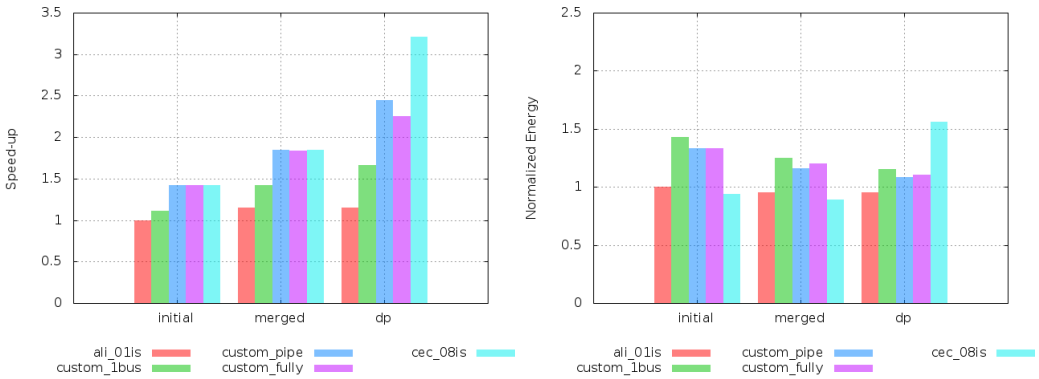
Comparing the customized ASIP with ali_01is we can conclude that customized can perform much faster (at least 1.5 times) without increasing much energy (around 20% more). Moreover, in comparison with the cec_08is it's possible to perceive that the speed-up is almost similar, but the energy consumption is much lower. Thus, it is possible to conclude that the DSE has made an acceptable design choice.

Figure 5.2 shows the final result (all transformations explored) for all the benchmarks selected. We can reach the same conclusions as before by looking to the other applications. In addition, we can observe the graph with the ASIP's utilization. As expected, the ali_01is has the biggest utilization because it has only one IS that is almost always executing an operation. On the other hand, cec_08is has the lowest utilization because we do not have enough parallelism to take advantage of all eight issue slots. The custom ASIPs show a tolerable utilization is some cases, but it should be improved by exploring more code transformations for instance.



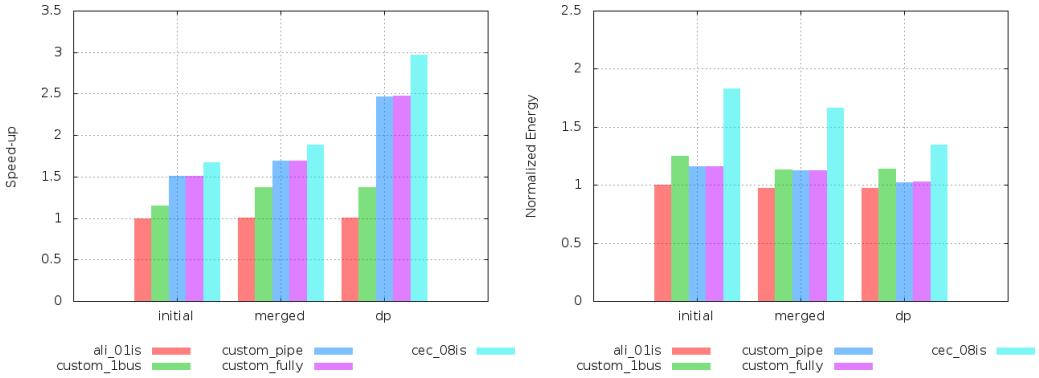
(a) 2mm: Speed-up results

(b) 2mm: Energy results



(c) 3mm: Speed-up results

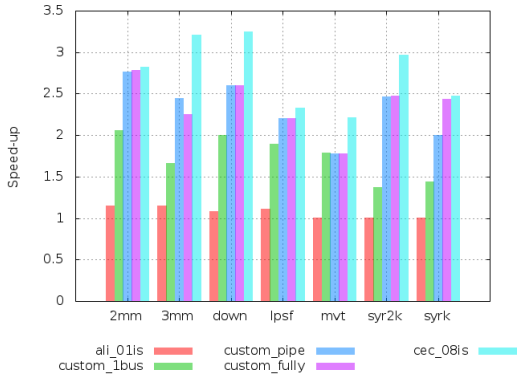
(d) 3mm: Energy results



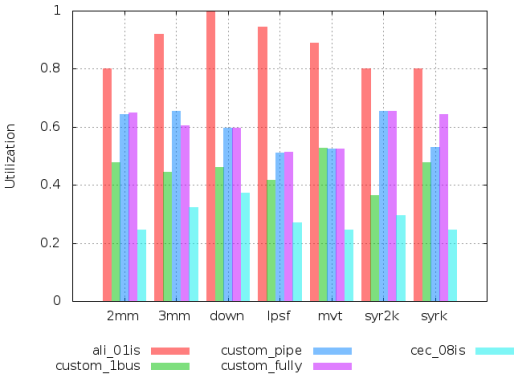
(e) syr2k: Speed-up results

(f) syr2k: Energy results

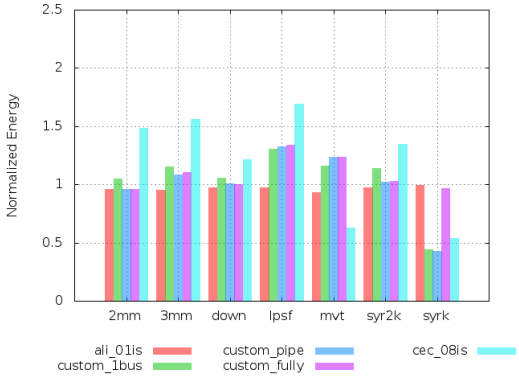
Figure 5.1: Code transformation results for 2mm, 3mm and syr2k.



(a) Final speed-up results



(b) Final utilization results



(c) Final energy results

Figure 5.2: Final results.

6 CONCLUSION AND PERSPECTIVES

In this document we have discussed the ASAM flow, introduced its challenges and issues and proposed solutions to different matters. In addition, part of the flow was evaluated and validated and the experimental results revealed an important design-time speed-up. Moreover, a component based design for multi-ASIP platform was proposed and validated. We have introduced and compared several strategies to connect different issue slots and we have shown that our approach give the best trade-off regarding application execution time and energy consumption.

Still, there is much work to be completed on the ASAM project. First, the trade-off involving the Model of Computation and the time spent to port the code should be further investigated and supportive methods to accelerate the porting phase should be proposed. In addition, the flow should be stressed with much more (and more complex) use cases in order to continue the integration process and to generate real consumer applications. The vectorization should be introduced and validated as soon as possible because we can only reach significant improvements on this kind of ASIPs systems through an efficient use of vectors. Finally, more code transformations should be implemented. As shown in the project description (Appendix G) there are much more code transformations that help to exploit the parallelism of the ASIPs and they are required to achieve important speed-up, targeting at least 20 times faster than the initial code as it is seen in the industry nowadays.

Future work could involve new features inside the ASAM flow. For example, fixed point conversion (numerical analysis), register size reduction (in number of bits) and custom operations could be investigated in the ASIP generation, optimizing even more the final result.

In conclusion, even though we have many things to do in order to finish the entire ASAM flow, we already have some parts of the full flow working that could help the designer, generating information about alternatives architectures and reducing the design-time. If one day we can have a complete flow working and generating real optimized consumer systems, the impact on the market can be huge and it can allow spreading embedded systems much faster and energy-efficient.

REFERENCES

Movidius Myriad SoC. **Software Programmable Media Processor, Movidius Myriad SoC, Project website**. Available on: <http://movidius.com/>. Last access: May 2013.

Intel Mobile SoCs. **Programmable Image Signal Processor, Medfield, Clover Trail, Project website**. Available on: <http://www.intel.com/>. Last access: May 2013.

JOZWIAK, L. **Quality-driven design in the system-on-a-chip era: Why and how?**, Journal of Systems Architecture, vol. 47, no. 3-4, pp. 201-224, Apr. 2001.

JOZWIAK, L; ONG, S. **Quality-driven model-based architecture synthesis for real-time embedded SoCs**, Journal of Systems Archi-tecture, vol. 54, no. 3-4, pp. 349-368, Mar. 2008.

DENSMORE, D; PASSERONE, R. **A Platform-Based Taxonomy for ESL Design**, IEEE Design & Test of Computers, vol. 23, no. 5, pp. 359-374, May 2006.

UCBerkeley. **Metropolis: Design Environment for Heterogeneous Systems, Project website**. Available on: <http://embedded.eecs.berkeley.edu/metropolis/platform.html>. Last access: May 2013.

SCHREIBER, R; ADITYA, S; MAHLKE, S; KATHAIL, V; RAU, B; CRONQUIST, D; SIVARAMAN, M. **Pico-npa: High-level synthesis of nonprogrammable hardware accelerators**, J. of VLSI Signal Proc. (2002)

MURRAY, A; FRANKE, B. **Compiling for automatically generated instruction set extensions**, In: proceedings of the Tenth International Symposium on Code Generation and Optimization, CGO '12 pp. 13-22. ACM, New York, NY, USA (2012).

FlexASP project. **TTA-based co-design environment, Project website**. Available on: <http://tce.cs.tut.fi/>. Last access: May 2013.

KATHAIL, V; ADITYA, S; SCHREIBER, R; RAU, B; CRONQUIST, D; SIVARAMAN, M. **PICO: automatically designing custom computers**, Computer, vol. 35, no. 9, pp. 39-47, 2002.

TERECHKO, A; THENAFF, E; GARG, M; EIJNDHOVEN, J; CORPORAAL, H. **Inter-cluster communication models for clustered vliw processors**, in High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on. IEEE, 2003, pp. 354-364.

JOZWIAK, L; LINDWER, M; CORVINO, R; MELONI, P; MICCONI, L; MADSEN, J; DIKEN, E; GANGADHARAN, D; JORDANS, R; POMATA, S; POP, P; TUVERI, G; RAFFO, L. **ASAM: Automatic Architecture Synthesis and Application Mapping**, DSD 2012 - 15th Euromicro Conference on Digital System Design, pages 1-11, Cesme, Izmir, Turkey, 2012.

JOZWIAK, L; LINDWER, M. **Issues and Challenges in Development of Massively-Parallel Heterogeneous MPSoCs Based on Adaptable ASIPs**, PDP 2011 - 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing, pages 483-487, 2011.

JOZWIAK, L; LINDWER, M. **Automatic Architecture Synthesis and Application Mapping for Application-specific Customizable MPSoCs**, HOPES 2010 - the First International Workshop on Hands-on Platforms and Tools for Model-based Engineering of Embedded Systems, in the scope of ECMFA 2010 - the Sixth European Conference on Modelling Foundations and Applications, pages 105-110, Paris, France, 2010.

ASAM. **ASAM Project website**. Available on: <http://www.asam-project.org>. Last access: February 2013.

JOZWIAK, L; LINDWER, M; MADSEN, J. **Model-based MPSoC Architecture Synthesis for Highly-demanding Embedded Applications**, DATE 2011 Tutorial presentation.

SILICONHIVE. **Hivelogic Configurable Parallel Processing Platform**, SiliconHive, 2010.

ASAM. **Polybench: The Polyhedral Benchmark Suite, Project website**. Available on: <http://www.cse.ohio-state.edu/ pouchet/software/polybench/>. Last access: May 2013.

GLITIA, C; DUMOND, P; BOULET, P. **Array-ol with delays, a domain specific specification language for multidimensional intensive signal processing**, Multidimensional Syst. Signal Process., vol. 21, no. 2, pp. 105-131, Jun. 2010. [Online]. Available: http://dx.doi.org/10.1007/s11045-009-0085-4

APPENDIX A EXAMPLE OF USER CONSTRAINS (XML FILE)

```
<?xml version="1.0" encoding="utf-8" ?>
<system id="1" area="10" power="12" frequency="133">
  <application id="1" name="jpeg0" deadline="220">
  </application>
</system>
```

APPENDIX B EXAMPLE OF PLATFORM DESCRIPTION (XML FILE)

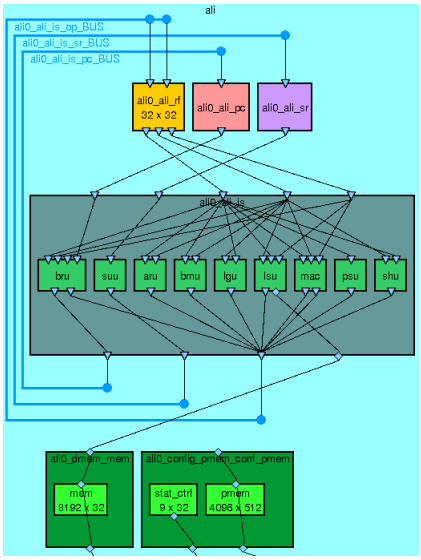
```
<?xml version="1.0" encoding="utf-8" ?>
<platform id="1">
<node id="1">
     <name>PU1</name>
     <type>PE</type> </node>
<node id="2">
     <name>PU2</name>
     <type>PE</type>
</node>
<node id="3">
     <name>PU3</name>
     <type>PE</type>
</node>
<node id="4">
     <name>PU4</name>
     <type>bus</type>
     <width id="1">32</width>
     <frequency id="1">10</frequency>
     <width id="2">16</width>
     <frequency id="2">10</frequency>
</node>
<edge source="1" target="4" />
<edge source="2" target="4" />
<edge source="3" target="4" />
<edge source="4" target="1" />
<edge source="4" target="2" />
<edge source="4" target="3" />
</platform>
```

APPENDIX C EXAMPLE OF TASK AND SYSTEM ANALYSIS (XML FILE)

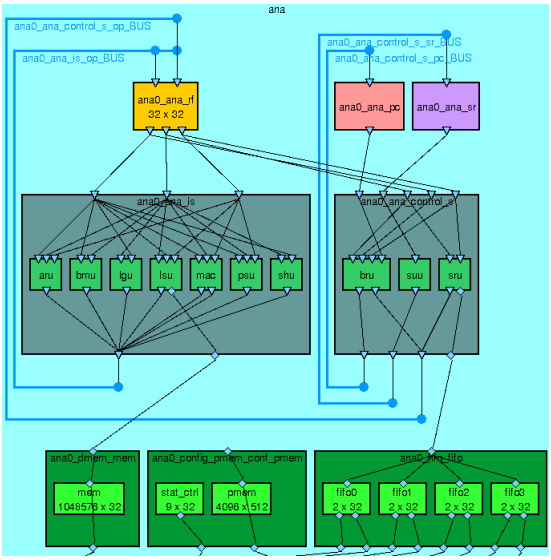
```
<?xml version="1.0" encoding="utf-8" ?>
<solutions>
  <solution id="1">
    <cluster id="1:1">
       <node id="1:1:1" >
          <name>T1</name>
          <WCET id="1">47</WCET>
          <WCET id="2">77</WCET>
          <WCET id="3">97</WCET>
          <WCET id="4">107</WCET>
          <WCET id="5">110</WCET>
          <WCET id="6">135</WCET>
          <priority>1</priority>
       </node>
       <node id="1:1:3" >
          <name>T2</name>
          <WCET id="1">55</WCET>
          <WCET id="2">60</WCET>
          <WCET id="3">90</WCET>
          <WCET id="4">47</WCET>
          <WCET id="5">100</WCET>
          <WCET id="6">35</WCET>
          <priority>3</priority>
       </node>
    </cluster>
    <cluster id="1:2">
       <node id="1:2:5" >
          <name>T3</name>
          <WCET id="1">60</WCET>
          <WCET id="2">65</WCET>
          <WCET id="3">100</WCET>
          <WCET id="4">137</WCET>
          <WCET id="5">150</WCET>
          <WCET id="6">175</WCET>
          <priority>5</priority>
       </node>
    </cluster>
    <cluster id="1:4">
    <edge id="1:4:2" source="1:1:1" target="1:1:3">
      <name>MT1</name>
      <data>7680</data>
    </edge>
```

```
   <edge id="1:4:4" source="1:1:3" target="1:2:5">
     <name>MT2</name>
     <data>1536</data>
   </edge>
   <edge id="1:4:6" source="1:1:1" target="1:2:5">
     <name>MT3</name>
     <data>696</data>
   </edge>
  </cluster>
  <parametersList>
      <parameters clusterId="1:1">
         <config id="1" area="70" power="50" />
         <config id="2" area="50" power="40" />
         <config id="3" area="30" power="30" />
         <config id="4" area="20" power="30" />
         <config id="5" area="10" power="10" />
         <config id="6" area="10" power="5" />
      </parameters>
      <parameters clusterId="1:2">
         <config id="1" area="80" power="60" />
         <config id="2" area="40" power="40" />
         <config id="3" area="10" power="20" />
         <config id="4" area="10" power="10" />
         <config id="5" area="5" power="10" />
         <config id="6" area="5" power="10" />
      </parameters>
      <parameters clusterId="1:4">
         <config id="1" area="80" power="60"/>
         <config id="2" area="90" power="40"/>
      </parameters>
  </parametersList>
</solution>
</solutions>
```

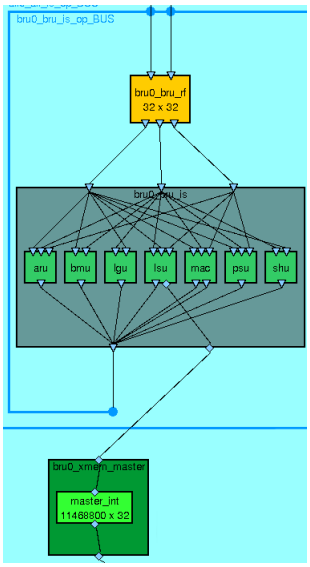
APPENDIX D EXAMPLE OF DEFAULT COMPONENTS



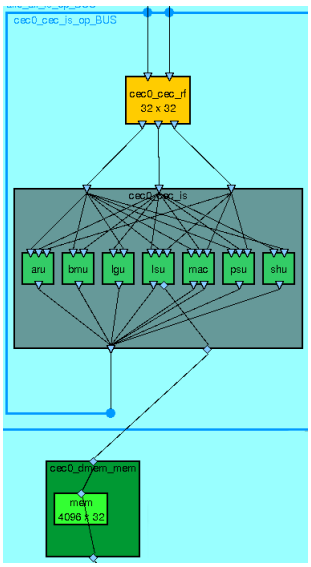
(a) Ali: 1 IS Sequencer.



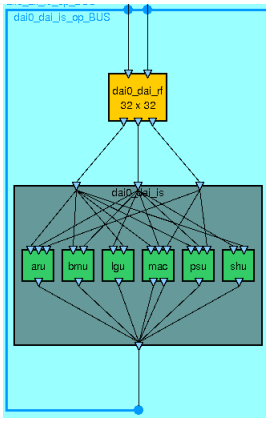
(b) Ana: 2 IS Sequencer with FIFOs.



(c) Bru: Scalar IS with scalar master interface.

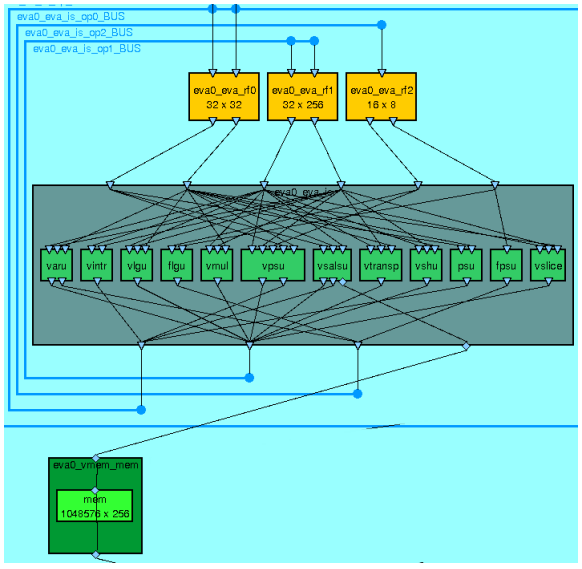


(d) Cec: Scalar IS with scalar memory.

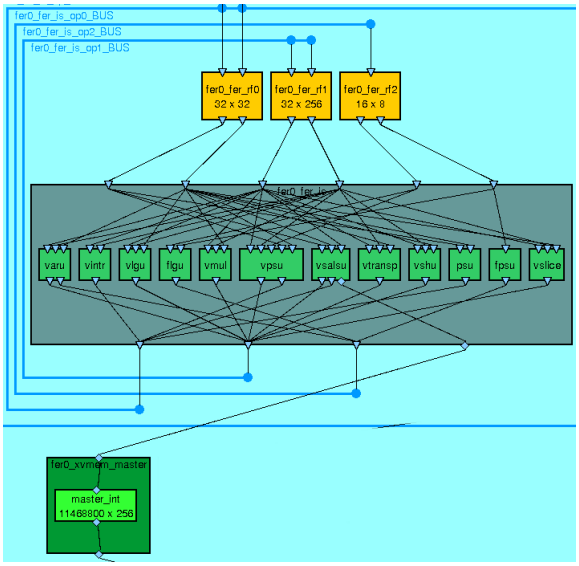


(e) Dai: Scalar IS with no memory.

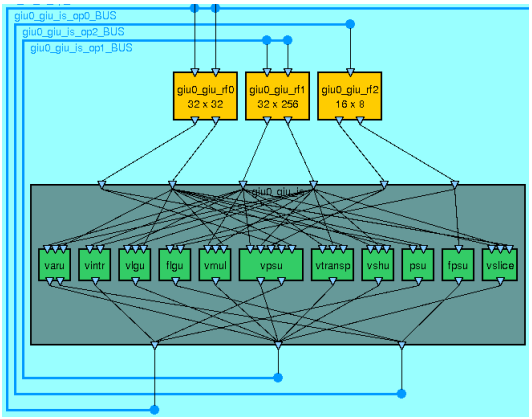
Figure D.1: Only scalar components.



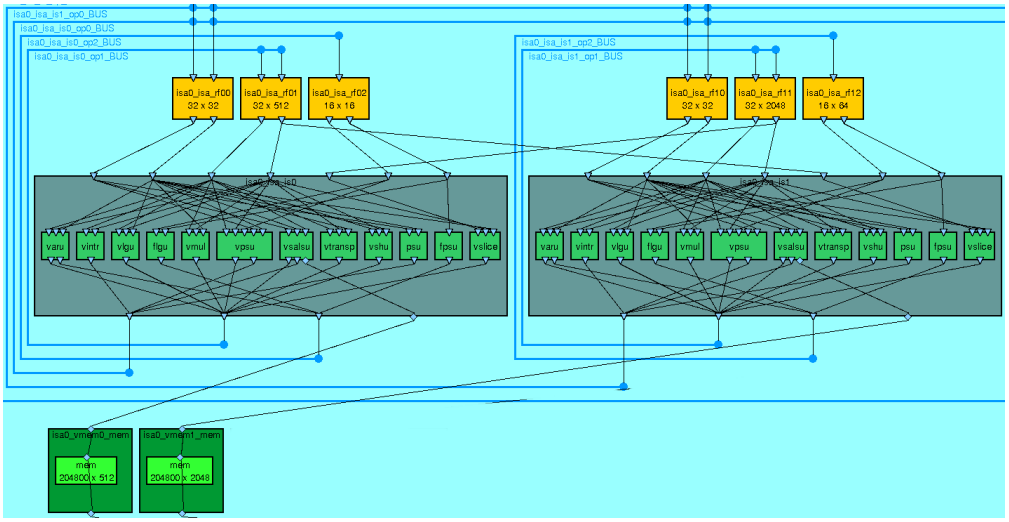
(a) Eva: Vector and scalar IS with vector memory.



(b) Fer: Vector and scalar IS with vector master interface.



(c) Giu: Vector and scalar IS with no memory.



(d) Isa: Two different vector and a scalar IS communicating through slice operations and connected to two different vector memories.

Figure D.2: Vector and scalar components.

APPENDIX E HOST CODE EXAMPLE

```
hrt_cell_load_program(CELL, "fir");
for ( i = 0; i < N_h; i++ ){
for ( j = 0; j < n_width; j++ ){
hrt_asam_indexed_store_signed_vector(CORE, CELL, fir, (&in_image[i][j*
    CORE_NWAYS]), image_hive, (i*(n_width)+j));
}
}


for ( i = 0; i < N_h2; i++ ){
for ( j = 0; j < n_width2; j++ ){
hrt_asam_indexed_store_signed_vector(CORE, CELL, fir, (&in_image2[i][j*
    CORE_NWAYS]), image_hive2, (i*(n_width2)+j));
}
}


hrt_asam_mem_store_nooffset( CELL, fir, height_hive, &height, sizeof(
    int) );
hrt_asam_mem_store_nooffset( CELL, fir, width_hive, &n_width, sizeof(
    int) );
hrt_asam_mem_store_nooffset( CELL, fir, height_hive2, &height2, sizeof(
    int) );
hrt_asam_mem_store_nooffset( CELL, fir, width_hive2, &n_width2, sizeof(
    int) );

hrt_cell_start_function(CELL, fir);
hrt_cell_wait(CELL);


for ( i = 0; i < N_h; i++ ){
for ( j = 0; j < n_width; j++ ){
hrt_asam_indexed_load_signed_vector(CORE, CELL, fir, image_hive, (i*(
    n_width)+j), (&out_image[i][j*CORE_NWAYS]));
}
}

for ( i = 0; i < N_h2; i++ ){
for ( j = 0; j < n_width2; j++ ){
hrt_asam_indexed_load_signed_vector(CORE, CELL, fir, image_hive2, (i*(
    n_width2)+j), (&out_image2[i][j*CORE_NWAYS]));
}
}
```

APPENDIX F ASIP CODE EXAMPLE

```
#include <hive/asam_support_cell.h>

#include "isp00.c"

SYNC_WITH(core01_IN) MEM(HIVE_MEM_core01_A)AT(HIVE_ADDR_core01_A)
    __intern int A[core01_N_BUFFERS_IN][N][N];
SYNC_WITH(core01_IN) MEM(HIVE_MEM_core01_x1)AT(HIVE_ADDR_core01_x1)
    __intern int x1[core01_N_BUFFERS_IN][N];
SYNC_WITH(core01_IN) MEM(HIVE_MEM_core01_x2)AT(HIVE_ADDR_core01_x2)
    __intern int x2[core01_N_BUFFERS_IN][N];
SYNC_WITH(core01_IN) MEM(HIVE_MEM_core01_y_1)AT(HIVE_ADDR_core01_y_1)
    __intern int y_1[core01_N_BUFFERS_IN][N];
SYNC_WITH(core01_IN) MEM(HIVE_MEM_core01_y_2)AT(HIVE_ADDR_core01_y_2)
    __intern int y_2[core01_N_BUFFERS_IN][N];

SYNC_WITH(core01_OUT) MEM(HIVE_MEM_core01_x1_out)AT(
    HIVE_ADDR_core01_x1_out)extern int x1_out[core01_N_BUFFERS_IN][N];
SYNC_WITH(core01_OUT) MEM(HIVE_MEM_core01_x2_out)AT(
    HIVE_ADDR_core01_x2_out)extern int x2_out[core01_N_BUFFERS_IN][N];

static int db_in[core01_N_CHANNELS_IN] = {0};
SYNC_WITH(core01_IN) static int db_read[core01_N_CHANNELS_IN] = {0};
SYNC_WITH(core01_OUT) static int db_out[core01_N_CHANNELS_OUT] = {0};

void ND_01() ENTRY
{
int i;

// request input
std_snd(HIVE_FCTRL_core01_host_00, HIVE_FIFO_core01_host_00, db_in[0])
    SYNC(core01_IN);
modinc(db_in[0], core01_N_BUFFERS_IN) SYNC(core01_IN);

for ( i = 0; i < 8; i++ ){

if( i > 0 ){
// output sent
std_snd(HIVE_FCTRL_core01_host_05, HIVE_FIFO_core01_host_05, db_out[0])
     SYNC(core01_OUT);
}

if( i < 8-1 ){
// request input
```

```
std_snd(HIVE_FCTRL_core01_host_00, HIVE_FIFO_core01_host_00, db_in[0])
    SYNC(core01_IN);
modinc(db_in[0], core01_N_BUFFERS_IN) SYNC(core01_IN);
}

// wait for input
std_rcv(HIVE_FCTRL_core01_host_00, HIVE_FIFO_core01_host_00, db_read
    [0]) SYNC(core01_IN);

// request to write output
std_rcv(HIVE_FCTRL_core01_host_05, HIVE_FIFO_core01_host_05, db_out[0])
     SYNC(core01_OUT);

// real function
isp_kernel_mvt( x1[db_read[0]], x2[db_read[1]], y_1[db_read[2]], y_2[
    db_read[3]], A[db_read[4]], x1_out[db_out[0]], x2_out[db_out[1]] );


}

// output sent
std_snd(HIVE_FCTRL_core01_host_05, HIVE_FIFO_core01_host_05, db_out[0])
     SYNC(core01_OUT);

}
```

APPENDIX G PROJECT DESCRIPTION <TG1>

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTE OF INFORMATICS
COMPUTER ENGINEERING

FELIPE AUGUSTO CHIES

Validation and Evaluation of the ASAM - Automatic Architecture Synthesis and Application Mapping - Flow

Project Description

Prof. Dr. Luigi Carro
Universidade Federal do Rio Grande do Sul
Advisor

Menno Lindwer
Intel Corporation
Coadvisor

Eindhoven, February 2013

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

ANSI-C	Americal National Standards Institute version of the 'C' programming language.
API	Application Programmers Interface
ARU	Arithmetic Unit
ASIP	Application-specific Instruction-set Processor
C++	Programming language, based on the 'C' language. SystemC is built on top of C++.
CIO	Intel'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
FU	Function Unit
GeneSys	HiveLogic tool for instantiating and interconnecting IP blocks (either HiveLogic or customer-supplied)
HDL	Hardware Description Language
HiveCC	Intel's Software Development Kit, including ANSI-C compiler

HiveLogic	Intel's configurable parallel processing flow
HiveRT	HRT: 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
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	Multipy-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
OLP	Operation-Level Parallelism: single operations performing multiple tasks simultaneously which, on a RISC processor, would have taken multiple operations
PC	Program Counter
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
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
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
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

LIST OF FIGURES

LIST OF CODE

ABSTRACT

Progressively, more complex and sophisticated embedded systems are required to perform real-time computations with high demands regarding energy, power, area, cost and efficiency. This result in serious design and development challenges, such as: multi-objective MPSoC optimization, adequate resolution of numerous complex design trade-offs, reduction of the time-to market, development costs without compromising the system quality, among others. These challenge systems can be effectively exploited only through the use of more adequate application-specific system architectures and more integrated system IP modules. This generates one need for integrated tools able help the designer in this process. Based on this, the ASAM - Automatic Architecture Synthesis and Application Mapping - project targets a uniform process of automatic architecture synthesis and application mapping for heterogeneous multi-processor embedded systems based on adaptable Application Specific Instruction-set Processors (ASIPs). This document is devoted to introduce the Graduation Project that will be realized by Felipe Augusto Chies. The aim of this project is to validate and evaluate the ASAM flow.

Keywords: Embedded systems, heterogeneous multi-processor system-on-chip (MPSoC), customizable ASIPs, architecture synthesis, MPSoC and ASIP design automation.

1 INTRODUCTION

Progressively, more complex and sophisticated embedded systems are required to perform real-time computations with high demands regarding energy, power, area, cost and efficiency. Furthermore, these systems are required to be flexible enough to enable adequate reuse among different product versions, reaction to the market shifts, adherence to evolving standards or user requirements and easy modification during development or even their field of use.

This all results in serious design and development challenges, such as: multi-objective MPSoC optimization, adequate resolution of numerous complex design trade-offs, reduction of the time-to market, development costs without compromising the system quality, among others.

These challenge systems can be effectively exploited only through the use of more adequate application-specific system architectures and more integrated system IP modules. One approach used to achieve these objectives is the use of heterogeneous MPSoC platform based on adaptable ASIPs. This platform provides the flexibility required and enables programmable SoCs with performance close to that of hardwired ASICs, yet at low cost and with much shorter time to market.

The main problem 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.

Therefore, the development of the system has to be supported by new design methods and electronic design automation (EDA) tools for an adequate system-level design exploration, rapid development of high-quality hardware platforms, and efficient automatic mapping of applications on the platforms.

The traditional algorithm and software development approaches require an existing and stable computation platform (HW platform, compilers, etc.), while for the modern embedded systems the architecture, algorithms, hardware and software have to be application-specific and must be developed largely in parallel. In particular, the hardware platform of the application-specific MPSoCs based on adaptable ASIPs has to be developed in parallel with development and mapping of the MPSoC software on this platform.

Formalisms exist for the high-level description of processors of any nature (RISC/CISC/DSP, SIMD/VLIW, etc.) and SoCs involving processors, networks-on-chip, memories

and other blocks. Tools are available to generate synthesizable hardware designs, multi-processor compilers and simulators from these high-level descriptions. What are needed are effective methods and tools to support the actual construction of the high-level system and processor designs.

This document introduces the validation and the evaluation process of an automatic flow that aims to be able to achieve all the challenges presented before. All the research and results presented here are related to a European project ASAM (Automatic Architecture Synthesis and Application Mapping) being currently executed in the framework of the ARTEMIS program.

The first part of the document introduces the ASAM project. The next section aims to describe the evaluation strategy that is the main objective of this project. This section is a brief description off all work that should be completed. After that, some steps that are needed to be accomplished by the flow are explained making use of an example (Motion JPEG decoder). This example was created in the initial period of this project will help to have an idea about what we want to produce using the flow in the end of the project.

2 THE ASAM PROJECT

2.1 Objectives

The main aim of the ASAM project is to enhance the MPSoC design efficiency, while substantially improving the result quality. The project will develop a design methodology and a design flow that provides efficient exploration of the architecture and application mapping alternatives and trade-offs. The compilation process will efficiently select the most appropriate ASIP processor types for different parts of a given application, reuse and instantiate the selected generic ASIPs, extend them with new hardware implemented as parts of their application-specific data or control-paths, correspondingly restructure the application's software and implement the software on the so constructed application specific multi-processor platform. The project will also advance the state-of-the-art in application parallelization, partitioning, scheduling and mapping, needed to facilitate the design-space exploration and to deliver applications running efficiently on the platforms.

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 for the SoC sub-system;
- Distribution of the application's computation processes among different processors of the multi-processor platform;
- 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.

2.2 ASAM Flow and Partners

The ASAM project is supported by different partners, each one with especial responsibilities in the flow. The partners are: Intel Corporation (where this research is being conducted), Tu/e (Technische Universiteit Eindhoven), DTU (Danmarks Tekniske Universitet), Compaan, ACE Associated Computer Experts, UNICA (Universita degli Studi di Cagliari), TUBS (Technische Universitat Braunschweig).

The project is based on the Intel (former Silicon Hive) design flow and it intends to automatize some of the manuals processes and introduce new ones. The Intel flow can be seen in the Figure 2.1 and 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 new automatized flow - Figure 2.2 and 2.3 - that will be developed by the ASAM partners can be divided in the follow process:

Design Space Exploration (DSE) for a multi-ASIP system (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

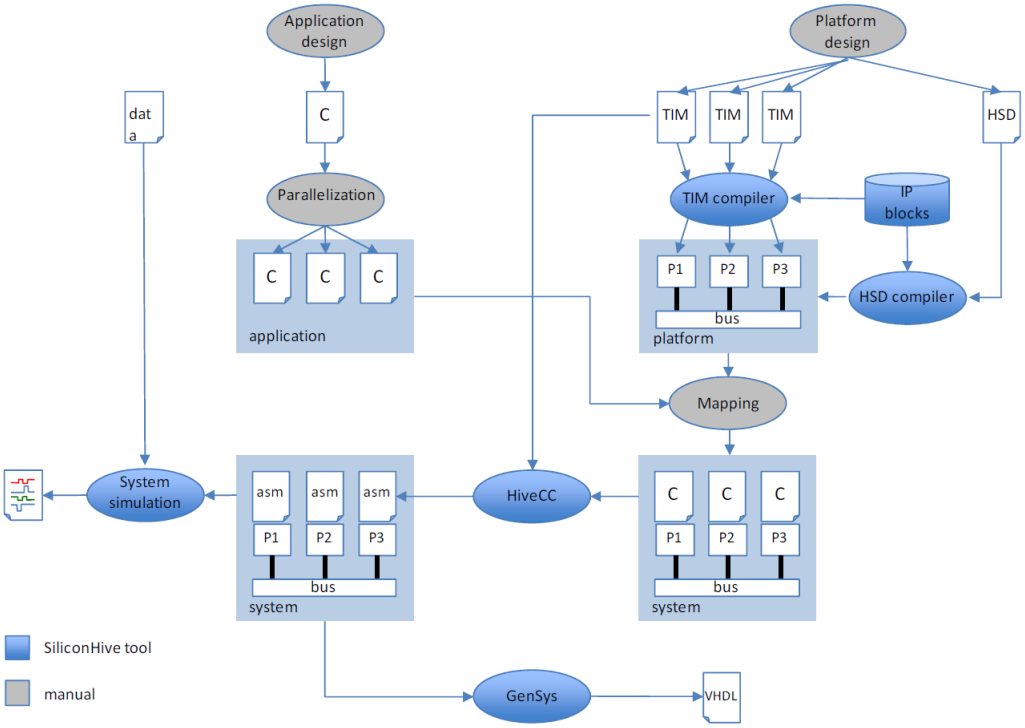


Figure 2.1: Intel design flow.

the inner loop ASIP design space exploration (ASIP or micro-level DSE - managed by Tu/e), and one for the outer loop of system level design space exploration (system or macro-level DSE - managed by DTU). Each of the DSE processes involves its own application analysis and restructuring, and computing system synthesis because they have to be performed for each architecture level with different precision and (partially) account for different aspects.

Application Analysis (TUe/DTU/Intel/Compaan): The application code written in C is translated into an equivalent KPN or other graph-based model exploring the parallelism of the application (managed by Compaan). 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.

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.

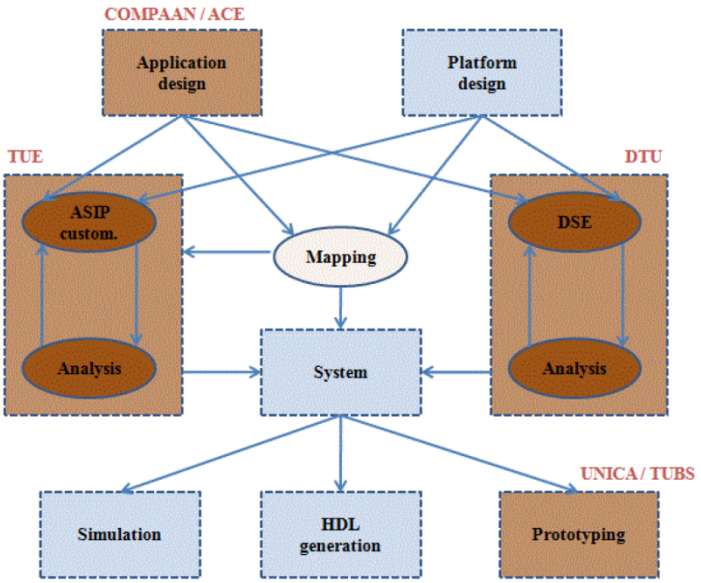


Figure 2.2: ASAM subsystems grouped by partners' tasks.

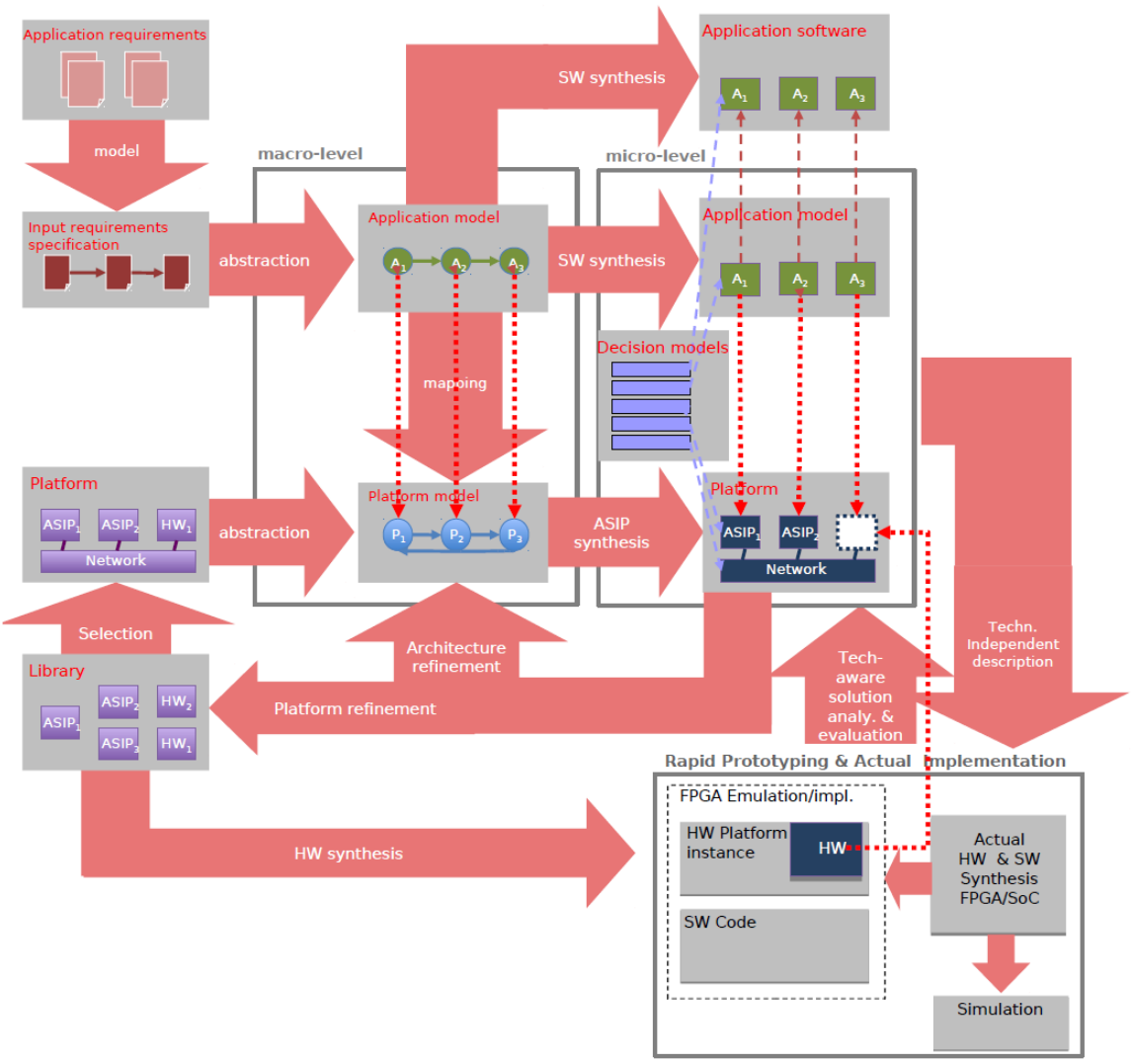


Figure 2.3: Complete diagram of the ASAM Flow.

3 EVALUATION STRATEGY

3.1 Description

More than only validate the ASAM flow, this research aims to give a feedback of the quality of the ASAM tools, their integration and the quality of the final results, based on the follow use-cases:

- Medical Application: Electrocardiogram (ECG) algorithm;
- Video encoding application: MPEG4 Encoder algorithm;
- Digital Hearing Aid System.

To ensure the completeness of the evaluation, these use-cases are issued from different domains, such as ultra-low power application for medical and multimedia, triggering different needs/requirements and focusing on different aspects of the proposed flow. The use-cases are provided with the present state-of-art details for each case study.

The ASAM flow will be evaluated in two different ways. The first scenario will evaluate the ASAM flow as a Novice User, in others words, the flow will run from a non-optimized application and platform. In this scenario, the ASAM flow should be able to generate one result that is on-par with 2010 state-of-the-art designs for the same application. In the second scenario, the ASAM flow will be used by an experienced team. In this case, the user should be able to quickly arrive at a solution which is on-par with 2013 state-of-the-art.

Under these considerations, four states of verification strategy were created. The first state consists of evaluating existing state-of the-art implementations of application use-cases, by using application-specific metrics. When evaluating the application implementations, the metrics reflect the intrinsic performance of the design (ratio timing performance / power consumption for example); when evaluating the design tools, the metrics reflect the way in which today's design techniques converge to an adequate solution (number of exploration paths and parameters usable by the exploration tool). These metrics can be found in Appendix C.

In the following state, these metrics will be compared to the industrial results provided by the partners of the project, in order to verify the behavior and the performance of these application use cases. The state 2 checks also the time to obtain the DSE results on application use case behavior and performance under standalone tool runs.

State 3 is essentially based on the verification of tool integration (interfaces and communication). This state also verifies operative delay time in making tool communication

available during the entire flow (changeover of input data formats, definition of max number of feedback loops from one tool to another during the optimization phase, and so on).

The last step of the approach consists of verification of the integrated design flow and calculating the improvement it brings compared to current design practices.

The ASAM project has one set of high-level goals, listed below:

- Integrated chain of tools;
- Complete process flow for development of embedded systems;
- Support architecture exploration;
- Supporting concurrent validation and verification at different abstraction levels;
- Constructs heterogeneous computing architectures instantiated from generic platform;
- Flexibility and adaptability to trade off performance, resources, and power usage;
- Generated platforms enable massive real-time data processing;
- Enable composition of platform-independent software on concurrent systems;
- Relevant for a broad range of applications: consumer, multi-media, telecom, imaging.

Derived from these high-level goals, Appendix D to G provide a number of detailed operational targets, functional requirements, technical objectives, and verification requirements. These data were used to generate different scenarios requirements for each application. These scenario requirements will be used as a checklist to validate the flow in the evaluation process.

3.2 Initial Steps

In order to perform all the states of the evaluation process described above, different initial steps were required. It was necessary to understand the whole flow that will be executed to be able to analyze the system created and, in the future, to run the ASAM flow as an Expert User. Thus, the following tasks were completed in a first moment:

- Study how to write a simples cores using TIM files (format used by Intel to describe the internal representation of the cores);
- Learn how to create a system using HSD files (format used by Intel to describe the internal representation of the systems);
- Understand how to instantiate and program multiples cores in a system.

With a good knowledge of the Intel/VIED tools, the next step was to study the current systems developed by Intel/VIED and their capabilities. This study helped to have an idea about the state-of-the-art of the systems that are being developed.

After this initial learning process, the objective is to assemble all the modules developed by the partners of the ASAM project in order to have the flow working. Before starting the evaluation with the use-cases proposed, one first application will be tested to ensure that the flow is working well. The application chosen is a Motion JPEG encoder.

With the Motion JPEG encoder example running well on the ASAM flow, the results will be evaluated in order to validate if the flow is able to create one system optimized to the Motion JPEG application. In the end, after having tested the flow as a novice and expert user, the flow should be validated to all the requirements introduced in Appendix H and we should be able to determine the improvements over the current Intel/VIED flow.

After the first steps mentioned before, the evaluation described in the evaluation strategy should start using the Use Cases. This evaluation will be the focus of this project.

4 MAPPING AND OPTIMIZING A MOTION JPEG ENCODER

In the following pages we describe the steps needed to map the consumer-oriented MJPEG (Motion JPEG) encoder on an ASAM multi-ASIP system. The steps that are introduced here can be used later as a guide that shows how some subsystems of the ASAM flow are supposed to work. The MJPEG encoder application is based on a JPEG encoder. It continuously encodes each input frame from an MJPEG flow - on (y, u, v) format - generating a sequence of JPEG images as output.

The next two sections (Before Mapping and Processor Architecture Features) should be viewed only as an introduction of the application and an illustration of the steps to be taken for obtaining an initial representative system on which the application will be mapped. Although all these steps were followed manually, they follow the same method as the first DSE steps of the ASAM flow.

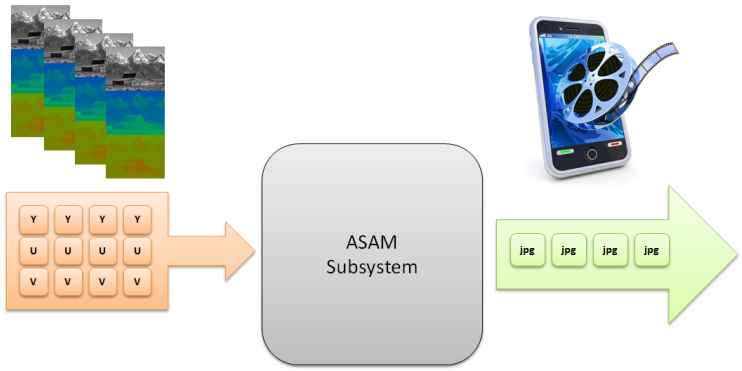


Figure 4.1: Graphical representation of MJPEG encoder.

4.1 Before Mapping

There are some things that we need to do before starting mapping an application. First, we need to certify that the application is ANSI-C compliant. In addition, we have to be sure that the application is validated and we need to have a set of tests ready to be used when we'll need to verify the application behavior after several modifications on the code.

More important than that, we need to know what we want to archive regarding time, power, area, among others. In our case we want to archive 24 frames per second. Area and power will be analyzed later on the project.

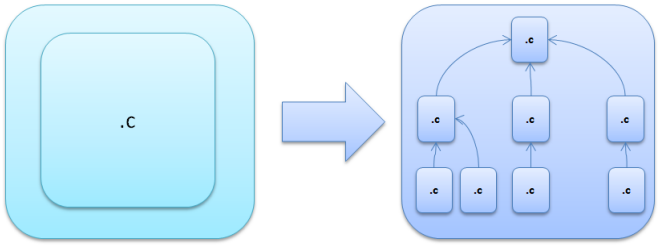


Figure 4.2: Initial C code needs to be split

4.1.1 Rewriting the Application

As mentioned before, the application should be supplied to the flow as strict ANSI-C compliant code. Those parts that are not ANSI-C compliant, need to be adapted. In addition, as we want to optimize the application, we should look for parts of the code that can be split in order to run it in parallel or, at least, be able to map it on different ASIPs, where each ASIP will be optimized for one of the parts.

4.1.2 Make use of a Model of Computation

As KPN (Kahn Process Networks) is the MoC (Model of Computation) chosen to be used on ASAM, the same model was used to map this application. Besides providing a model of computation, KPNs also help to handle the communication between different tasks and it can be very useful when mapping the application because different nodes (functions) can be unfolded many times and so mapped on different cores.

One thing that we need to take into account before using a MoC is that it requires extra effort. The code should be modified in order to be compatible with the application that will generate the MoC. In our case, we've used Compaan Compiler to generate the KPN. The code used as input to the tool is shown below.

```
void mjpeg(void) {
  int t, j, i;
  THeaderInfo hi;
  TPackets    stream;
  TBlocks     block[VNumBlocks][HNumBlocks];
  for (t = 0; t < NumFrames; t++) {
    initVideoIn (&hi);
    for (j = 0; j <= VNumBlocks; j++) {
      for (i = 0; i <= HNumBlocks; i++) {
        mainVideoIn(&block[j][i]);
        mainDCT(block[j][i], &block[j][i]);
        mainQ(block[j][i], &block[j][i]);
        mainVLE(block[j][i], &stream);
        mainVideoOut(hi, stream);
      }
    }
  }
}
```

Code 4.1: New main created to use a MoC.

We observe that all functions should be written from the template F ($type\ input0$, $type\ input1$, ..., $type* output0$, $type* output1$, ...). Figure 4.3 shows two possible KPNs generated by the tool.

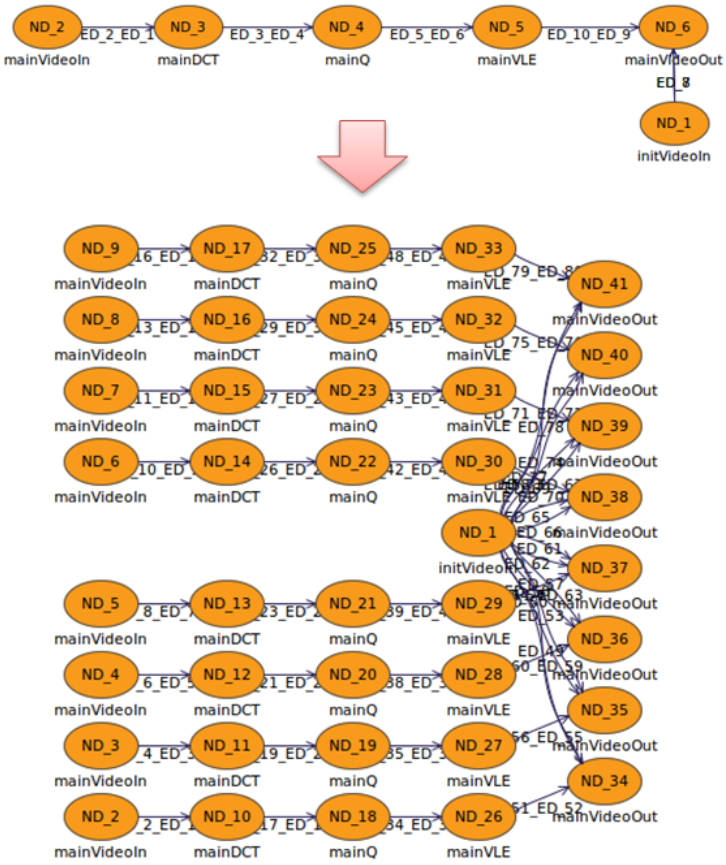


Figure 4.3: KPN graph unfolded eight times.

4.1.3 Clustering

Clustering is a difficult task because we have to make use of different assumptions, which means that we can make mistakes and create non-efficient solutions. The reason why it's hard to create good cluster is because we still do not know on which system we will map the clusters. Figure 4.4 shows different ways that we could have mapped the MJPEG encoder.

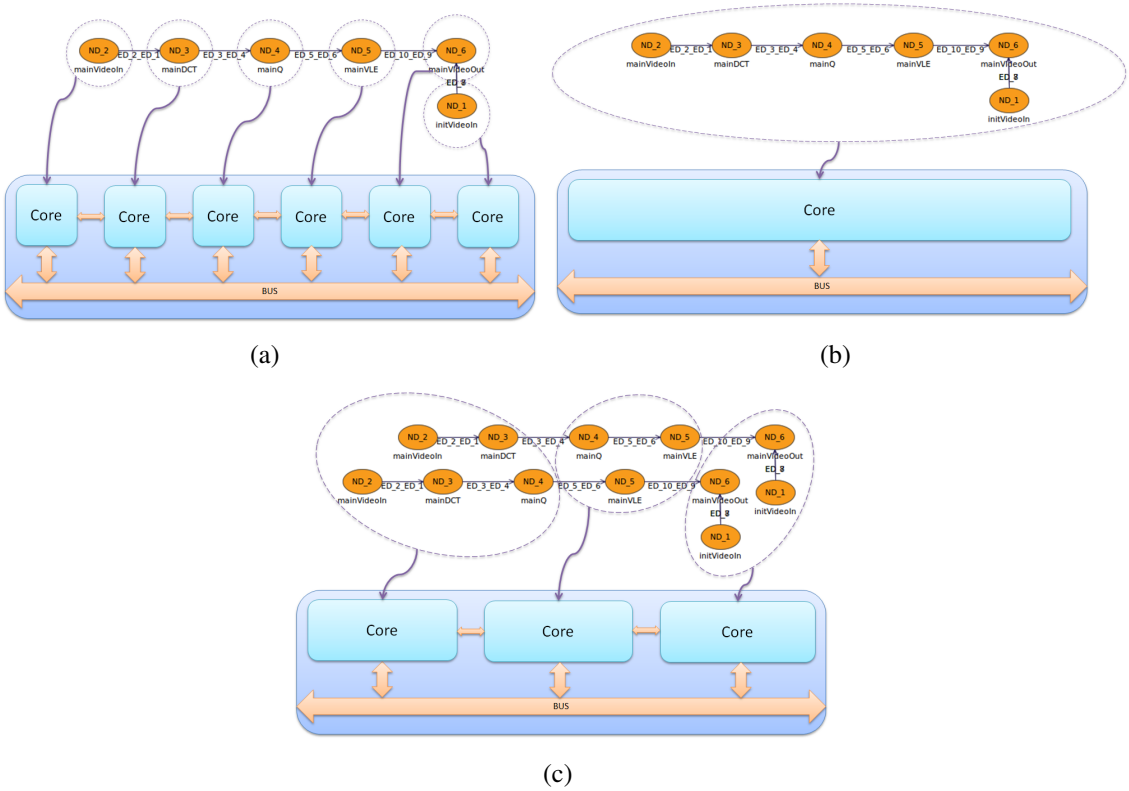


Figure 4.4: Different ways of clustering an application.

A good way to start the clustering process is analyzing the application (profiling for example). The problem is that we need to know where the cluster will be mapped to do a good analysis. On other hand, to choose the ASIP onto which we will map a cluster, we need to know the clusters to be able to choose the best ASIP available that can run the cluster in an optimized way. To solve this "chicken-egg" problem we start analyzing the tasks mapping them on a "default" core. That will give us some idea about the time spent to execute each task on this core. Another important thing to do here is to determine for each KPN node certain characteristics, such as: vectorizable, non-vectorizable, I/O-bound, CPU-bound, among others. These kinds of analyses allow us to choose the correct kind of ASIP, for example an ASIP with vector operations (for vectorizable code) or an ASIP with parallel access to memories (for I/O-bound code).

Figure 4.5 shows the results that were obtained investigating the MJPEG code. We ran the code on Core04 (Appendix A, Figure A.1) which is a VLIW scalar core. The total number of cycles was used to calculate the amount of time. The communication between tasks was not taken into account during this phase. The cluster chosen to start mapping the application can be found at Figure 4.6.

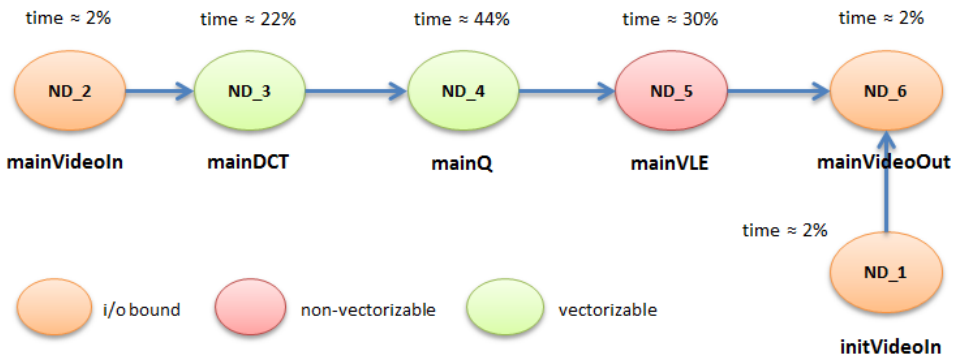


Figure 4.5: Analysis of MJPEG's nodes.

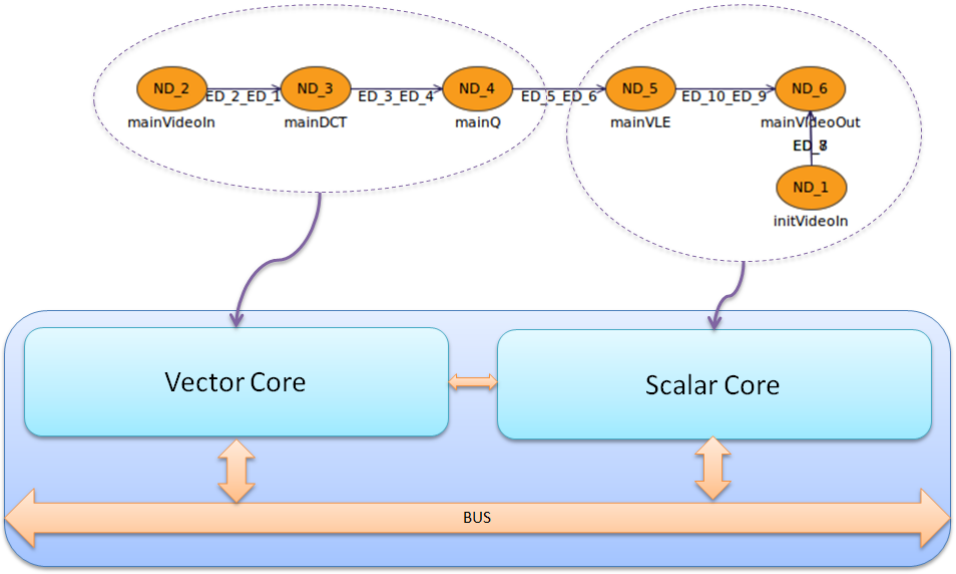


Figure 4.6: Cluster chosen to map the MJPEG application.

4.1.4 System Selection

As we now have a good idea of the kind of system that we need, we can choose a system to map the application (or we can create a new one). The system chosen was the MJPEG system that can be found in Appendix B. This system was created specifically for this example.

As analysis revealed that some of the clusters would benefit from vector processing, a vector ASIP, Core02, was constructed (Appendix A, Figure A.3). This core has 4 vector IS (Issue Slot), 2 Scalar IS and 1 Control IS. In order to access data simultaneously each one of the vector IS has one load/store FU (Functional Unit) connected a vector memory (4 vector memories in total). We have the data split on 4 different vector memories because each one of the components of the image (y1, y2, u, v) can be processed in parallel. This vector has a master interface in order to read/write data from the bus and 4 FIFOs that will be used to handle the communication with other cores (Core03 on this example).

Core03 (Appendix A, Figure A.2) is a scalar ASIP. This core has 5 scalar IS and 1 Control IS. It has 5 scalar memories for the same reason that Core02 has 4 vector memories. As Core02, this core has a master interface and FIFOS in order to communicate with the system.

It's important to observe that the cores and the system can be further optimized (FU, connections and Register could be removed). This hardware optimization process is the next step that should be realized after the mapping and it will be done later on the project.

4.2 Processor Architecture Features

Before discussing the mapping of the application on the system, it is important to realize what features the constructed ASIPs offer. There are five different kinds of features that should be taken into account:

- Operation-level Pipelining
- Multi-RISC Operations
- Instruction-level Parallelism
- Vector-, Data- or SIMD-level Parallelism
- Task-, Thread- or MIMD-level Parallelism

This section will not describe each one of these features because the focus here is only to highlight the features used to map the MJPEG encoder and point out what can still be optimized. We will try to illustrate these features with examples that were used when mapping the application. Further explanations of these features can be found in (LINDWER; DIKEN, 2011).

4.2.1 Operation-level Pipelining

Operations with complex silicon implementations (e.g. multipliers) may be executed in several stages, separated by pipeline registers. This generally results in better balancing of hardware resources and higher attainable overall clock speeds.

Looking at Core02 and Core03, we can see that both have the *std_mul* operation. This operation takes 2 cycles to finish as we can see on Figure 4.7. Modifications and analysis needed in order to introduce pipelines stages on the core will be done later on the project.

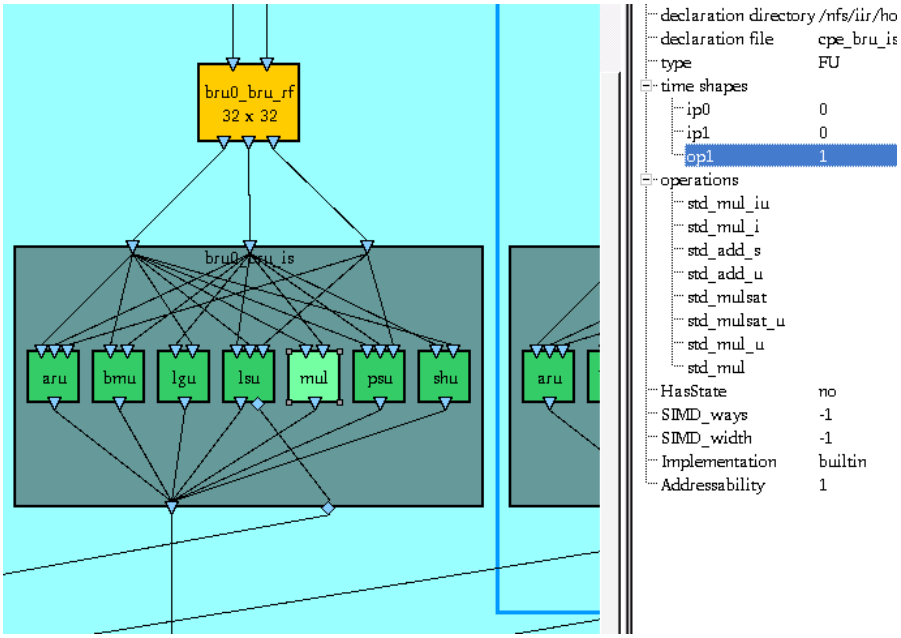


Figure 4.7: Example of FU with Operation-level Pipelining.

4.2.2 Multi-RISC Operations

These kinds of operations are often referred to as custom operations. When searching for ways to increase efficiency of application specific processing, processor designers may choose to collapse dataflow graphs which combine several relatively simple operations into a single, more complex processor operation. Since this may result in function units with complex/large hardware implementations, additional operation-level pipelining may be needed, as described above.

In order to use multi-RISC operations in the ASAM project, the application code needs to be transformed, explicitly introducing intrinsics. One example of a custom operation can be found at Figure 4.8. We can see the description of the *vec_avgrnd_cu* operation at Figure 4.9. If we want to use this operation in the code, we should invoke it like: *vec_out = OP_vec_avgrnd_cu (vec_in, scalar_in)*.

At the moment, we did not introduce new operations on the cores in order to optimize the MJPEG encoder execution. This process will be done later on the project.

4.2.3 Instruction-level Parallelism

Instruction-level Parallelism requires support through either VLIW or superscalar processors. Both types of processors contain multiple parallel instruction pipelines in which, during a single cycle, multiple operations can be started.

In the ASAM project we will be generating VLIW processors. In this case, the instructions themselves encode exactly which operations are being executed in parallel. In order to increase the amount of available parallelism certain code transformations need to be applied, such as loop unrolling.

4.2.4 Vector-, Data- or SIMD-level Parallelism

This kind of parallelism is comparable to multi-RISC operations, in the sense that each vector operation is equivalent to a multitude of non-vector operations. However, vector operations typically apply a single non-vector operation to a sequence of data items of the

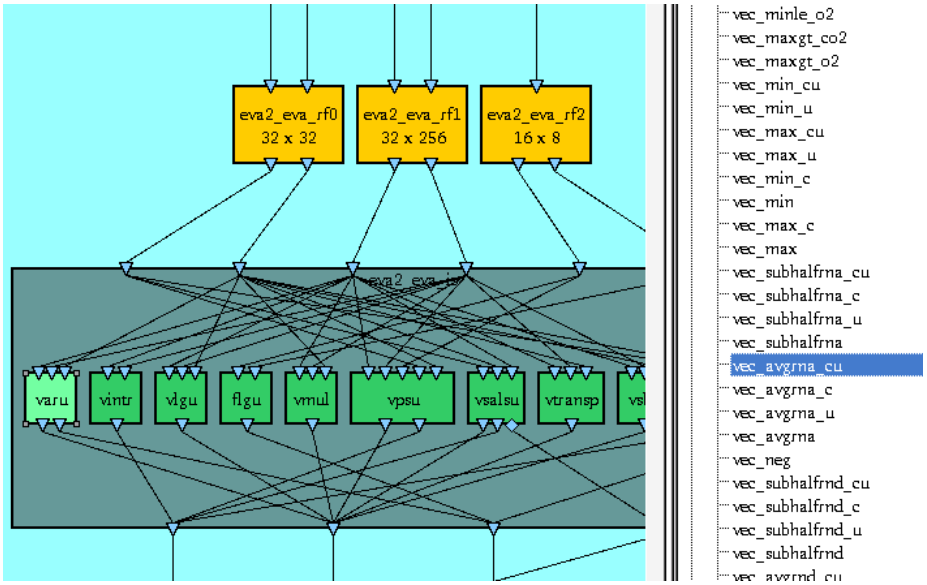


Figure 4.8: Example of a custom operation.

vec_avgrnd_cu - vector unsigned average with rounding to nearest even with cloned scalar

Description

This operation computes the average value between the unsigned values of the element i of vector $A and the scalar $B.
The rounding to nearest even is applied to the average result.
The scalar input $B is cast to the element width of the vector $A.

Syntax

vec_avgrnd_cu (<FU>, A, B, R);

R = OP_vec_avgrnd_cu (A, B);

Semantics

```
for i=0 to NWAY-1 do {
  RND((A[i]+B)/2)
}
```

Operands

Operand	*Direction*	*C type*	*Width (bits)*	*Cycle*
A	Input	__int256	256	0
B	Input	unsigned int	32	0
R	Output	__int256	256	0

Functional Units

The functional unit <FU> can be one of the following:

- eva0_eva_is_varu
- eva1_eva_is_varu
- eva2_eva_is_varu
- eva3_eva_is_varu

Figure 4.9: Description of the custom operation *vec_avgrnd_cu*.

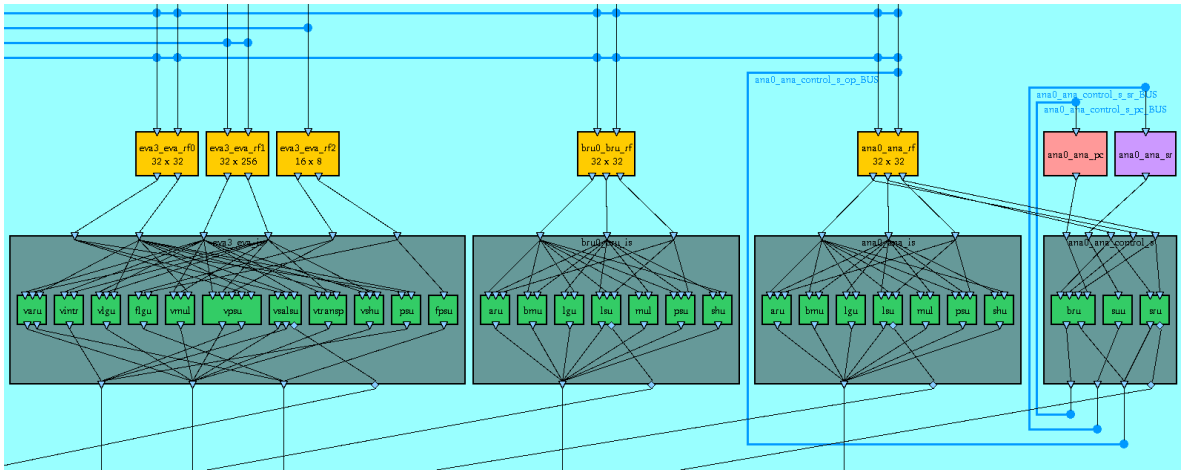


Figure 4.10: Example of a core with multiple IS and different kinds of data types.

same type.

ASAM ASIPs can mix-and-match vector and scalar processing. We can see in Figure 4.10 that we have an IS that combines function units that operate on scalar data types with function units that operate on vector data types. The scalar operations in that issue slot could serve e.g. for address calculation when loading and storing vectors. Also, this IS combines vectors of single-bit flags and vectors of data elements.

4.2.5 Task-, Thread- or MIMD-level Parallelism

Within ASAM multi-ASIP systems, task-, thread, and MIMD parallelism are the same: running multiple data plane tasks in parallel, on different ASIPs. In the MJPEG system, we map different parts of the code on Core02 and Core03 and run these parts in parallel. On the other hand, when mapping multiple KPN nodes on a single ASIP, they do not run as multiple tasks. The ASIPs do not run an OS which schedules the KPN nodes. The ASIP needs to be explicitly programmed to run the code for the KPN nodes as a single task, executing the different KPN nodes one-by-one.

4.3 Code Transformations and Mapping

By employing data flow analysis, compilers indeed do determine which operations can be executed in parallel, without affecting the defined functionality. However, this kind of analysis often is not enough to extract and drive all possible multi-ASIP parallelism. Next to ILP-style parallelism, SoCs with ASIPs support many other kinds of parallelism. Even though compilers can extract a certain level of VLIW-style parallelism, higher-level code transformations are needed to allow compilers to fully exploit all kinds of parallelism present in the application.

The kinds of code transformations that are identified within the ASAM project are the following: vectorization; compiler pragmas; task- and loop-level code transformations; abstractions. They determine the input formalism for compilers to map code on ASIPs and they express parallelism in ways that compilers can understand.

The objective of this section is not to describe each one of these code transformations. Instead, we will show the code transformations applied to the MJPEG encoder in order to map and to optimize the applications. Further explanation of these transformations can be found in (LINDWER; DIKEN, 2011), (BANERJEE, 1993) and (BANERJEE, 1994).

4.3.1 Vectorization

When using vectors, data may need to be organized in a specific manner, such that they can be loaded through vector load operations. Even more, loop- and task-level code transformations are often required to make efficient use of the vector capabilities of application-specific processors.

When showing the use vector operations in the consumer application, the modifications needed to include vector operations in the code will be discussed together with the associated code transformations (Section 4.3.3).

4.3.2 Compiler Pragmas

Compiler pragmas are a collection of additional constructs that drive compilers to perform certain optimizations or to make use of certain architectural features. For example, loop unrolling and software pipelining are often driven through compiler pragmas. Also compilers may offer different heuristics, in order to trade off speed, power, and register pressure. Such heuristics are also controlled by compiler pragmas. Compiler pragmas for the consumer application will be shown in conjunction with the associated code transformations (Section 4.3.3).

4.3.3 Task- and Loop-level Code Transformations

Generally, code transformations have three goals: code partitioning according to type of processing; reduction of dependencies, and improving regularity. Task-level loop transformations may be based on layering of the code, such as control layer and data plane layer. They may also result in merging of similar activities, in order to facilitate vectorization. Other task-level transformations result in code that can be scheduled more easily, such as rewriting in static single assignment form and inlining. Loop-level transformations are typically meant to reduce dependencies and improve regularity. For example, loop unrolling removes loop dependencies and may improve regularity, thereby increasing the compiler's possibilities to schedule parallel issue slots.

In the following subsections, the code transformations that have been applied in the MJPEG encoder will be illustrated.

4.3.3.1 Loop Normalization

Loop Normalization makes the dependence testing process as simple as possible and simplifies the application of many other loop transformations. In the function *PreshiftDctMatrix* for example the testing part of the "for" was modified as we can observe comparing Code 4.2 with Code 4.3.

4.3.3.2 Unroll-and-Jam

Unroll-and-Jam is a process of unrolling outer loops and fusing the new copies of the inner loops. It increases the size of the loop body and hence improves the available parallelism (e.g ILP). It can also improve the data locality. Moreover, the inner-most loop can be a candidate for vectorization.

We will use the function *PreshiftDctMatrix* as an example (Code 4.2 until Code 4.5). The first step needed is to normalize the loop as explained before. Code 4.3 shows the loop normalized. As *BLOCKSIZE* is equals to 64, we can divide it by 8 and create a new loop inside this loop that iterated 8 times. The reason for doing this is that now it is easy to see that a vector with 8 elements could be used to replace this new loop. It is

not mandatory to use 8 as the size of the vector. Eight was chosen here because it is the number of vector ways of Core02. Code 4.5 shows the new loop, now with vector.

```
void PreshiftDctMatrix(int *matrix, int shift){
  int *mptr;
  for(mptr=matrix;mptr<matrix+BLOCKSIZE;mptr++)
  {*mptr -= shift;}
}
```

Code 4.2: PreshiftDctMatrix initial code.

```
int mptrY1;
for( i = 0; i < BLOCKSIZE; i++){
  *mptrY1 -= DCTShift;
  mptrY1++;
}
```

Code 4.3: PreshiftDctMatrix normalized.

```
int mptrY1;
for( i = 0; i < BLOCKSIZE/8; i++){
  for( j = 0; j < 8; j++){
    *mptrY1 -= DCTShift;
    mptrY1++;
  }
}
```

Code 4.4: PreshiftDctMatrix with a refactored loop.

```
tvector *mptrY1;
for( i = 0; i < BLOCKSIZE/8; i++){
  *mptrY1 -= DCTShift;
  mptrY1++;
}
```

Code 4.5: PreshiftDctMatrix after vectorization.

4.3.3.3 Loop (Software) Pipelining and Loop Unrolling

Software pipelining is a technique which schedules an entire loop at a time to take full advantage of the parallelism across iterations. Basically, loop pipelining allows a new iteration of a loop to be started before the current iteration has finished. The compilers used in the ASAM project support automatic software pipelining, which can be enabled by pragma *#pragma asam pipelining*. Loop unrolling is a process of replication of the loop body, usually the innermost loop of a nest. It is supported in the project by pragma *#pragma asam unroll*.

Being Code 4.6 the initial code, Code 4.7 shows how to use pragmas in order to apply loop unrolling and pipelining on the ASAM project. It was decided to apply loop unrolling on the first loop because it iterates only eight times (that is a relative small amount of times). Unrolling completely the loop, we lose the time spent to control the loop, optimizing the code. However, it is important to keep in mind that unrolling the loop will increase the amount of code generated and we will need more memory to store it (which can use more area on the core and we would spend more time loading the program).

Software pipelining was used in the second loop because it iterates 64 times. Even if 64 is not so big, unroll would usemore memory, yet provide little improvement over pipelining. Again, these are trade-offs that need to be analyzed and they are linked with our initial constraints.

```
void ZigzagMatrix(int *imatrix, int *matrix){
  int omatrix[BLOCKSIZE];
  int i;
  int *tptr;
  for(tptr=zigzag_index;tptr<zigzag_index+BLOCKSIZE;tptr++) {
    omatrix[*tptr] = *(imatrix++);
  }
  for(i = 0; i < BLOCKSIZE; i++) {
    matrix[i] = omatrix[i];
  }
}
```

Code 4.6: ZigzagMatrix initial code.

```
for( i = 0; i < 8; i++) {
  for( j = 0; j < 8; j++) {
#pragma asam unroll
    omatrixy1[*pzig] = OP_vec_get (*mptry1, j);
    pzig++;
  }
  mptry1++;
}
for(i = 0; i < BLOCKSIZE; i++) {
  *pmainVLE_Y1 = omatrixy1[i];
  pmainVLE_Y1++;
#pragma asam pipelining=0
}
```

Code 4.7: ZigzagMatrix with unroll and software pipeline applied.

4.3.3.4 Loop Fusion / Combining / Jamming / Merging

Merging two or more loops into one nested loop enables to exploit the parallel execution (e.g. ILP) of the statements as long as no data dependencies exist between the merged statement, and loop bounds are identical. Moreover, loop merging can improve spatial and temporal data locality.

Code 4.8, 4.9 and 4.10 show the modifications applied in order to exploit the benefits of loop merging. In Code 4.8 we can observe that the function *PreshiftDCMatrix* is used with *Y1* and *Y2* and there is no dependency between the two function calls. Code 4.9 shows that it would be interesting to merge both functions because they are relatively simple, the loop bound is the same and both of them use *DCTShift*. So, if *DCTShift* is kept on register, it could optimize the execution time. The final result can be seen on Code 4.10.

```
void mainDCT(const TBlocks input, TBlocks *output)
{
  //Y1
  PreshiftDctMatrix((int*)&(input).Y1.pixel, DCTShift);
  IntArithDct((int*)&(input).Y1.pixel, (*output).Y1.pixel);
  BoundDctMatrix((*output).Y1.pixel, DCTBound);
  //Y2
```

```
  PreshiftDctMatrix((int*)&(input).Y2.pixel, DCTShift);
  IntArithDct((int*)&(input).Y2.pixel, (*output).Y2.pixel);
  BoundDctMatrix((*output).Y2.pixel, DCTBound);
  // ...
}
```

Code 4.8: MainDCT - initial code.

```
//================================
//PreshiftDctMatrix(Y1, DCTShift);
//================================
for( i = 0; i < BLOCKSIZE/8; i++){
#pragma asam unroll
  *mptrY1 -= DCTShift;
  mptrY1++;
}
//...

//================================
//PreshiftDctMatrix(Y2, DCTShift);
//================================
for( i = 0; i < BLOCKSIZE/8; i++){
#pragma asam unroll
  *mptrY2 -= DCTShift;
  mptrY2++;
}
```

Code 4.9: Body of PreshiftDctMatrix on the MainDCT code.

```
//================================
//PreshiftDctMatrix
//================================
for( i = 0; i < 8; i++){
#pragma asam unroll
  *mptrY1 -= DCTShift;
  mptrY1++;
  *mptrY2 -= DCTShift;
  mptrY2++;
}
```

Code 4.10: PreshiftDctMatrix for Y1 and Y2 merged on the code.

The results of all these modifications applied on the function *PreshiftDctMatrix* can be observed comparing Figure 4.11 (initial schedule) with Figure 4.12 (schedule after transformations on the code). In the first schedule it is easy to see that basically only one issue slot is really use to execute the body of the loop. As this loop will be executed at least 64 times for each frame, this can be considered a bad schedule. On other hand, Figure 4.12 shows a more optimal schedule. First, it is important to note that we need much less cycles to execute the body of the loop and, as we are using vectors, the loop will be executed eight times less. The total acceleration gained regarding the number of cycles needed to execute this loop can be calculated as follows:

- 8 times better due to the use of vectors;
- 2 times better attributable to loop merging;
- The total number of cycles is reduced from 455 (7 + 64*7) to 18 (2 +8*2) due to a better possibility of schedule (25 times faster).

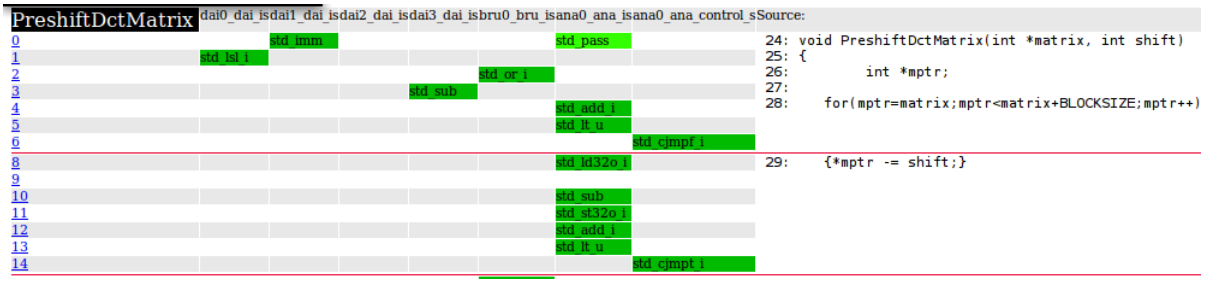


Figure 4.11: Initial schedule for PreshiftDctMatrix function on Core02.

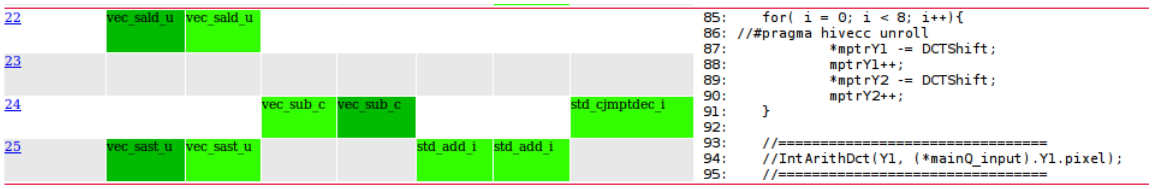


Figure 4.12: Schedule of PreshiftDctMatrix after vectorization and loop merging on Core04.

The optimization achieved here is a code that will need around 400 less cycles to be executed and it shows how important are the transformations introduced before.

In Figure 4.11, scalar issue slot "ana0_ana_is" is the one that is most heavily utilized. The actual loop body runs from instructions 8 to 14, i.e. 7 instructions generating a single scalar result. In Figure 4.12, as we are using Core02, the left-most four columns - that before were scalar issue slots (dai0...dai3) - were replaced by vector issue slots (eva0...eva3). In this case we see a much more even distribution, and we see the vector operations (prefixed with "vec_") within the vector issue slots being used. The loop body now spans 4 instructions, i.e. 22 to 25, generating 16 results per iteration.

4.3.3.5 Scalar Expansion with Vectorization

Scalar Expansion breaks the inter-loop dependencies by expanding a scalar into an array. Scalar expansion enhances the fine-grained parallelism by breaking dependencies and easing application of other transformations. Since a variable is expanded to an array, scalar expansion requires extra memory and more complex addressing.

Scalar expansion was used in the the function IntArithDct. Code 4.11 is the initial code of the function. We start our transformation here modifying the varable "s". "s" is used here as an accumulator and we convert "s" into an array of eight elements (Code 4.12). We've chosen eight due to the fact that we aim to use vectors in the future and eight is the number of elements in a vector on Core02. Looking at Code 4.12, now it's easy to see that we can apply vectorization in the most inner loop. In order to do it so, an important trick was used. As "s" was being used as an accumulator and as we knew that we could use the operation *OP_vec_isum* on the Core02, we have introduced the variable "acs1" to store the results of the accumulator in a vector. The final code can be found on Code 4.13.

```
void IntArithDct (int *block, int *outdata)
{
  int i,j,k;
  long tmp[64];
  for (i=0; i<8; i++) {
```

```
    for (j=0; j<8; j++) {
      long s = 0;
      for (k=0; k<8; k++) {
        s = s + (block[8*i+k])*(c[j][k]>>16);
      }
      tmp[8*i+j] = s >> 8;
    }
  }
  // ...
}
```

Code 4.11: IntArithDct - initial code.

```
void IntArithDct (int *block, int *outdata)
{
  int i,j,k;
  long tmp[8][8];
  for (i=0; i<8; i++) {
    for (j=0; j<8; j++) {
      long s[8];
      for (k=0; k<8; k++) {
#pragma asam unroll
        s[i] = (block[8*i+k])*(c[j][k]>>16);
      }
    tmp[i][j] = (s[0] + s[1] + s[2] + s[3] + s[4] + s[5] + s[6] + s
        [7]) >> 8;
    }
  }
  // ...
}
```

Code 4.12: IntArithDct after scalar expansion.

```
for ( i = 0; i < 8; i++ ) {
  tvector acs1;
  for ( j = 0; j < 8; j++ ) {
    tvector s = (vec_dct_c[i] >> 16) * ntmpy1[j];
    acs1 = OP_vec_set(acs1, j, OP_vec_isum(s));
  }
  temp[i] = acs1 >> 8;
}
```

Code 4.13: IntArithDct after scalar expansion and vectorization.

We can analyze the improvements reached applying scalar expansion with vectorization comparing Figure 4.13 (initial schedule on Core04) with Figure 4.14 (schedule after the transformations on Core02). The most important thing to be observed here is the usage of the vectors issue slots (the first four columns) on Core02. When looking at the schedule of the loop bodies (instructions 10-15;26-32 on Core04 and instructions 39-40;44-48 on Core02) we can see that the vector issue slots are almost full, which is very good. Even more, we can observe that fewer cycles are needed and the code is again more tightly scheduled.

4.3.3.6 Function Inlining

Generally, we use functions in order to increase maintainability and re-use of the code. However, by invoking functions, performance is reduced (relative to just flattening

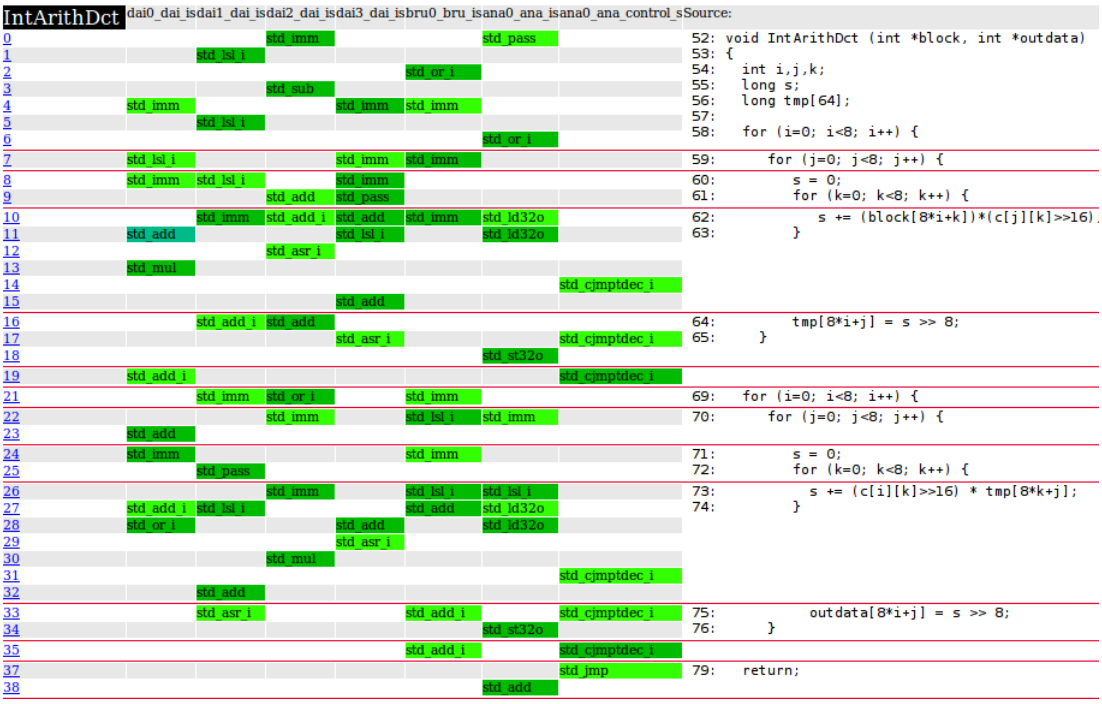


Figure 4.13: IntArithDct initial schedule on Core04.

Figure 4.14: IntArithDct after scalar expansion and vectorization scheduled on Core02.

the code and repeating the function bodies at the locations where they are called). Again, this is a trade-off that should be analyzed.

If a function is not called many times and if the body is relative small, it makes sense to inline this function. However, there are a lot of different cases that should be analyzed. One example in the MJPEG application is the function *initiVideoIn*. This function is called only one time for each frame and the body is small. Thus, we decide to apply inlining to this function.

4.3.3.7 Other Code Transformations

Not all the possible code transformations supported by the ASAM project were applied on the MJPEG application. The following transformations are examples of supported transformations that the flow could be able to analyze and to apply to the code when possible and beneficial for the application:

- Loop Fission / Distribution
- Loop Blocking / Tiling / Partitioning
- Strip-mining
- Loop Interchange / Permutation / Reordering
- Loop Reversal
- Loop Skewing / Bumping
- Scalar and Array Renaming
- Loop Unswitching
- Index-set Splitting
- Loop Peeling / Splitting

4.3.4 Abstractions

Abstractions are used to indicate a set of processes that can be fired in parallel. When using abstractions, two situations can happen:

- The code is written in ANSI-C without abstractions. In this case abstractions can be added to the code in order to introduce parallelism.
- The code already contains some abstractions. In this case the abstractions may have to be ported to the new target platform.

There are five kinds of abstractions that will be presented in the following sections, they are: intrinsics; data memory type and memory addressing; buffering and synchronization; vector operations; operations overloading.

4.3.4.1 *Intrinsics*

Intrinsics are direct representations of processor operations. They resemble inlining assembly code and they take regular variables as arguments. Operator overloading (discussed on Section 4.3.4.5) may be used to have certain intrinsics mapped onto operations symbols.

Intrinsics can be written in two different forms:

- **Direct form:** *<operation>(<func.unit>, Tin0 in0, Tin1 in1, ..., Tout0 out0, Tout1 out1, ...);*
- **Functional form:** *Tout OP_<operation>(Tin0 in0, Tin1 in1, ...);*

As we can see on Code 4.14, we make use of intrinsics to use operations like *vec_set* and *vec_get*.

```
for ( i =0 ; i < 8; i++ ) {
  for ( j = 0; j < 8; j++) {
    ntmpy1[i] = OP_vec_set(ntmpy1[i], j, OP_vec_get(tmpy1[j], i));
    // ...
  }
}
```

Code 4.14: Using intrinsics in the code.

4.3.4.2 *Data Memory Type and Memory Addressing*

ASAM's application-specific processors may employ different kinds of data memories:

- Scalar memories
- Vector memories
- Vector-addressable memories

ASIPs with multiple data memories typically also have multiple load/store units (LSUs), residing in different VLIW issue slots, in order to benefit from the combined bandwidth of the memories. Thus, it is important to distinguish which data is placed in which memory, because the data types must match. Additionally, if multiple memories of the same type are present in the ASIP, it is still important to explicitly identify which data resides in which memory.

The ASAM project makes use of attributes to designate locations of data items. The MEM() attribute is used to assign a variable to a particular named memory. If no memory attribute is used, the compiler will assume that the data item is to be located in the default memory of the ASIP. The AT() attribute is used to assign a specific address to a variable.

We can observe on Code 4.15 that MEM was used to distribute data over different memories. Even more, we make use of AT attribute to place the variables on the memory in the location that we want.

```
__extern int MEM(cec0_dmem_mem) AT(0x100) mainVLE_Y1[NBUFFER][BLOCKSIZE
    ];
__extern int MEM(cec1_dmem_mem) AT(0x100) mainVLE_Y2[NBUFFER][BLOCKSIZE
    ];
```

```
__extern int MEM(cec2_dmem_mem) AT(0x100) mainVLE_U1[NBUFFER][BLOCKSIZE
    ];
__extern int MEM(cec3_dmem_mem) AT(0x100) mainVLE_V1[NBUFFER][BLOCKSIZE
    ];
__extern int MEM(bru0_xmem_master) AT(0x2000) external_memory_img_out[
    NBUFFER][V_SIZE*4];
```

Code 4.15: Example of the use of MEM and AT attributes when declaring variables.

On an ASIP with function units that can process vectors of N signed elements of each E bits, the ASAM project defines that the ANSI-C data type for such vectors is:

- __spackedNxE: Arrays of vectors on an ASIP would thus be declared as follows:
  - *__spackedNxE MEM(vec_dmem) svar[10]; // array of vectors of signed*
  - *__upackedNxE MEM(vec_dmem) uvar[10]; // array of vectors of unsigned*

As we can see on Code 4.16, two new data types were defined (*tvector* and *tflags*).

```
typedef __packed8x32 tvector;
typedef __upacked8x1 tflags;
```

Code 4.16: Example of types used.

Code 4.17 is a good example of how to write code in order to optimize the memory access. In this code, each component of the image was split in a different memory. It was done because we know that they can be computed in parallel. Thus, if we allow simultaneous access to the input data, we should get better results.

```
// Y1
tvector MEM(eva0_vmem_mem) *mptry1;
tvector MEM(eva0_vmem_mem) *pQCoefy1;
int omatrixy1[BLOCKSIZE];
TPixel MEM(bru0_xmem_master) *pmainVLE_Y1;
// Y2
tvector MEM(eva1_vmem_mem) *mptry2;
tvector MEM(eva1_vmem_mem) *pQCoefy2;
int omatrixy2[BLOCKSIZE];
TPixel MEM(bru0_xmem_master) *pmainVLE_Y2;
// U1
tvector MEM(eva2_vmem_mem) *mptru1;
tvector MEM(eva2_vmem_mem) *pQCoefu1;
int omatrixu1[BLOCKSIZE];
TPixel MEM(bru0_xmem_master) *pmainVLE_U1;
// V1
tvector MEM(eva3_vmem_mem) *mptrv1;
tvector MEM(eva3_vmem_mem) *pQCoefv1;
int omatrixv1[BLOCKSIZE];
TPixel MEM(bru0_xmem_master) *pmainVLE_V1;
```

Code 4.17: Example of memory distribution on the code.

Figure 4.16 shows the improvement gained regarding the first schedule on Figure 4.15. Each column in the graph represents the use of one issue slot. Thus, more black spaces we find, worst is the utilization of the core. Figure 4.15 shows the initial code mapped on Core04. As we can observe, the bottleneck here came from the load/store that are realized on "ana0_ana_is". This is the issue slot connected to the memory where all data is placed (default memory).

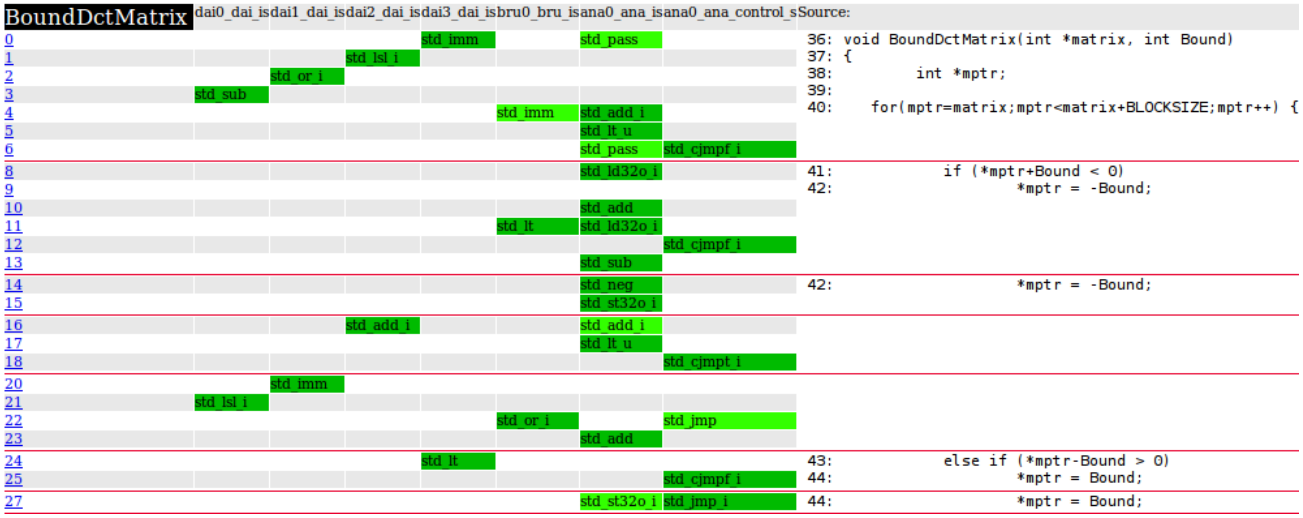


Figure 4.15: Scheduler of BoundDCMatrix before data distribution.

On other hand, Figure 4.16 shows the new schedule after splitting the data in four different memories. As we can see, the four vector issue slots are completely full, which means that we are making a good use of them. The blank spaces came from the control and scalar issue slot that are not needed in this part of the code.

Note that in below Figure 4.16, the load operations on all 4 vector memories take place in parallel in instruction #86 ("vec_sald_u") and the store operations all take place in instruction #96 ("vec_sast_u").

4.3.4.3 Buffering and Synchronization

When using multi-ASIP systems the communication between cores can represent a huge bottleneck in the application that needs to be analyzed. Typically, data is often organized in a way that input, output and intermediary data are located in physically different memories. In order to optimally make use of these distributed memories double-buffering needs to be applied.

As we intend to use double-buffer on ASAM, a default way to do the synchronization between cores was established. First, the inputs will be placed in the internal memory of the ASIP and the outputs should be accessed by a master interface connected to the bus. To synchronize, we will use blocking send and receive commands. The send and receive commands access FIFOs with limited depth. When sending a token into a full FIFO, the sending block is stalled, until the receiver reads a token. Similarly, when a block tries to receive a token from an empty FIFO, the receiver is blocked until the sender writes a token into the FIFO.

When scheduling code containing external buffer data and synchronization primitives, compilers act in a conservative way, assuming that these buffers contain volatile data. This means that compilers are severely restricted in the way in which they can exploit parallelism between input, output, and intermediary data processing. The ASAM project applies a more fine-grained way of indicating that certain specific operations need to be synchronized only with accesses to certain specific buffers. This leaves the compiler with maximum freedom to schedule all other operations which would otherwise have to be regarded as volatile or potentially impacting volatile data.

For this purpose, the *SYNC* and *SYNC_WITH* attributes are used. A buffer declaration

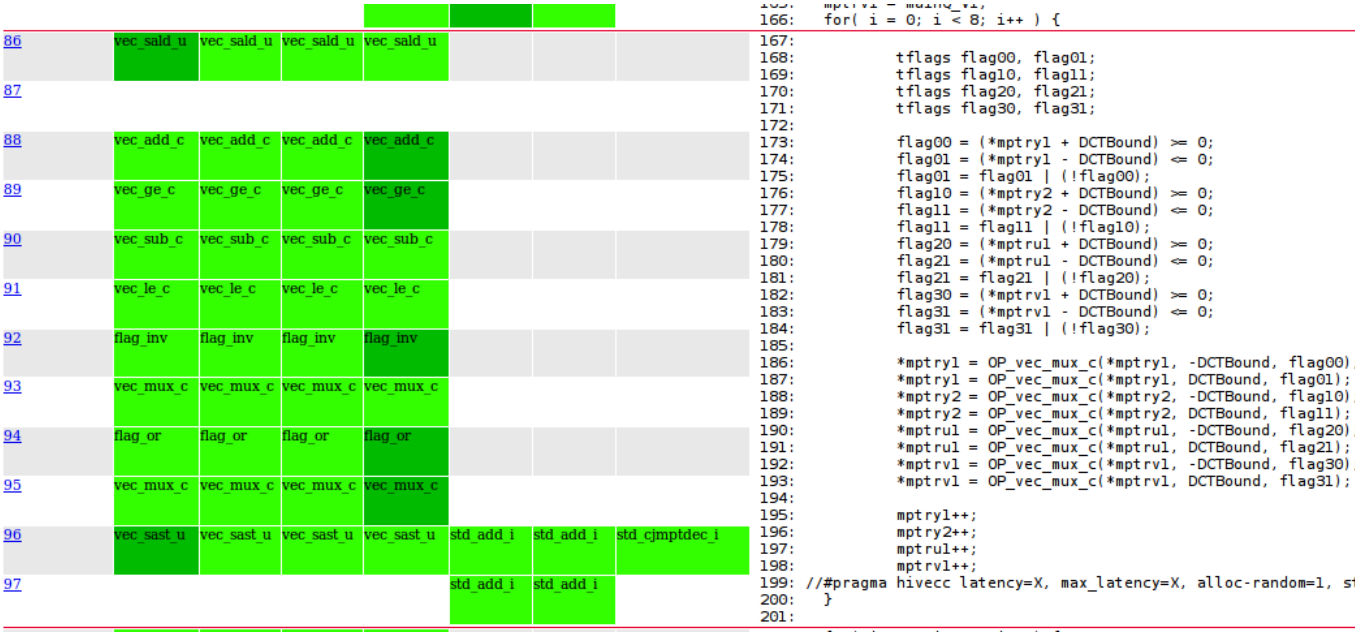


Figure 4.16: Scheduler of BoundDCMatrix after data distribution.

is extended with the *SYNC_WITH* attribute, as we can see on Code 4.18. This causes accesses to this buffer to not be regarded as volatile, but only needing to be synchronized with specific other operations.

```
#define VLEVIDEOOUT_INPUT0 0
#define VLEVIDEOOUT_OUTPUT0 1
SYNC_WITH(VLEVIDEOOUT_INPUT0) __extern int MEM(cec0_dmem_mem) AT(0x100)
    mainVLE_Y1[NBUFFER][BLOCKSIZE];
                __extern int MEM(cec1_dmem_mem) AT(0x100) mainVLE_Y2[
                    NBUFFER][BLOCKSIZE];
                __extern int MEM(cec2_dmem_mem) AT(0x100) mainVLE_U1[
                    NBUFFER][BLOCKSIZE];
                __extern int MEM(cec3_dmem_mem) AT(0x100) mainVLE_V1[
                    NBUFFER][BLOCKSIZE];
SYNC_WITH(VLEVIDEOOUT_OUTPUT0) __extern int MEM(bru0_xmem_master) AT(0
    x2000) external_memory_img_out[NBUFFER][V_SIZE*4];
```

Code 4.18: SYNC_WITH example.

Synchronization primitives (send and receive operations) typically have no data flow relation to the buffer which they guard. The *SYNC* attribute is to be regarded by the compiler as enforcing this relation, as we can see on Code 4.19. All of the send and receive operations will be scheduled such that only the ordering with respect to each other and loads from and stores to the variable buffer is maintained. Other operations are freely scheduled.

Simulating the original code, more than 70% of all cycles executed on the cores were stall cycles. After applying double-buffering on the code, we reduced the number of stall cycles by 54% and the total number of cycles was reduced by 37%.

```
void com_mainVLEmainVideoOut() ENTRY
{
  int i;
  THeaderInfo hi;
  initVideoIn(&hi);
```

```
  // request input
  std_snd(HIVE_COMM_FICTRL_mainVLEmainVideoOut_0,
      HIVE_COMM_FINPUT_mainVLEmainVideoOut_0, db_in) SYNC(
      VLEVIDEOOUT_INPUT0);
  modinc(db_in,NBUFFER);
  // wait permission to write output
  std_rcv(HIVE_COMM_FOCTRL_mainVLEmainVideoOut_0,
      HIVE_COMM_FOUTPUT_mainVLEmainVideoOut_0, db_out) SYNC(
      VLEVIDEOOUT_OUTPUT0);
  for (i = 0; i < N_BLOCKS; i++) {
    TPackets mainVideoOut_stream;
    // request input
    if( i < N_BLOCKS-1 ){
      std_snd(HIVE_COMM_FICTRL_mainVLEmainVideoOut_0,
          HIVE_COMM_FINPUT_mainVLEmainVideoOut_0, db_in) SYNC(
          VLEVIDEOOUT_INPUT0);
      modinc(db_in,NBUFFER);
    }
    // wait for input
    std_rcv(HIVE_COMM_FICTRL_mainVLEmainVideoOut_0,
        HIVE_COMM_FINPUT_mainVLEmainVideoOut_0, db_read) SYNC(
        VLEVIDEOOUT_INPUT0);
    mainVLE(mainVLE_Y1[db_read], mainVLE_Y2[db_read], mainVLE_U1[
        db_read], mainVLE_V1[db_read], &mainVideoOut_stream);
    mainVideoOut(hi, mainVideoOut_stream, external_memory_img_out[
        db_out]);
  }
  // output sent
  std_snd(HIVE_COMM_FOCTRL_mainVLEmainVideoOut_0,
      HIVE_COMM_FOUTPUT_mainVLEmainVideoOut_0, db_out) SYNC(
      VLEVIDEOOUT_OUTPUT0);
}
```

Code 4.19: SYNC example.

4.3.4.4 Vector Operations

As vectors have already been discussed before, we will only present some code introduced on the MJPEG application on this section. Generally, one can identify four different types of vector operations: Element-wise operations; Cloning operations; Intra-operations and Permutation Operations.

Element-wise operations combine all elements of the input vectors with vector elements at the same locations in the other input vectors. As we can see on Code 4.20, the *OP_vec_mux_c* combines **mptry1* that is a *tvector* with *flag00* that is a *tflag*.

Cloning operations are operations in which a vector input and a single scalar input are combined. We can see on Code 4.20 that *DCTBound* is a scalar that will be cloned into a vector to realize the mux operation. We can see that the same happens on Code 4.21, where the number 16 will be cloned before execute the mux. In this case, it's important observe that the shift operation here were overloaded to be used with vectors (Section 4.3.4.5).

```
for( i = 0; i < 8; i++ ) {
  // ...
  *mptry1 = OP_vec_mux_c(*mptry1, -DCTBound, flag00);
  *mptry1 = OP_vec_mux_c(*mptry1, DCTBound, flag01);
  // ...
```

```
}
```

Code 4.20: Element-wise and cloning operation.

Intra-operations combine all or several of the elements of vectors with other elements of the same vector. For example, the intra-sum operation (Code 4.21) adds all vector elements together, producing a scalar result.

```
for ( j = 0; j < 8; j++ ) {
#pragma asam unroll
  s0 = Y1[i] * (vec_dct_c[j]>>16);
  s2 = Y2[i] * (vec_dct_c[j]>>16);
  s4 = U1[i] * (vec_dct_c[j]>>16);
  s6 = V1[i] * (vec_dct_c[j]>>16);
  acs0 = OP_vec_set(acs0, j, OP_vec_isum (s0));
  acs2 = OP_vec_set(acs2, j, OP_vec_isum (s2));
  acs4 = OP_vec_set(acs4, j, OP_vec_isum (s4));
  acs6 = OP_vec_set(acs6, j, OP_vec_isum (s6));
}
```

Code 4.21: Intra-vector and cloning operations.

Permutation operations either shuffle vector elements within the same input vector or between multiple input vectors. Examples of these operations are pass, shuffle and slice. We do not have examples to show on the MJPEG code, but compiler usually use pass operations in order to transfer data from one register file to another.

4.3.4.5 Operation Overloading

As an extension to ANSI-C, compilers used in the ASAM project should support operation overloading. This means that it should be possible to specify that the same C operator may result in different processor operations, depending on the types of the arguments of the C operator.

Looking at Code 4.22 we can find out how to overload operators. In this example, the operator "+" is overloaded to be used with vectors or vectors and scalar together. Operation overload is good because allow us to have a "clear" code. In Code 4.23, for example, we make use of shift, division, addition, subtraction and comparison operations overloaded. This makes the code much clear than using intrinsics everywhere.

```
/* ================================
* Operator +
*  Non-saturated add has priority
* ================================ */
#if HAS_vec_add
HIVE_BINARY_OPERATOR (+,OP_vec_add,tvector,tvector,tvector)
HIVE_ASSIGN_OPERATOR (+,OP_vec_add,tvector,tvector,tvector)
#else/* HAS_vec_add */
#if HAS_vec_addsat
HIVE_BINARY_OPERATOR (+,OP_vec_addsat,tvector,tvector,tvector)
HIVE_ASSIGN_OPERATOR (+,OP_vec_addsat,tvector,tvector,tvector)
#endif/* HAS_vec_addsat */
#endif/* HAS_vec_add */
#if HAS_vec_add_c && HAS_vec_add_ic
HIVE_BINARY_AND_IMM_OPERATOR (+,OP_vec_add_c,OP_vec_add_ic,tvector,
    tvector,int)
HIVE_ASSIGN_AND_IMM_OPERATOR (+,OP_vec_add_c,OP_vec_add_ic,tvector,
    tvector,int)
```

```
#else /* HAS_vec_add_c && HAS_vec_add_ic */
#if HAS_vec_addsat_c && HAS_vec_addsat_ic
HIVE_BINARY_AND_IMM_OPERATOR (+,OP_vec_addsat_c,OP_vec_addsat_ic,
    tvector,tvector,int)
HIVE_ASSIGN_AND_IMM_OPERATOR (+,OP_vec_addsat_c,OP_vec_addsat_ic,
    tvector,tvector,int)
#else /* HAS_vec_addsat_c && HAS_addsat_add_ic */
#if HAS_vec_add_c
HIVE_BINARY_OPERATOR (+,OP_vec_add_c,tvector,tvector,int)
HIVE_ASSIGN_OPERATOR (+,OP_vec_add_c,tvector,tvector,int)
#else /* HAS_vec_add_c */
#if HAS_vec_addsat_c
HIVE_BINARY_OPERATOR (+,OP_vec_addsat_c,tvector,tvector,int)
HIVE_ASSIGN_OPERATOR (+,OP_vec_addsat_c,tvector,tvector,int)
#endif/* HAS_vec_addsat_c */
#endif/* HAS_vec_add_c */
```

Code 4.22: Overloading Operator "+".

```
for( i = 0; i < 8; i++ ) {
  tvector aux00, aux01;
  aux00 = (*mptry1 + ((*pQCoefy1) >> 1)) / (*pQCoefy1);
  aux01 = (*mptry1 - ((*pQCoefy1) >> 1)) / (*pQCoefy1);
  *mptry1 = OP_vec_mux( aux00, aux01, (*mptry1 > 0));
  pQCoefy1++;
  mptry1++;
}
```

Code 4.23: Example of code using operators overloaded.

5 FUTURE WORK

As described in the previous sections, there is still work to be done. The evaluation process is the focus of this project and all tools of all partners should be evaluated. This means that some time is required to first learn how to use the tools developed. After that, the integration between the tools should be evaluated to, finally, allow the final evaluation of the flow using the use-cases.

In order to evaluate the flow, we need to compare our results with results from other tools available nowadays. In this case, another part of this research will be designated to find good tools to compare with the ASAM project. More than that, it is required to learn how to use these tools and generate good results.

Finally, with all the information mentioned before, we would be able to point out the benefits of using the ASAM project and where the project should be improved in order to allow an efficient exploration of architectures and help the application mapping process.

REFERENCES

LINDWER, M.; DIKEN, E. **D5.1: Specification of architecture-driven code transformations**, ARTEMIS 2009 ASAM Funded Project, May 2011.

BUYUKKURT, B; GUO, Z; NAJJAR, W. A. **Compiler Optimization for Configurable Accelerators**, ODES'05, 2005, Carnegie Mellon U

KENNEDY,K; ALLEN, J. R. **Optimizing Compilers for Modern Architectures: A Dependence-Based Approach**, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001

BANERJEE, U. **Loop transformations for restructuring compilers - the foundations**, Kluwer 1993: I-XVIII, 1-305

BANERJEE, U. **Loop parallelization**, Kluwer 1994: I-XVIII, 1-174

LINDWER, M.; NOTARANGELO, G. **D1.3: Flow integration, verification and acceptance requirements**, ARTEMIS 2009 ASAM Funded Project, November 2012.

JOZWIAK, L; LINDWER, M; CORVINO, R; MELONI, P; MICCONI, L; MADSEN, J; DIKEN, E; GANGADHARAN, D; JORDANS, R; POMATA, S; POP, P; TUVERI, G; RAFFO, L. **ASAM: Automatic Architecture Synthesis and Application Mapping**, DSD 2012 - 15th Euromicro Conference on Digital System Design, pages 1-11, Cesme, Izmir, Turkey, 2012.

DIKEN, E; JORDANS, R; CORVINO, R; JOZWIAK, L. **Application Analysis Driven ASIP-based System Synthesis for ECG**, Embedded World Conference, pages 1-8, Germany, 2012.

QIAO, P; CORPORAAL, H; LINDWER, M. **A 0.964mW Digital Hearing Aid System**, DATE 2011 - Design, Automation & Test in Europe, pages 1-4, Grenoble, France, 2011.

DIKEN, E; JORDANS, R; CORVINO, R; JOZWIAK, L; LINDWER, M. **Automated architecture synthesis and application mapping for ASIP based adaptable MPSoCs**, ACACES 2011 - 7th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems, pages 135-138, Academia Press, Ghent, Belgium, Fiuggi, Italy, 2011.

JOZWIAK, L; LINDWER, M. **Issues and Challenges in Development of Massively-Parallel Heterogeneous MPSoCs Based on Adaptable ASIPs**, PDP 2011 - 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing, pages 483-487, 2011.

JOZWIAK, L; LINDWER, M. **Automatic Architecture Synthesis and Application Mapping for Application-specific Customizable MPSoCs**, HOPES 2010 - the First International Workshop on Hands-on Platforms and Tools for Model-based Engineering of Embedded Systems, in the scope of ECMFA 2010 - the Sixth European Conference on Modelling Foundations and Applications, pages 105-110, Paris, France, 2010.

SILICONHIVE. **Hivelogic Configurable Parallel Processing Platform**, SiliconHive, 2010.

ASAM. **ASAM Project website**. Available on: <http://www.asam-project.org>. Last access: February 2013.

APPENDIX A CORES

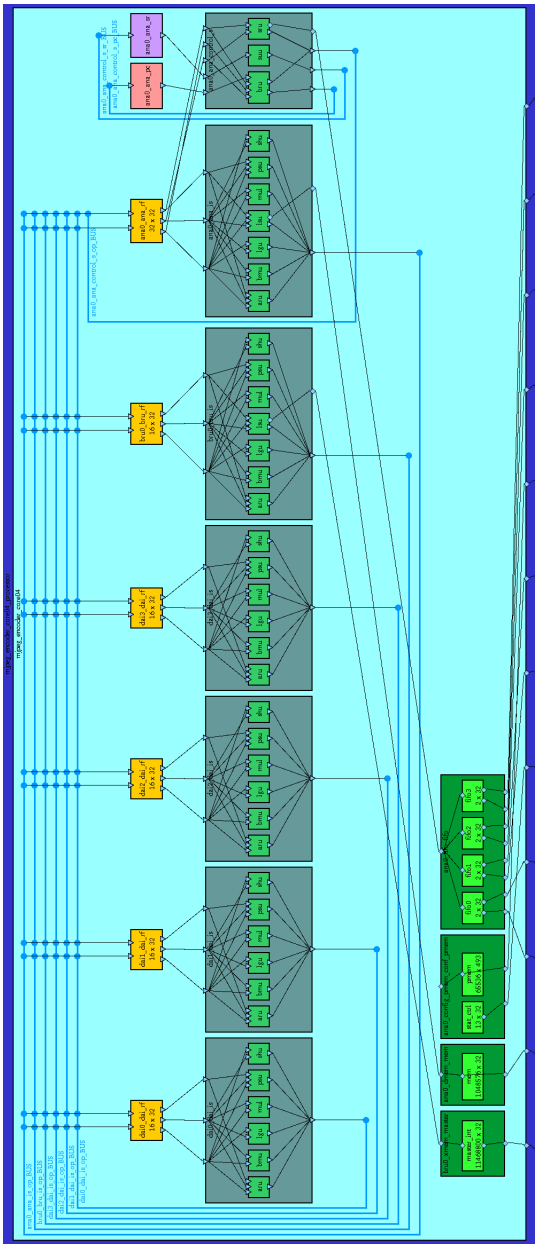


Figure A.1: Core04

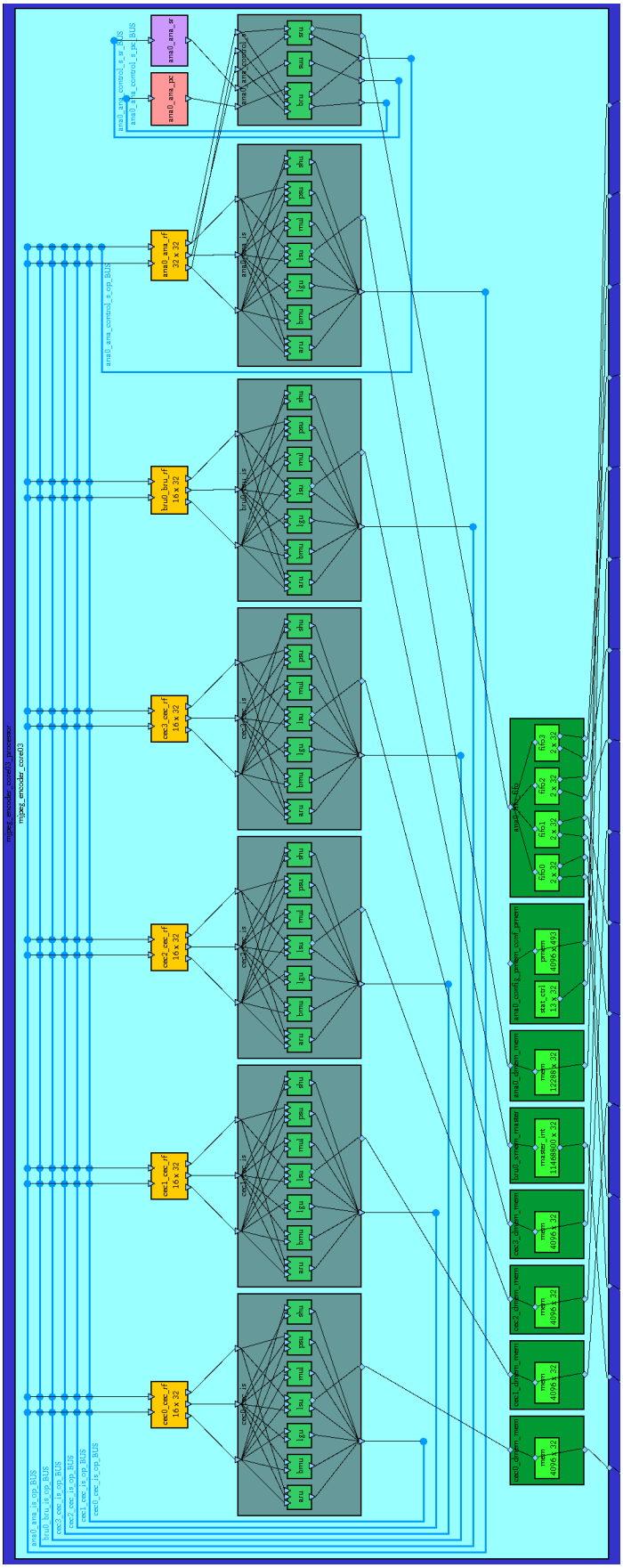


Figure A.2: Core03

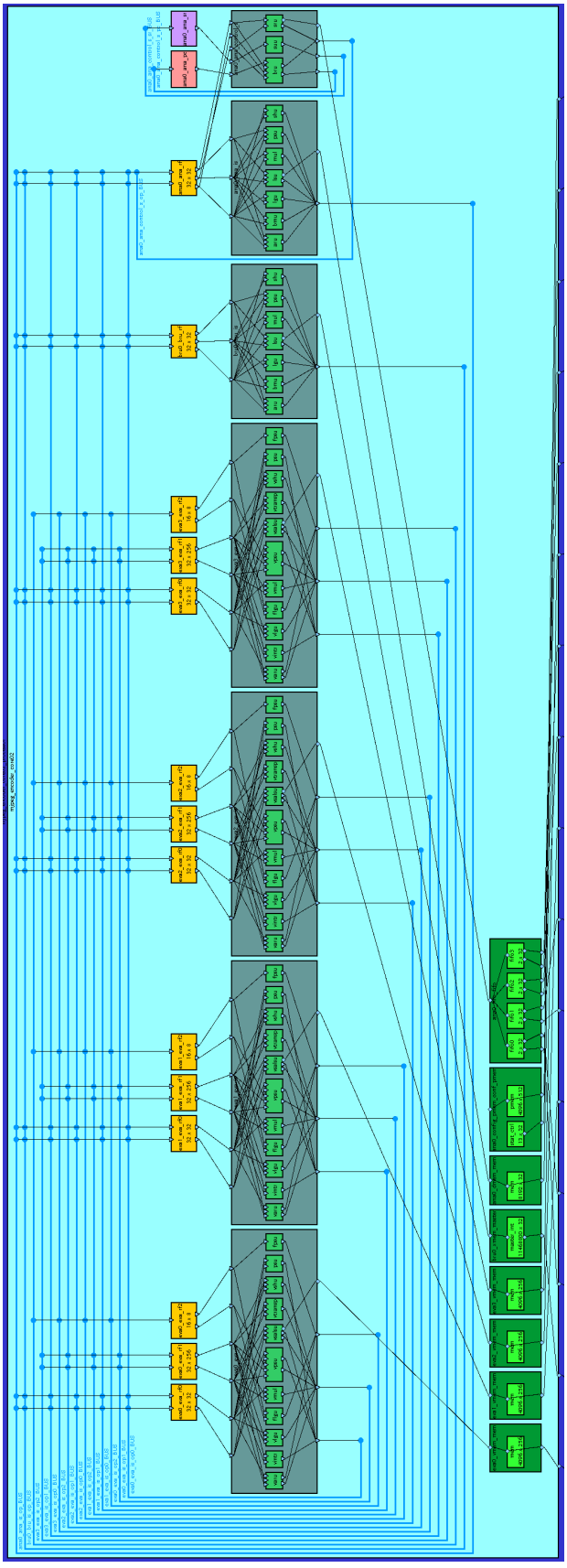


Figure A.3: Core02

APPENDIX B SYSTEMS

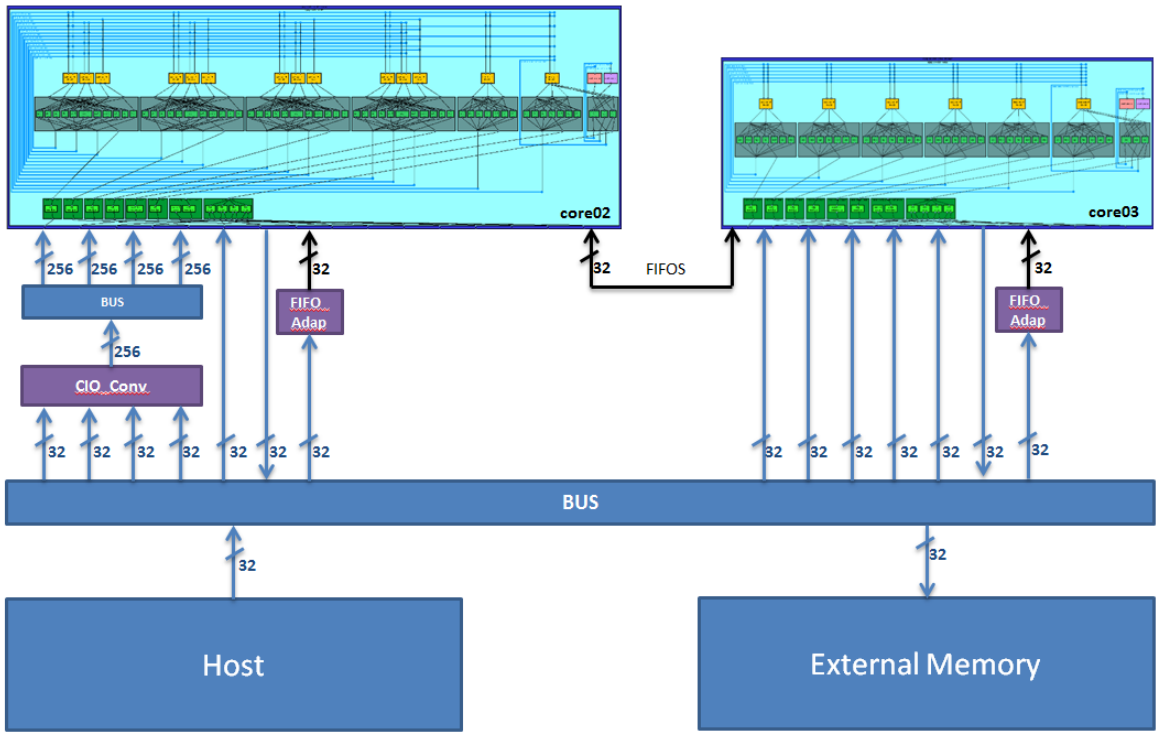


Figure B.1: MJPEG System

APPENDIX C METRICS

ECG Metric	Description
Power Consumption	The power consumption of the overall circuit is the most important result to be controlled in a low-power system. Both for a complex systems such as the REISC SoC and for simpler platforms, a power-optimized system will have to include power saving mechanism able to manage the right system configuration fit to the application to map on. Power measures should be reported, where possible, outlining the real configuration mode in order to give an add value to the result.
Area occupation	In medical markets, cost often comes second to safety, dependability and performance. However, Less area means, in general, lower power (both static and dynamic). So this metric is closely related to the power consumption metric. REISC SoC is a relatively large design both for ECG or other medical applications. An area reduction of about 20%, by utilizing the results of the project, should be feasible.
Wake up time	Is related the above power consumption result metric. If a low-power system includes a power saving mechanism, there needs to be a fast wake up mechanism able to bring the power-switched parts of the system up as soon as possible. The power-on latency depends both on the applied switching technology (pMOS/nMOS switches) and on the nature of the low-power solutions. REISC SoC achieves a wakeup time of 1.1 us: this is considered as a good result. If the latency were even lower, this would be very good. However, specific power switching features are not an objective of the ASAM project. Therefore, wake-up time is also not measured specifically.
Fault tolerance	A platform for medical applications has to be fault-tolerant. There is no simple measure for fault tolerance. The concepts include (i) robustness of the software, (ii) reliability of the technology, and (iii) redundancy of the system resources. However, most of these concepts are outside the scope of the ASAM project. Application software robustness is an aspect of the input specification. The project aims to be agnostic to the choice of middleware layers and to the choice of other implementation technologies. Redundancy is an architectural aspect and can, as a specification aspect/constraint be taken into account, at least in order to be able to compare other quality aspects based on comparable architectural constraints.
Real-time acquisition performance	Health monitoring only becomes viable once the system is able to acquire results in real-time. Real-time performance implies the capability, from the system, at least to capture data in the time range in which they are generated. This means an efficient streaming flow from the source to the buffer/memories where they will be stored waiting for the post-processing stage.

Table C.1: ECG Metrics. Source: (LINDWER; NOTARANGELO, 2012)

MPEG4 Metric	Description
Power Consumption	The overall circuit power consumption is the most important result to be controlled in multi-processor systems, if targeted for the mobile market. Power optimizations take place both at the circuit level and at the architecture level. Specific power saving mechanisms which manage system power configurations based on application monitoring fit are part of the first category, which is not a specific target of the ASAM project, since the ASAM project deals with architectural improvements which reduce overall system power (regardless of the state of the application). Power measurements should be reported, if possible taking into account the actual configuration mode, in order to give an added value to the result. Moreover image/video compression has important applications in specific low-power fields (i.e: gastrointestinal endoscopy). With some simplified algorithms (and of course at the cost of lower compression ratio), it is possible to attain much lower energy dissipation. For a multimedia application mapped on a network architecture, power consumption is also linked to the memory access rate and the interconnect infrastructure between the processing elements.
Area occupation	In mobile and consumer markets, cost is one of the most important metrics, which is directly related to area occupation. In addition, lower area means, in general, lower power (both static and dynamic).
Real-time acquisition performance	Real-time performance is truly a system capability, and is of the highest importance in any video capture or video display application. When encoding video, it means that, data needs at least to be captured in the time range which they are generated: this means an efficient streaming flow from the source to the buffer/memories where they will be stored waiting for the post-processing stage.
Bus usage	In systems though for multimedia applications, the bus subsystem is one of the bottlenecks that limits the performance of the overall system. The communication through the system bus has to be reduced to a minimum level in order to keep the bus complexity and cost low, while keeping performance high.
Latency	In video and audio-oriented applications, one of the main factors to be taken into account is the latency of the transmission of a given communication context. For this reason, this parameter has to be maintained at a low level or hidden from intelligent data access mechanisms (at least within the requirements).
Throughput	As for the latency case, the throughput of the overall system (how much information can the system transmit in a given time) is one of the key parameters that will have to be optimized.
Scalability	An architecture design targeted for multimedia algorithms should be highly scalable to support a rich range of applications, including them that require standard formats and higher performances. Scalability is very hard to certify and measure, yet can to some degree be expressed through the number of applications supported with at their proper performance levels. Scalability of ASIC designs is very poor while processor based design typically shows high scalability.
Performance / power	The best tradeoff between performance and power consumption implies combining the flexibility and scalability of microprocessors, while providing the performance/power ratio comparable to that of ASIC designs.
Interconnect structure and parameters of NoC or bus segmentation	This represents a critical design parameter for high-density platforms. The basic topologies of processor interconnection include bus/ring, crossbar and mesh. Bus/ring architecture is simple, but its throughput is quite limited. Especially, when the node number is large, the throughput is not enough to maintain performance. Crossbar can ensure maximum bandwidth, and its architecture is simple. However, its implementation cost (area and power dissipation) increases exponentially with the number of processor nodes. Thus, it typically not used for larger processor networks. Mesh (NoC) and segmented bus are reasonable trade-offs between bus and crossbar: they ensure scalability but their architectures are more complex.

Table C.2: MPEG4 Metrics. Source: (LINDWER; NOTARANGELO, 2012)

HA Metric	Description
Power Consumption	In the hearing aid market, the overall circuit power consumption, including analog parts, is the most important metric. A dynamic power saving mechanism is important. However, it is not a topic of the ASAM project. And since a hearing aid must be always switched on and since, under certain circumstances, it must be guaranteed that no sound is lost, a hearing aid cannot be fully powered down.
Area occupation	Currently, cost is not one of the most important metrics in the hearing aid market. Several types of hearing aids exist, some of which fit within the human ear. The aesthetic aspects of hearing aids dictate that they are very small. Therefore, the total chip area cannot exceed 5 mm2. Also, die size is often related to power consumption (both static and dynamic), the other main aspect of a hearing aid design.
Real-time acquisition performance	Real-time performance is truly a system capability, and is of the highest importance in a hearing aid application. When processing audio, it means that data needs to be captured and processed in the time range which they are generated; this means an efficient streaming flow from the microphones to speakers, through both analog and digital parts. The audio sampling rate has to be maintained under all circumstances.
Latency	In hearing aid applications, one of the main factors to be taken into account is the latency of audio processing. This latency should be very low, in order for the user to be able to respond immediately to sound impulses.
Throughput	As for the latency case, the throughput of the overall system (how much information can the system transmit in a given time) is one of the key parameters that will have to be optimized. Since the real-time capability needs to be maintained at all times, the throughput is determined by the complexity of the algorithms. On systems with high-performance audio processing capabilities, real-time processing can be guaranteed for complex algorithms, while maintaining high throughput.
Scalability	Scalability is an important feature for hearing aids, because the audio processing algorithms are constantly being updated. In addition, different users have different kinds of audio processing requirements. Therefore, audio DSPs have traditionally been applied extensively in hearing aids.
Performance / power	The best tradeoff between performance and power consumption implies combining the flexibility and scalability of microprocessors, while providing the performance/power ratio comparable to that of ASIC designs

Table C.3: Hearing Aid Metrics. Source: (LINDWER; NOTARANGELO, 2012)

APPENDIX D OPERATIONAL TARGETS (O)

1. Reduce productivity gap for designing ASIP-based embedded systems, by reducing effort levels for system design, relative to 2005, by 10%. In D1.2, this requirement was made more strict and more specific, through the specification of a set of usage scenarios:
   (a) Using the (as much as possible) automated flow: a novice team of engineers should achieve results that are on-par with 2010 state-of-the-art.
   (b) Using an automated flow, but possibly exercising some manual controls: an expert team of engineers should achieve results which are on-par with 2013 state-of-the-art, yet with 25% lower effort.
2. Or the ASAM results enable an expert team to manage an application complexity increase of 25% with similar effort levels as when using traditional flows.
3. Or the ASAM results enable solution efficiency improvement of 20% to 30% in terms of power, area, or performance (PPA). In D1.2, this requirement was made more specific:
   (a) Using manual controls, an expert team of engineers should exceed by 20% the 2013 state-of-the-art, at similar effort levels as when using traditional flows.

Source: (LINDWER; NOTARANGELO, 2012)

APPENDIX E FUNCTIONAL REQUIREMENTS (F)

1. Input is an embedded system specification with the following qualifications/components:
   (a) high-level,
   (b) algorithmic,
   (c) constraints (PPA numbers and implementation technology)
2. Output is an embedded system with the following qualifications/components:
   (a) correctly executing the algorithmic input specification,
   (b) heterogeneous,
   (c) ASIP-based,
   (d) satisfying the input constraints

Source: (LINDWER; NOTARANGELO, 2012)

APPENDIX F DETAILED TECHNICAL OBJECTIVES (T)

1. Technology-aware, rapid, multi-objective design space exploration, taking into account micro- and macro system trade-offs
2. Estimators will be used to drive fast design space exploration
3. Identify or construct a common input formalism for parametric requirement definitions for embedded systems
4. Identify or construct common system platform template formalism
5. Design space covers full range of parameters of multi-ASIP systems, including physical parameters
6. The impact of the floorplanning on the power consumption distribution on the die will be evaluated
7. Automatically construct and synthesize application-tailored system architecture
8. System and processor analysis for performance, area, and power consumption
9. Support purely hardware-implemented system modules
10. At micro-architecture level, synthesize ASIPs, accelerators, communication structures, and memory structures, by identifying, generating, and instantiating IP blocks, such as:
    (a) sequences of operations to be replaced by new instructions,
    (b) parallel issue slot clusters, register files, memories,
    (c) and other datapath elements
11. Synthesize power islands that allow the switch-off of parts of the processing and storage elements.
12. Identify or construct a common input formalism for application behavior, which is standard-based, near to industry practice, and have powerful (a. o. abstract) constructs to express parallelism
13. Support application analysis, parallelization, and partitioning, by analyzing how application domain specific computation and communication patterns interact in the interconnections at the runtime

14. (Semi-)automatic application mapping of embedded software, originating from a single source, on the resulting platform

Source: (LINDWER; NOTARANGELO, 2012)

APPENDIX G VERIFICATION REQUIREMENTS (V)

1. Design technology will be demonstrated and evaluated using standard and generated benchmarks
2. Demonstration will be based on (parts of) real-life industrial applications provided by the consortium's industrial partners.
3. Showing applications running in (near) real-time on actual implementation platforms (Because of practical and commercial constraints, the target system cannot have the same complexity as a complete product.)
4. Prove that the proposed new technology is capable of producing working systems, showing mapping both hardware and software systems to existing FPGA boards
5. resulting systems can be constructed with less effort or with higher quality (i.e. improved PPA) than when using traditional methodologies
   (a) SH and ST will measure how much effort currently is being spent on designing similar systems; methodology will be used to design an ultra low-power hearing aid platform. Area and power consumption should be 20% to 30% lower than state-of-the-art
   (b) Methodology will be used to design a high-performance video codec platform decoding HD streams within the power budget of a mobile phone.
6. The accuracy of results of estimators will be compared against the results of more traditional measurement systems2
7. The flow documentation will be checked on timely availability and quality
8. The prototype models will be checked for existence of their code, functionality, and quality of their execution
9. Demonstration on prototype tools of the applicability and effectiveness of the design methodology, flow, and tools.

Source: (LINDWER; NOTARANGELO, 2012)

APPENDIX H SYSTEM REQUIREMENT

ECG Requirement	Applicability to the ASAM Design Flow
SR-01 (relates to T3+T6+T12): Must be aligned with the pre-existent power format related to different abstraction levels.	The success and exploitation of new methods and tools depends on the ability to answer the design's needs that are not covered by existing design flows. A new architecture explorer and ASIP implementation tool only generates real interest from semiconductor companies, primarily through the possibility to be easily integrated in the existing industrial design flows. The ASAM flow should therefore respect and facilitate the RTL-to-layout flow. In particular, if ASAM is to generate power control mechanisms, there are already formats to express power-saving techniques in the design flow, like UPF (Unified Power Format) and CPF (Common Power Format). In that case, having the same format to represent power mechanism, shall be an objective, taking into account constructing a UPF/CPF representation in the RTL model of the SoC.
SR-02 (relates to T11): May allow representation of certain aspects of existing implementations of power control mechanisms.	Power mechanisms are already well exposed in D6.1 and D6.2 when describing the Reisc SoC architecture which includes the ECG application use case global definition and properties. The hardware representation of general industrial power mechanisms at all levels of abstraction (retention states, multi-supply voltage techniques, dynamic voltage and frequency scaling techniques, power domains, power gating clock gating etc.), may also be described at the macro and micro architecture level.
SR-03 (relates to T13+T14) Must provide an efficient SW representation of application blocks.	Since Silicon Hive's Software Development Kit (HiveCC) includes a full ANSI-C compiler, a well-defined methodology that allows mapping onto micro and macro-architecture models a pure functional C model is required.
SR-04 (relates to T9): Must allow the use of predefined HW blocks.	When dealing with generic platform exploitation and related design generator (as ASAM should be), at the interface between synthesis final steps of the implementation flow, it shall be possible to replace independent architecture nodes by already existing IP models within the resource library, even if modeled, for example, only in RTL.
SR-05 (relates to F2.1+T2+T8+V4+V6): Must provide accurate results, as specified in the application use cases.	The accuracy of characterization results is necessary to take the appropriate decisions in all usage scenarios (power dissipation for power management implementation, for example). The accuracy and reliability of the results have to be as expected in the requirements and have to allow the correct analysis of each application use case.
SR-06 (relates to F1.3+F2.4+T1): Must be able to take into account the characteristics of the technology.	When adding power/timing information to the HW generated block, all tools should be driven by generic parameters reflecting the characteristics of the technology (90nm, 65 nm, etc.).
SR-07 (relates to T7+T10+T12): Should use and follow open standards.	A platform generator fully compliant with open standard (i.e.: SystemC TLM 2.0) could be a plus good feature for industrial flow inheritability and should be considered a truly IP reuse follow up. The open standard compliance requires a heavy effort that should be evaluated in the next timeline review.
SR-08 (relates to V2+O1+O2+O3+T4+T5+T8+V5): Must provide a complete simulation analysis of the whole system.	ASAM environment should be able to support the simulation of communication engines among embedded parts of the system since this aspect is significant in the assessment of the performance of the design solutions and therefore in the design-space exploration and optimization.
SR-09 (relates to T6+T11): May provide characterization of power modes for DSE framework and runtime management.	Since (or if) the best solution of low power HW platform for ECG application will provide different power modes, exploration take into account power management strategies. Impact of software-driven power management strategies may be considered during power consumption measurement.
SR-10 (relates loosely to T4+T5+T10): Memory subsystem exploration phase should include on-chip and offchip memory models.	The memory hierarchy modeling and memory access profiling developed in design exploitation and implementation should consider dimensioning of the whole memory subsystem, both on-chip and off-chip. For a low power use-case definition (as ECG) the use and traffic exploitation with memory subsystem is a key point for power dissipation final results.
SR-11 (relates to SR-01, T3+T6+T12) Should provide standard interface implementation.	The interfaces between the flow and the application use case and between tools in the flow have to be as standard as possible. Using tools built to standards, which are non-proprietary, increases the ability to incorporate those tools into a diverse array of new or existing flows and minimizes the risk of 'data lock-in' in the future. For more details on the interface definition between tools refer to D1.2.
SR-12 (relates to O1+O2): Flow should have a proper simulation time.	In particular the system environment should have an order of magnitude gain of efficiency in term of simulation time respect to RTL level. This is crucial for Design Exploration step, where many simulations have to be performed.

SR-13 (loosely relates to O1+O2): Flow may be able to perform a RTL to a generic behavioral language transformation.	SR-04 requirement can lead to improper simulation time if RTL blocks are present. To avoid this, RTL blocks may be replaced, in an easy and low-effort way, by representations in a generic behavioral language
SR-14 (related to O1.1) Novice users should be able to achieve 2010 state-of-the-art results	Using ASAM's largely automated flow and starting from non-optimized code and system design, novice users should be able to achieve the same power and performance results as mentioned in ASAM deliverable D6.2, section 2.9.
SR-15 (related to O3.1) Expert users should be able to exceed 2010 state-of-the-art results by 20%.	Starting from 2010 state-of-the-art, expert users should be able to design a system that has 20% better PPA numbers (either power, performance, or area 20% improved) than mentioned in ASAM deliverable D6.2, section 2.9.

Table H.1: ECG Requirements. Source: (LINDWER; NOTARANGELO, 2012)

MPEG4 Requirement	Applicability to the ASAM Design Flow
SR-16 (loosely related to F1+F2+T12+V2): Must allow access to shared data and to the application code.	The application might need to access some shared constants or arrays of reference for example. As a consequence, it is important to design a way to access shared data from the software side. Accesses may be implicit or explicit, but a clear method to access data located in memory from the software side should be defined.
SR-17 (loosely related to T1): Should allow access to data located in the heap or stack.	If code integrated in a module is definitely intended to be implemented in CPU software (as opposed to ASIP software), we would have the possibility to use dynamic memory allocation as in pure software. We need to control the growth of the stack and heap and spread data in different memory sections when implemented on the final device.
SR-18 (loosely related to F1+F2+T12) Should provide a methodology to ensure ANSI-C compliance.	Application developers usually develop applications in pure ANSI C. Because, even if optional, some specific ANSI-C extensions need to be used to achieve code which is optimally targeted to the platform (vector operations, HiveCC pragmas, etc., see D1.2 and D5.1).
SR-19 (loosely related to F1+F2+T12): Should use standard C tools.	To create a C specification that can be handled by the ASAM framework tools, designers will most probably have to rely on specific libraries. However, libraries developed in the scope of the project should not restrict the use of commercial ANSI-C tools.
SR-20 (loosely related to T12): Should use last ANSI-C compiler (gcc) version.	The majority of the tools of the market offer the possibility to handle designs based on gcc kernel latest version. This makes ASAM developers to be able to use the latest debugging facilities.
SR-21 (related to T12+T13): Must provide a way to express parallelism in the application SW.	This will be directly offered by ASAM tools, but we need to be aware of any restriction on the number of potential parallel threads. We also need to know if the number of parallel threads would have any impact on the efficiency of the timing annotation tool.
SR-22 (related to F2.3+F2.4+T2+T8+T13): Should provide a clear analysis during HW/SW task separation.	The entry of the MPEG4 encoder application is composed of several signal processing algorithms that might be mapped onto hardware accelerators in order to reduce the load on CPUs and ASIPs. The 'HW/SW task separation' step of ASAM framework should provide an analysis of the potential implementations: hardware or ASIP software or CPU software. This analysis should integrate precise metrics such as timing performance and power consumption.
SR-23 (related to F2.2+T7+T10): May integrate automatic HW generation.	Studying two implementation variants by developing all components from scratch would be too expensive. As a consequence, the methodology proposed may integrate a synthesizer to implement software elements in hardware. The intervention of the designer should be reduced as much as possible. This is not part of the ASAM project proposal.
SR-24 (related to F2.2+F2.3+T10+T13+T14+V2): Must provide communication management.	The implementation of a task into hardware has to integrate the management of communications. Indeed, exchange of data between hardware and software elements has to respect the communication protocol defined by the hardware platform. The separation of hardware and software tasks should propose a way to handle communications efficiently by proposing for example an API of functions that can be used by the designer.
SR-25 (loosely related to T11+T13+T14): May support control of power management by software.	Since the evaluation study platform of the MPEG4 encoder application provides different power (see D6.2) contribution visibility, exploration may focus on power management strategies. Implementation of SW power management would in that case be a requirement.
SR-26 (loosely related to O1+O2): Should change the structure of code as little as possible.	Generated/transformed code shall resemble the original as much as possible. The overall structure (top-level functions, global variables, modules of functions etc.) should be preserved and should be recognizable in output code.
SR-27 (related to F1.3) Must allow using pre-characterized technologies.	When adding power information to the HW generated block, it is important that the tools are driven by the characteristics of the hardware technology (ASIC process node, FPGA, etc.).
SR-28 (related to O1+O2+F1.1+T14): Should provide tracking of code transformations.	When it is not possible to preserve the original structure of the code, as required in SR-26, and transformations are needed, the generated code should give information to the developer on the structure and how the original structure has been modified. It should be, for example, possible to track the input and output of functions.
SR-29 (related to O1+O2+T2+T7): System generator should be autonomous.	One advantage of ASAM platforms is the ability to use them without any cost expensive hardware: evaluation board, development board, JTAG connectors, etc. To favor exploitation, the generated virtual platform should be able to run in an autonomous way. It should be possible to run the simulation on a PC without attached specialized hardware components.
SR-30 (related to O1+O2+T8+V3): Virtual system generator should guarantee simulation performance.	The performance of the simulation should be sufficient to offer the possibility to debug applications. Simulation time should be efficient enough to allow many simulations in reasonable time.

SR-31 (related to F2.1+F2.4+T2+T8+V3+V4+V6): Should guarantee an adequate level of information.	All quality-related measurements, such as power, area, and performance will be estimated with different trade-offs between accuracy and analysis/simulation speed, as needed for a particular stage of the ASAM flow. The project is mostly concerned with power reduction through architectural features of the ASIPs, their constituent components and other architectural blocks, than with explicit power management, which is not directly in the scope of the project. The ASAM framework will analyze the implications of power management, by assuming that certain power control features are in place and adjusting power estimates accordingly. The most detailed quality-measurements will be sufficiently accurate such that users can determine whether the results of the ASAM flow satisfy the usage scenarios for this application use case.
SR-32 (related to F1.3+F2.4+T5+V6): Should show parameter's effects.	It is also important for the user to observe directly the impact of the modification of one parameter. As a result of changing a parameter, quality related measurements, such as power, area, and performance can be estimated with different trade-offs between accuracy and analysis/simulation speed, as needed for a particular stage of the ASAM flow.
SR-33 (related to O1.1) Novice users should be able to achieve 2010 state-of-the-art results.	Using ASAM's largely automated flow and starting from non-optimized code and system design, novice users should be able to achieve the same power and performance results as mentioned in ASAM deliverable D6.2, section 3.2.
SR-34 (related to O3.1) Expert users should be able to exceed 2010 state-of-the-art results by 20%.	Starting from 2010 state-of-the-art, expert users should be able to design a system that has 20% better PPA numbers (either power, performance, or area 20% improved) than mentioned in ASAM deliverable D6.2, section 3.2.

Table H.2: MPEG4 Requirements. Source: (LINDWER; NOTARANGELO, 2012)

HA Requirement	Applicability to the ASAM Design Flow
SR-35 (related to O1.1+O1.2+T12+T13): Must process straight-line ANSI-C code.	The application code is provided as regular non-optimized ANSI-C code. The application analysis phase of the project must autonomously partition such code, according to different processing characteristics found in the code.
SR-36 (related to T12): Should use last ANSIC compiler (gcc) version.	The majority of the tools in the market offer the possibility to handle designs based on Gnu Compiler Collection (GCC). The ASAM flow should incorporate functional verification of application code, based on compiling (and executing) applications using the latest GCC version. This gives the ASAM developers the possibility to use the latest debugging facilities for initial straight-line ANSI-C development.
SR-37 (related to F2.2+F2.3+T2+T8+T13+T14): Must provide a clear analysis to drive optimization strategies for code partitions	The hearing aid application is composed of several signal processing algorithms that might need to be mapped onto different types of accelerators or other architectural components, in order to improve processing efficiency. The macro-level design space exploration step of ASAM framework should provide an analysis of the potential implementations: hardware or ASIP software or CPU software.
SR-38 (related to O1.2): Should process partially optimized ANSI-C code and suggest further improvements.	The application code is also provided as optimized ANSI-C code, expressing different kinds of parallelism and mapped onto an optimized platform. The ASAM flow should be able to process this code and, if opportunities exist to further optimize this code, the flow should identify these opportunities or indicate to the experienced user that these opportunities might exist. The flow should also assist the experienced user in evaluating possible further optimizations.
SR-39 (related to O1.1+O1.2+O2+F2.2+F2.3+T4+T7) Must automatically generate an appropriate macro architecture design.	In order to map partitioned applications, the ASAM flow must generate those micro architectural components that are suitable to map the different partitions on. The ASAM flow must generate a macro architecture which instantiates those components.
SR-40 (related to O1+O3+F2.2+F2.3+T1+T5+T8+T10): Must analyze and potentially optimize architectural components.	The ASAM flow must provide analyses about the performance, power, and area characteristics of the different micro architectural components, when they execute their assigned application partitions. These application partitions may originate from initial straight-line code or from partially-optimized code. These analyses must be sufficiently accurate to allow comparison of different possible micro architectural optimizations and to analyze the effects of such optimizations.
SR-41 (related to O1.2+O3+T4+T7+T8): Must analyze and potentially optimize macro architecture.	The ASAM flow must provide analyses about the performance, power, and area characteristics of macro architectures, which instantiate (optimized) micro architectural components. These analyses must be sufficiently accurate to allow comparison of different possible macro architectural optimizations and to analyze the effects of those optimizations.
SR-42 (related to T4+T5+T7+T9): Macro-level analyses should take into account the characteristics of 'external' components.	In the hearing aid application use case, piezoelectric devices (microphones, speakers) and AD/DA converters consume a certain amount of system-level area and power. These devices are not generated by the purely digital ASAM flow. However, the area and power impact of these devices does partially depend on the processing characteristics of the application (e.g. sample rate). Even though the ASAM flow cannot influence the characteristics of these devices, when analyzing overall system-level PPA characteristics, the influence of such components on maximum achievable parallelism should be taken into account (Amdahl's law)

SR-43 (related to F1.3+F2.4+T1+T3+T5+T6): Should support power-aware technology choices and strategies synthesis	Hearing aids are always on devices. During operation, the device continuously processes the flow of incoming sound. Therefore, the choice of silicon implementation technology should result in the lowest leakage current possible. In order to reduce dynamic power consumption, the flow should use implementation technology and synthesis strategies which make use of very low voltages, low clock speeds, and specific (clock) duty cycles. On the other hand, the current optimized application does not make use of dynamic power control mechanisms and for this application. This is also not expected of the ASAM flow.
SR-44 (related to F2.2+F2.3+T10+T13+T14+V2): Must provide communication management between different architectural components.	Even when the complete hearing aid application is mapped onto a single ASIP, the flow will have to model the communication behavior between the 'external' components and the ASIP. Indeed, exchange of data between such 'external' hardware and software elements has to respect specific communication protocols. When the application is mapped onto several ASIPs, the communication between these ASIPs has to be facilitated as well. The separation of hardware and software tasks should propose a way to handle communications efficiently by proposing for example an API of functions that is automatically used to facilitate different application partitions and that can also be used by the designer to implement communication with 'external' components.
SR-45 (loosely related to O1+O2): Should change the structure of code as little as possible.	Generated/transformed code shall resemble the original as much as possible. The overall structure (top-level functions, global variables, modules of functions etc.) should be preserved and should be recognizable in output code.
SR-46 (related to F1.3) Must allow using pre-characterized technologies.	When adding power information to the HW generated block, it is important that the tools are driven by the characteristics of the hardware technology (ASIC process node, FPGA, etc.).
SR-47 (related to O1+O2+F1.1+T14): Should provide tracking of code transformations.	When it is not possible to preserve the original structure of the code, as required in SR-45, and transformations are needed, the generated code should give information to the developer on the structure and how the original structure has been modified. It should be, for example, possible to track the input and output of functions.
SR-48 (related to O1+O2+T2+T7): System generator should be autonomous.	One advantage of ASAM platforms is the ability to use them without any cost expensive hardware: evaluation board, development board, JTAG connectors, etc. To favor exploitation, the generated virtual platform should be able to run in an autonomous way. It should be possible to run the simulation on a PC without attached specialized hardware components.
SR-49 (related to O1+O2+T8+V3): Virtual system generator should guarantee simulation performance.	The performance of the simulation should be sufficient to offer the possibility to debug applications. Simulation time should be efficient enough to allow several seconds of real-time operations to be simulated in reasonable time, i.e. at most one hour of simulation time.
SR-50 (related to F2.1+F2.4+T2+T8+V3+V4+V6): Should guarantee an adequate level of information.	All quality-related measurements, such as power, area, and performance will be estimated with different trade-offs between accuracy and analysis/simulation speed, as needed for a particular stage of the ASAM flow. Since the project is mainly concerned with power reduction through architectural features of the ASIPs, their constituent components, and other architectural blocks, explicit power management is not directly in the scope of the project. The ASAM framework will analyze the implications of power management, by assuming that certain power control features are in place and adjusting power estimates accordingly. The most detailed quality-measurements will sufficiently accurate such that users can determine whether the results of the ASAM flow satisfy the usage scenarios for this application use case. The flow will automatically improve the performance of the system in terms of power without affecting its performance. The optimizations should be detailed to the user by printing the different power reduction techniques implemented and the optimization of the software code.
SR-51 (related to F1.3+F2.4+T5+V6): Should show parameter's effects.	It is also important for the user to observe directly the impact of the modification of one parameter. As a result of changing a parameter, quality-related measurements, such as power, area, and performance can be estimated with different trade-offs between accuracy and analysis/simulation speed, as needed for a particular stage of the ASAM flow.
SR-52 (related to O1.1+F2.4+V5+V6): When used by novice users, effort should be 20% less than using traditional methods.	ASAM Deliverable 6.2 reports that effort level for a single novice user to achieve a 2010 state-of-the-art design and mapping was 2 person/months. Using the new ASAM flow, this should be reduced to 7 man/weeks. Refer to D6.2 for PPA numbers of 2010 state-of-the-art.
SR-53 (related to O1.2+O3.1+F2.4+V5+V6) When used by expert users, 20% additional complexity should be managed.	ASAM Deliverable 6.2 requires that it should take expert users 2 man/months to extend the optimized version of the application with a feature such as Acoustic Scene Classification or with sampling frequency increased to 20 KHz. The ASAM flow should construct a new optimized application and system design to include this feature, yet still fitting the same PPA budget as 2010 state-of-the-art.

Table H.3: Hearing Aid Requirements. Source: (LINDWER; NOTARANGELO, 2012)