Optimizing Dataflow Programs for Hardware Synthesis

THÈSE N^{O} 6059 (2013)

PRÉSENTÉE LE 23 JANVIER 2014
À LA FACULTÉ DES SCIENCES ET TECHNIQUES DE L'INGÉNIEUR
GROUPE SCI STI MM
PROGRAMME DOCTORAL EN MICROSYSTÈMES ET MICROÉLECTRONIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Ab Al Hadi Bin AB RAHMAN

acceptée sur proposition du jury:

Dr G. Boero, président du jury
Dr M. Mattavelli, directeur de thèse
Prof. A. Prihozhy, rapporteur
Dr M. Raulet, rapporteur
Dr A. Schmid, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE
Suisse
2014

For my parents,
my wife, and our two daughters.

Acknowledgements

I would like to express my gratitude and thanks to all individuals that have made this thesis possible. First and foremost is to my thesis director, Dr. Marco Mattavelli for giving me the opportunity to work on this very exciting research project, and also for all the advice and encouragement he has given me throughout the years. Next, I would like to thank these people who have contributed directly to the work in this thesis: Prof. Anatoly Prihozhy with the formulation of the pipeline synthesis and optimization tool and technique, Dr. Ihab Amer and Hossam Amer on the work on dataflow program refactoring, Dr. Christophe Lucarz on the first dataflow program analysis tool, Dr. Richard Thavot on the work to optimize buffer size in dataflow programs, and the hardware implementation of the MPEG-4 decoder on an FPGA evaluation board, Damien De Saint Jorre with the design of the MPEG-4 AVC/H.264 decoder, and fellow Ph.D students Endri Bezati and Simone Casale Brunet with their respective synthesis and analysis tools Xronos and TURNUS, and the frequently lengthy discussions on several design and implementation issues with dataflow programming. I would also like to thank other lab members and scientists who have contributed indirectly to the work in this thesis: Dr. Jörn Janneck, Dr. Ghislain Roquier, Dr. Junaid Ahmad, Dr. Romuald Mosqueron, and Pascal Faure.

This work will not have been possible without the necessary financial support. For this, I would like to thank the Malaysian Ministry of Higher Education and the Univesiti Teknologi Malaysia, along with the arrangements with EPFL for the funds throughout the duration of the work.

Last but not least, I want to thank my parents, Ab Rahman and Zaiton, my wife Zue, and my two young daughters, Zahra and Dania, for their continuous love, dedication, and support.

Ab Al-Hadi Bin Ab Rahman
Lausanne, October 15th 2013

Abstract

Digital signal processing (DSP) systems can be represented by a Dataflow Process Network (DPN), which is a model of computation (MoC) based on a directed graph where a number of concurrent processes communicate through unidirectional FIFO channels. One of the formal programming languages that directly captures this MoC is the CAL dataflow language. It features among others, platform-agnostic specifications and an actor-oriented approach in systems design that makes it suitable for heterogeneous implementation of data-intensive DSP applications as compared to classical and other design methodologies. In this thesis, a new systems design methodology based on CAL is presented and validated for a complete design flow from specification to implementation, with high-level synthesis and analysis. The methodology is extended to rapidly explore design alternatives at high-level by the application of several novel optimization techniques on critical parts of complex CAL designs for hardware synthesis. The first optimization technique is by dataflow program refactoring with an objective of minimizing system latency with data and task parallelism, and with memory access reductions. The second optimization technique is refactoring by exploiting pipeline parallelism with an objective of maximizing operating frequency and minimizing resource. The third and final optimization technique is by minimizing and optimizing the FIFO buffer interconnection sizes using two distinct approaches on the implementation and the dataflow program level. As case studies, the efficacy of the optimization techniques are proven using the standardized CAL specification of the MPEG-4 Part 2 Visual Simple Profile (SP) decoder and the MPEG-4 Advanced Video Coding (AVC)/H.264 Constrained Baseline Profile (CBP) decoder respectively for hardware and heterogeneous hardware/software implementations. Results show that using appropriate combinations of the optimization techniques, throughput can be improved by up to 12x compared to the original design. Compared to similar works in literature using classical and other design methodologies, some design alternatives have also shown to be comparable and superior in various performance criteria. Consequently, this work proves the viability of complex systems design and implementation using dataflow programming, not only for higher degree of productivity but also real-time performance.

Keywords: optimization techniques, dataflow programs, hardware implementation, heterogeneous platforms, digital signal processing, performance evaluation, design space exploration.

Résumé

Les systèmes de traitement de signaux numériques peuvent être représentés par des réseaux de processus flux de données, qui est un modèle de calcul graphique, où un certain nombre de processus concurrents communiquent via des canaux de communication unidirectionnels de type FIFO. L'un des langages de programmation formels qui capte directement ce MoC est le langage flux de données CAL. Ce language de haut niveau, indépendant de la plate-forme, est également capable de synthétiser la description du modèle pour des implémentations logicielles et matérielles. Dans cette thèse, une nouvelle méthodologie de conception dédiée au langage CAL est présentée, validée et utilisée pour explorer rapidement des alternatives de conception à un haut niveau d'abstraction, par l'application de nouvelles techniques d'optimisation sur des descriptions CAL complexes. La première technique d'optimisation est la technique dite de remaniement qui vise à minimiser la latence du système en exploitant le parallélisme des tâches et des données, l'optimisation de la mémoire par la fusion de données ainsi que l'élimination des accès à la mémoire redondants. La seconde technique d'optimisation est une refactorisation qui consiste à diviser le chemin combinatoire le plus long dans le flux de données, de telle sorte qu'une fréquence maximale de fonctionnement plus élevée soit obtenue. Cette technique consiste à trouver le meilleur partitionnement tel que les ressources soient minimisées. La troisième et dernière technique d'optimisation consiste à minimiser et à optimiser la taille des tampons mémoires de type FIFO, en utilisant diverses approches et techniques sur le modèle de flux de données lui-même, ainsi que sur sa mise en œuvre sur la cible. Comme cas d'étude, l'efficacité des techniques d'optimisation est étudiée sur l'implantation des spécifications CAL des décodeurs vidéo MPEG-4 Simple Profile (SP) et MPEG-4 Advanced Video Coding (AVC)/H.264 Baseline Profile Contraint (CBP) sur des cibles matérielles et des cibles hétérogènes logicielles/matérielles. Les résultats montrent que l'utilisation des techniques d'optimisation permet d'améliorer les performances (en terme de débits) par 12 comparés au modèle originale. Certaines alternatives de conception ont également montré qu'elles étaient comparables ou de qualité supérieure dans divers critères de performance comparé à l'état de l'art. Par conséquent, ce travail prouve la viabilité de la méthodologie pour la conception et la mise en œuvre de systèmes complexes en utilisant la programmation flux de données, non seulement pour son degré élevé de productivité ainsi que pour ses performances en temps réel.

Mots-clés: techniques d'optimisation, des programmes de flux de données, le matériel la mise en œuvre des plates-formes hétérogènes, traitement numérique du signal, de la performance évaluation, la conception de l'exploration spatiale.

Contents

List of Figures

List of Tables

1 Digital systems design and implementation: state-of-the-art

Digital systems and devices nowadays are prevalent in almost all aspects of everyday life: from consumer electronics such as mobile phones, digital television, and personal computers, to specialized electronics such as those used in industrial process, medicine, and transportation. One of the major tasks behind these digital systems is the processing of signals characterized by discrete symbols, usually by a mathematical manipulation to achieve a certain objective or functionality. Systems that mainly perform this process are known as *Digital Signal Processing* (DSP) systems, where the figure of merit or quality is typically defined by three criteria: throughput, power, and resource. In mobile electronics for example, high throughput system equates to high performance device; low power means longer battery life; and low resource translates to low cost. Since the past few decades, the goal has always been to improve these criteria by *optimizing* the algorithm and/or architecture of the design with some target objectives and specified constraints. The optimization process typically begins with design analysis and profiling to detect system bottleneck; followed by the task of improving a given criteria by some design modifications that may (or may not) affect another criteria. The exploration and evaluation of different design alternatives during the optimization process is called the *Design Space Exploration* (DSE).

The quality of DSP systems also depend on the implementation platform. Software platforms with limited processing cores and generally high operating frequency may not offer the best combinations of performance and power compared to hardware platforms. On the other hand, hardware designs are not always easy and known to be time consuming to develop, which do not bode well with the ever increasing algorithm complexity and the decreasing time-to-market. With the quest to achieve close to ideal electronic devices in the shortest amount of time, the trend today is to exploit the synergism of software and hardware through their concurrent designs, also known as *co-designs* [91]. This means that system implementations involve both software and hardware platforms that are designed to run in parallel, with their synchronization and communication using the relevant interfaces. This is otherwise known as software/hardware *heterogeneous* implementation. In this chapter, the state-of-the-art in specifying and implementing heterogeneous digital systems is presented, first using imperative languages, then using high-level of abstraction, and finally a concluding remark based on these methodologies.

1.1 Design methodology with imperative languages

Figure 1.1 depicts a classical design flow for performing software/hardware co-design and implementation. It is divided into six different stages. In stage-1, the overall design architecture is conceptualized at a high level of abstraction, either manually or using a dedicated platform-independent models such as UML and SysML ([28], [19]). Some of these models also allow simulation and profiling, where results can be used for model refinements. Once this top-level architecture is defined, the system can be mapped and partitioned into software and hardware parts in stage-2. This can also be performed manually or automatically based on some design criteria and attributes such as parallelism potential and complexity. In stage-3, the relevant parts are further refined to obtain a suitable architecture for a given platform. During this stage, the design language is chosen, and the specification is studied in detail for implementation. In stage-4, the design is coded in the chosen language, typically using imperative languages such as C/C++/Java and VHDL/Verilog respectively for software and hardware implementation.

Once the design is completely specified, the next step is the compilation process in stage-5. For software designs, the compilation process begins with the pre-processing of the included header files and symbolic constants. This generates the so-called *expanded source code* that goes through the assembler for generating the *assembly language* for the platform. The assembly language specification is then assembled into *object code*. The final step is to link all the object codes in the program (and the library functions) to produce an executable file that can be run on a microprocessor. As for hardware designs, the compilation process (also known as RTL synthesis), begins with the analysis and translation of the program code from behavioral or structural description to a gate-level representation. This representation is then further refined to use platform specific gates and devices. For reconfigurable hardware or FPGAs, the next step is the place and route process for the selected device, followed by the generation of the bit stream for physical implementation. For custom hardware or ASICs, an additional step of floor-planning is required before place and route, followed by a layout process that can either be done manually or automatically. The layout is typically extracted as a *GDSII* file that represents among others, the geometric shapes and text labels. This file is given to the foundry for fabrication. In both FPGAs and ASICs, the final result is a custom hardware block on the device that can be executed using relevant input stimulus. Note the difference between software and hardware implementation: software design executes a sequence of instructions using an existing hardware (i.e. microprocessor), while hardware design creates a new specific hardware block for the design or function.

The final stage-6 is where the software and hardware designs are physically implemented and verified on the platform. During this stage, the relevant communication interfaces are also developed to synchronize the operations between the different platforms. The design is first verified for correct operation individually for each platform, and then co-verified for heterogeneous platforms. If all aspects of the design are satisfied, then the design can be documented and delivered. However, if the program behavior is incorrect and/or the required performance are not met, then the designer may 1) reiterate the design process at the design entry (stage-4) by debugging and optimizing the code, and/or 2) modify the high level design architecture (stage-1), by exploring different mapping and partitioning schemes (stage-2), and different platform-dependent architecture (stage-3).

It should be noted that this classical systems design methodology is a *tried* and *tested* method, and possibly the most common way today of specifying and implementing digital systems. The imperative C and Verilog/VHDL programming languages respectively have also been around since the 1970s and 1980s. For this reason, the compilation tools have matured and proven to be very efficient. Many practical designs from various application domain have been successfully implemented using this methodology. The main reason for the efficiencies is the nature of the imperative languages that are generic, and their low level of abstraction that provides very high level of details and controls. This is in fact, both an advantage and limitation. The advantage is that designs can be implemented in almost infinitely different ways. The drawback is that for substantially complex designs, there are too many implementation details to consider that makes the use of imperative languages less productive, especially in the case of hardware languages.

DSP algorithms and systems are getting increasingly complex with the incorporation of higher functionality and/or quality by means of more advanced algorithms. This is evident for example in video codecs, where the successive standards have shown to be significantly more complex in order to support more advanced features. The result of this higher complexity is a better compression ratio with superior subjective and objective quality. The trend of higher design complexity is also evident in other DSP application domains such as in communication systems, image processing, and cryptography. As pointed out however, the main drawback of using classical design methodology is when the design is very complex. Therefore, we make the following assertion: **In time, it can be argued that the classical design methodology may no longer be feasible to be used as DSP systems are becoming more and more complex.**

Besides this major drawback, there are other limitations of using imperative languages for current and future DSP systems, summarized as follows:

- **Lower degree of analyzability**. Imperative design languages are widely used due to their

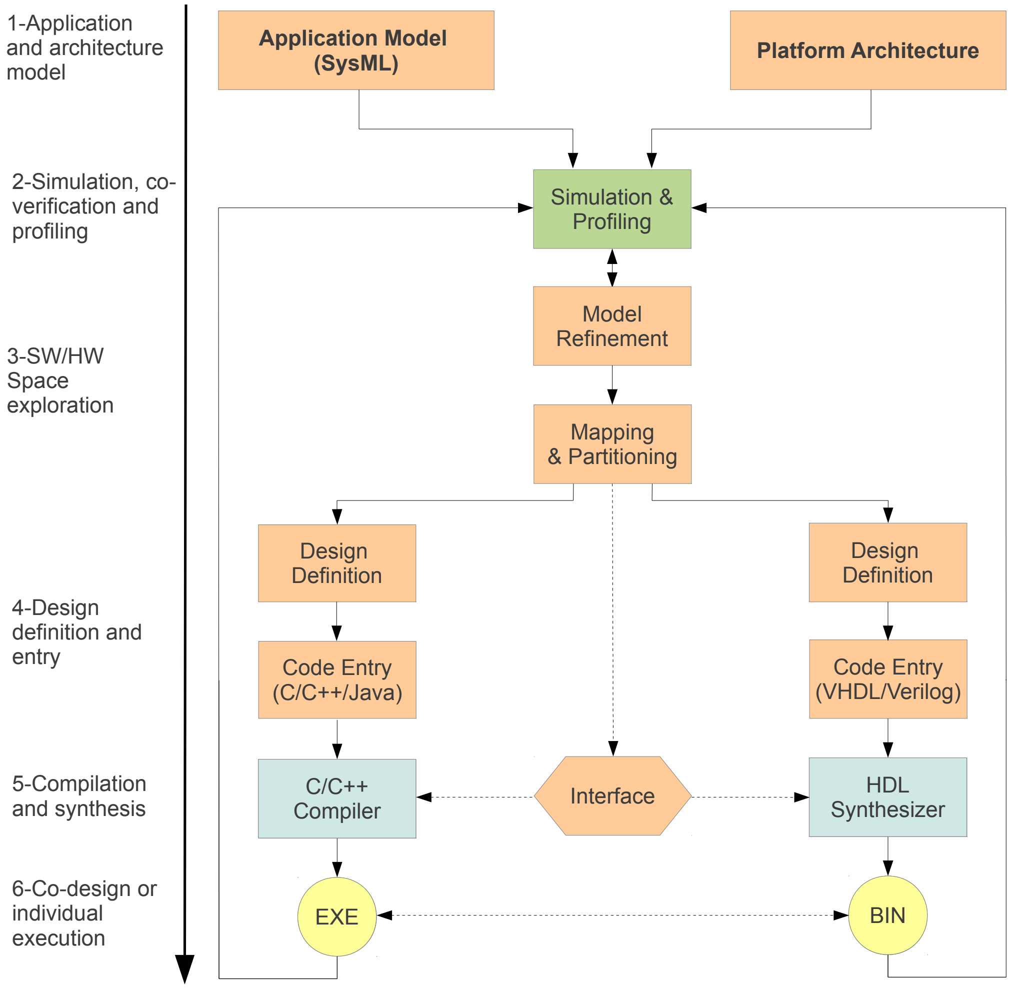


Figure 1.1: Design flow of classical software/hardware co-design and implementation using imperative languages. Note that design definition and entry in stage-4 is platform dependent.

flexibility to model any generic design. Because of this, imperative programs are also more difficult to analyze for bottlenecks and optimizations, especially for an automated approach.

- **Platform dependent**. Software specification of a program cannot be used directly for hardware implementation and vice versa. In order to explore wide range of partitioning schemes, a system has to be designed in its entirety for each platform, which increases development time and cost.
- **Do not explicitly expose parallelism**. This is especially true for software implementation languages, where it is known to be difficult to specify and model parallelism in the program, whereas current and future implementation focus on utilizing multi- and many core processors.
- **Slow design cycle**. This is especially true for hardware implementation languages, where design specification is known to be slow and tedious. With the ever decreasing time-to-market, future systems and devices have short lifetime, and new designs are asked to be released as quick as possible.
- **Scalability is becoming more challenging**. Due to the flexibility of imperative design languages, there is often too much low level components and features that have to be controlled. For DSP systems that focus mainly on data processing, the additional controls (and sometimes unnecessary) make it challenging to scale for future designs.

1.2 Design methodology with high-level languages and models

Due to some of the limitations of implementing modern heterogeneous DSP systems using imperative languages as discussed in the previous section, many high-level languages and models have emerged, one of which is the CAL dataflow language used in this work. The following reviews other design languages and models that are used as an alternative to the imperative languages.

1.2.1 SystemC and C

Perhaps the most widely used specification for software/hardware co-design and simulation is *SystemC* [94]. It aims to establish a common design environment consisting of C++ libraries, models and tools, for both software and hardware parts of the design. SystemC enables design specification at various level of abstraction from system, to behavioral, and to RTL level. In SystemC co-design and *simulation* methodology, software designs remain the same in C++ specification; hardware designs on the other hand utilize the SystemC class library supported by the Open SystemC Initiative (OSCI), which provides the implementation of

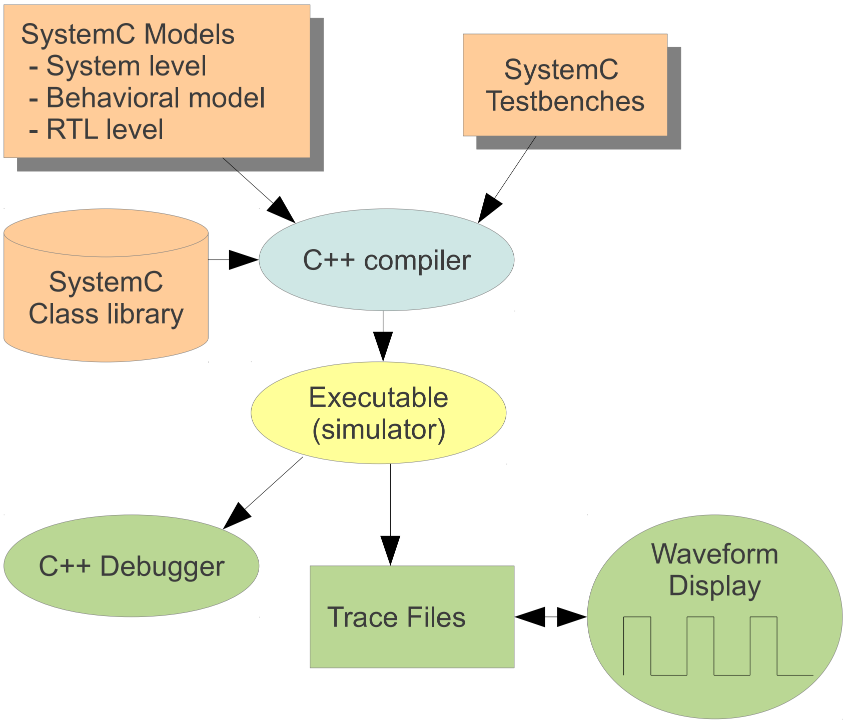


Figure 1.2: SystemC simulation methodology for hardware part of design. SystemC models and testbench can be compiled using a standard C++ compiler, and can then be simulated either using a standard simulator, or a waveform display tool.

hardware specific objects such as concurrent and hierarchical modules, ports, and clocks. It also contains a lightweight kernel for scheduling processes. The systemC hardware model written by a designer, along with the testbench, can be compiled using a standard C++ compiler (with the OSCI library). In addition to simulating the design using a standard C++ debugging environment, trace files can also be generated to view signals using a standard waveform display tool. Figure 1.2 illustrates this systemC hardware design and simulation methodology.

A higher level approach that is typically used with SystemC is called the Transaction Level Modeling (TLM) [41]. It raises the level of abstraction one step above SystemC and features a discrete-event model of computation. In TLM, the details of communication among computational components are separated from the details of the computational components. Communication is modeled by channels, while transaction requests take place by calling interface functions of these channel models. TLMs speed up simulation and allow exploring and validating design alternatives at a higher level of abstraction. The use of TLM typically coincides with the SystemC model specification that is synthesizable to RTL for hardware implementation. The generated RTL code from SystemC however, are not nearly as efficient as manual specification. For these reasons, hardware design is still mostly implemented using RTL languages, but with rapid functional verification using systemC and TLM.

While systemC and TLM are mainly used today for systems verification, there has been an intensive research for synthesizing C to RTL for quite sometime. This again exploits the commonality of specification languages for both software and hardware parts for higher degree of co-design productivity. There is a large variety of system design tools and methodology that use C programs for generating hardware code, for example the GAUT tool from LabSTICC [86] and the Spark framework [49]. There is also a strong interest from major vendor industries such as those from Altera (C2H [9]), Xilinx (Vivado HLS [11]), and Mentor graphics (Catapult C [5]). Figure 1.3 illustrates a design flow for Catapult C that is capable of synthesizing C++ and SystemC description into RTL for simulation and implementation (other tools follow similar design flow). However, as pointed out in [90], synthesizing C model to hardware is still a formidable task due to the following: 1) dynamic memory allocation cannot be casted to hard-wired circuits, 2) pointer resolution in hardware is difficult, if not impossible, 3) C programs were originally conceived for uni-processor sequential execution, which does not translate well to concurrent processes in hardware, 4) the specification of hardware circuits entail some structural information which is missing in C specification, and 5) detailed timing information is missing, which is important in hardware for performance and interface requirements. While C program is convenient and productive for co-designs due to the absence of HDL specifications, these issues may limit the performance that can be achieved for such system.

1.2.2 Synchronous languages

As an alternative to C language based methods, another important class of high-level design language is called the *synchronous* language. It is in fact a language designed to program reactive systems, i.e. systems which maintain a *permanent* interaction with their environment. This includes applications in automatic process control, monitoring, and signal processing. The difference with imperative languages is the use of the following abstraction: 1) operators react instantaneously with their inputs, i.e computations have zero duration, and 2) time is just a succession of events, i.e. no explicit reference to a notion of physical time. The advantage is that it permits a high level modular programming style that is simpler and more rigorous for manipulation. Some of the variant of this class of language allows the synthesis of its description to both software and hardware implementation languages, for example in the case of the first of such language, called Esterel [1]. While Esterel adopts an imperative language style, other variants such as Signal [21] and LUSTRE [51] adopt a dataflow based architecture.

The key feature of synchronous languages is the design abstraction that allows programmers

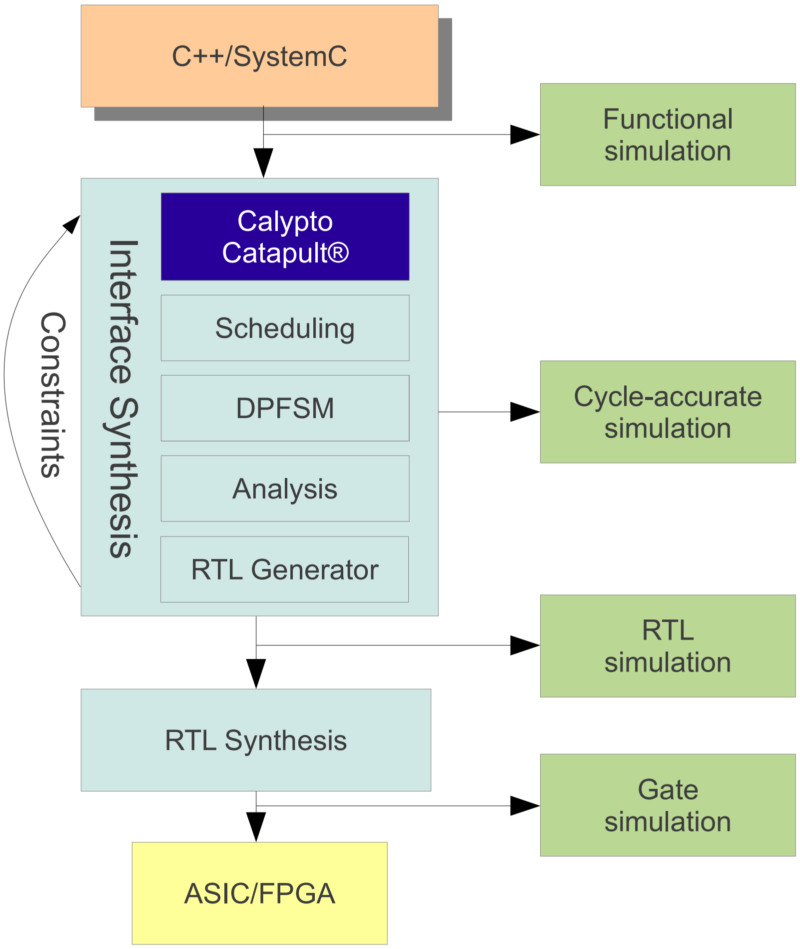


Figure 1.3: Design simulation and implementation flow from C++/SystemC to RTL using Catapult C. DPFSM in the interface synthesis stands for the Datapath Finite State Machine.

to think of their programs as reacting *instantaneously* to external events. The hypothesis also assumes that programs are able to react to an external event, before any further event occurs. This is in contrast to *interactive* systems, which interact continuously with environments that possess synchronization capabilities such as in operating systems. This is also in contrast to *transformational* systems, whose data are available at the beginning, and which provide results when terminating such as the case in DSP applications of video compression and multimedia processing. While synchronous language provides a good design abstraction for control-dominated systems, the semantics are still quite limited when it comes to programming data-dominated DSP applications.

1.2.3 Pre-configured blocks and templates

Another design method consists of describing design architectures using pre-defined templates. It offers the advantage of using a well defined and pre-configured IP blocks that can lead to optimized and efficient code generation for implementation. Using this methodology, designers could rapidly specify a component from a library of IP blocks, and may integrate and specify custom blocks (that do not exist in the library) using RTL languages and other models. Examples of such method include the work of Lahiri et. al [73], the PICO framework [20], and simpleScalar [117]. There also exists some commercially available tools such as AccelDSP [6] from Xilinx , DSPBuilder [8] from Altera ,and SPW [3] and Cocentric System Studio [4] from Synopsys. Some of these tools could also be integrated with MATLAB and Simulink from Mathworks [10], and provide a subset of synthesizable constructs to RTL for hardware implementation. Figure 1.4 illustrates an example of such design methodology for the case of using Xilinx System Generator with design input from Simulink, DSP blockset, or CoreGen for automatically generating synthesizable RTL description. The RTL description can then be used directly for hardware simulation and implementation. Other design tools follow roughly the same methodology. While these models have proven to be efficient in their own right, frequently the designers still have to utilize low level RTL codes for some specific functions. Furthermore, the methodologies are not standardized and some are quite specific to a particular vendor/framework, which thus requires deep knowledge of the set of tools and their appropriate usage. They also lack a formal high-level representation and the semantics of the processes within the blocks and the associated model for the flow of data. Dataflow programming models overcome these issues by formally defining the way processes are executed and how the processing nodes interact with each other.

1.2.4 Dataflow programming models

Dataflow programming models have a rich history dating back to at least the early 1970s, including the seminal work by Dennis [31] and Kahn [67]. Such model can be represented as a

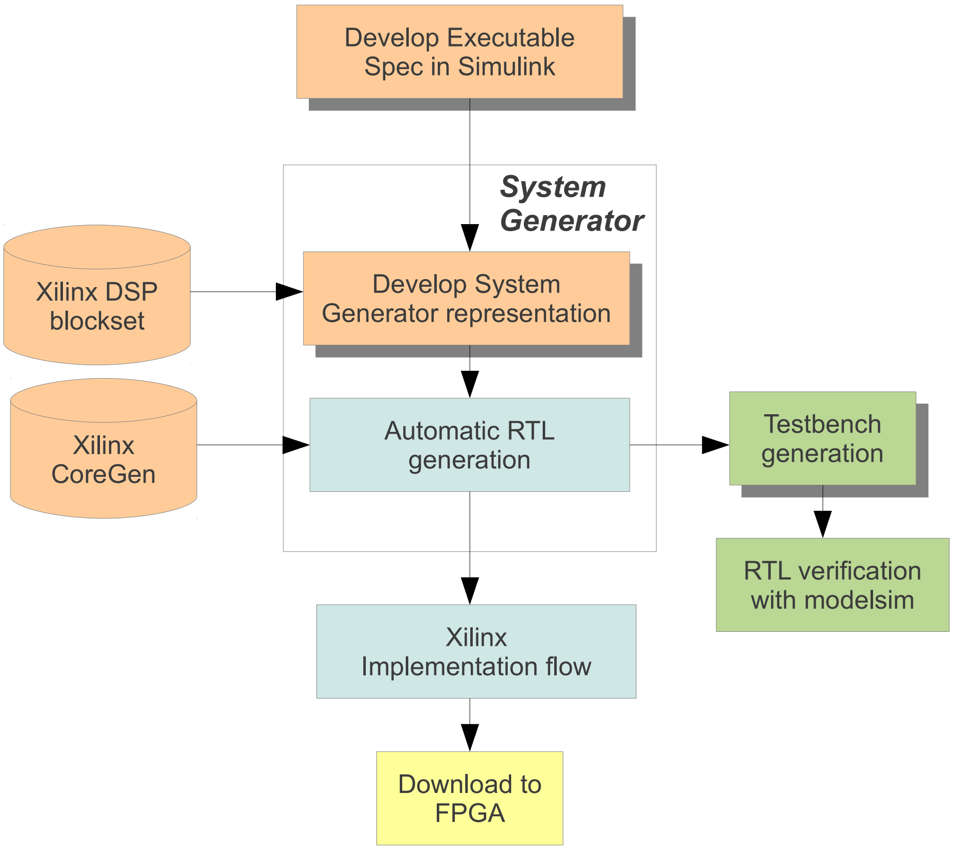


Figure 1.4: Design methodology for automatically generating synthesizable RTL description from Simulink, DSP blockset, and CoreGen using Xilinx System Generator. The generated RTL description can be used directly for hardware simulation and implementation, thus relieving the need to manually specify programs in RTL.

directed graph with nodes as concurrent processes and edges as interconnections through a uni-directional FIFO, also known as the Kahn Process Network (KPN). A special case of KPN is called the Dataflow Process Network (DPN) [77], where the processes are divided into repeated *actor firing* that defines an execution of a quantum of computation. This reduces the considerable overhead of context switching that incurs in KPN. However, DPN is a generic model of computation whereby the processes could not be scheduled at compile time, and the required bounds for the FIFO interconnections cannot be determined statically. The reason for this is because the execution order typically depends on the input values. Different values could change this order, and therefore may require different bounds. A special case of DPN that overcomes these issues is called Synchronous Dataflow (SDF) [76], in which the number of data samples produced or consumed by each processing node on each invocation is known a priori. Using this model, execution order is static, regardless of the input values. An extension to SDF is called the cyclo-static dataflow (CSDF) [26], where an actor contains finite cycles for different sequence of firing. SDF and CSDF are highly analyzable due to their static nature, but trades-off with the high expressive power (i.e. flexibility) of DPN. The different variations of dataflow model is often referred to as the *Model of Computation* (MoC). One common property across all of these dataflow models is that individual actors encapsulate their own state, and thus do not share memory with one another. Instead, actors communicate with each other exclusively by sending and receiving tokens along the channels connecting them.

The concept of actors as processing nodes was first introduced in [53] as means of modeling distributed knowledge-based algorithms. It proposes an approach where the whole system acts as a society of communicating knowledge-based problem-solving experts. These experts can be further decomposed into *actors* at the lowest granularity level. The actors are objects that interact purely in a local way by sending messages to one another. Actors have since then been evolved over time, and also widely used, especially in embedded systems, where *actor-oriented-design* (AOD) is a natural match to the heterogeneous and concurrent nature of such systems. The interface of an actor is defined by ports and parameters, with the precise semantics of dataflow actors depend on the MoC, but conceptually, it represents signaling between components. This is in contrast to *object-oriented-design* (OOD) using software imperative languages, which emphasizes inheritance and procedural interface. Figure 1.5 depicts the difference between AOD and OOD, taken from [29]. It is clear from this figure that AOD explicitly abstracts parallelism among the actors, whereas OOD is viewed as a purely sequential call to class methods.

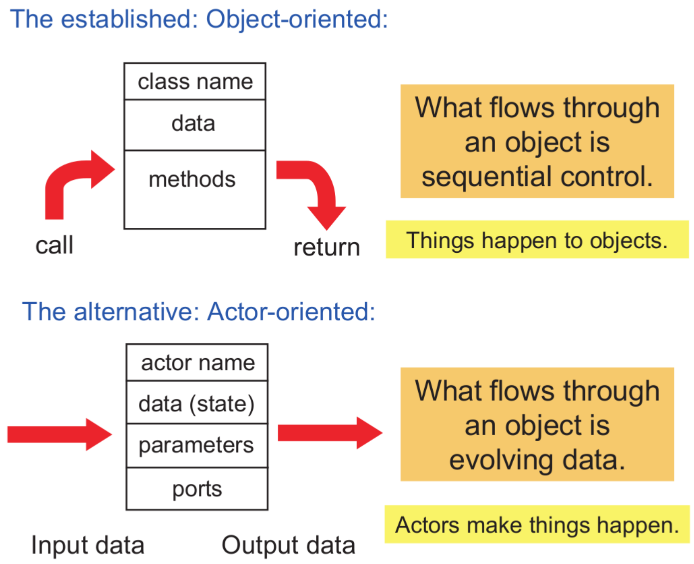


Figure 1.5: Actor-oriented-design versus object-oriented-design.

The following reviews some of the available tools and languages for modeling dataflow-based (primarily actor-oriented) digital systems, along with their strengths and limitations.

Ptolemy-II framework. One of the most long-running efforts on modeling, simulation, and design of embedded systems using dataflow and actor-oriented architecture is the Ptolemy project [34]. The core of the project is a heterogeneous simulation and design environment supporting multiple models of computation, which started with *Ptolemy classic*, and later evolved to *Ptolemy II*. In Ptolemy II, the semantics of a model is not determined by the framework, but by a software component in the model called the *director*, which implements a model of computation. A major emphasis of the project has been on *understanding* the heterogeneous combinations of models of computation realized by these directors. One of the main goals is to build frameworks that help deep understanding of the different domains in embedded systems applications, and their implication on analysis and code generation.

PeaCE framework. PeaCE [50] is an acronym for *Ptolemy extension as Codesign Environment*. It aims to extend the original Ptolemy project with the following objective: a design framework common to all system level design activities, which include specification, cosimulation, design space exploration, interactive partitioning, synthesis of software, hardware, and their interface. Figure 1.6 depicts the design flow for PeaCE. The essential part here is that dataflow and architectural specification is specified only *once* at high level, with manual partitioning and scheduling, and automatic software and hardware code generation for implementation. The dataflow specification for PeaCE is restricted to SDF for obtaining a deterministic program. In order to improve design flexibility, the SDF model is extended to enable control structures

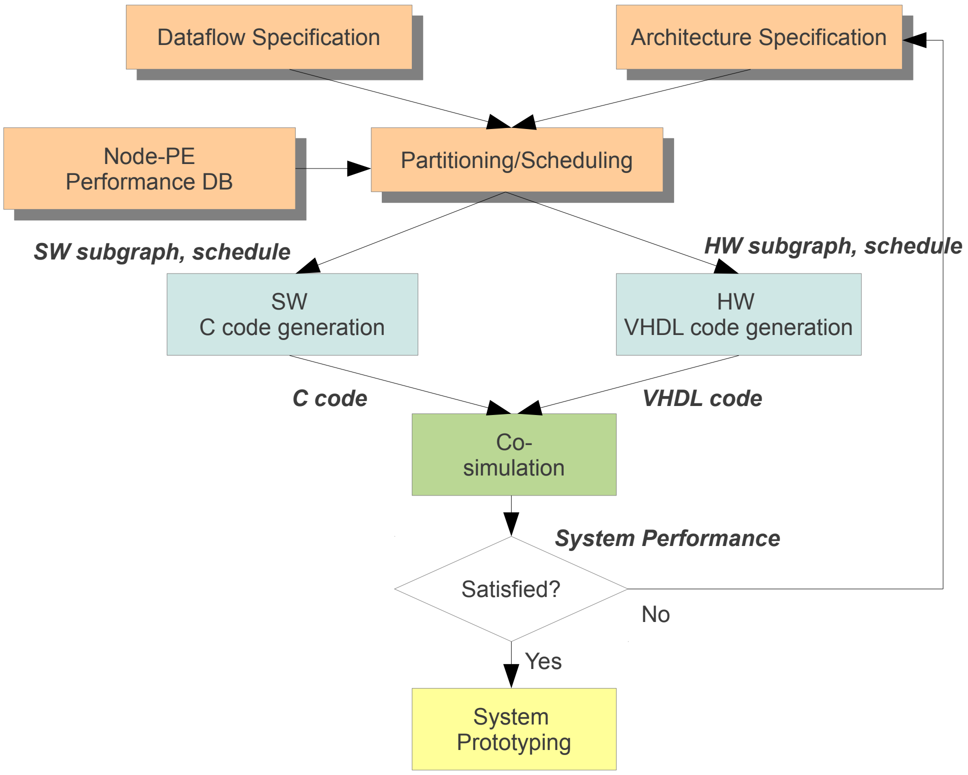


Figure 1.6: Systems design methodology using the PeaCE framework.

defined by FSM inside SDF vertices. This however, still imposes a restriction on complex systems designs compared to using a generic DPN. The PeaCE framework also does not report any formal language or semantics for specifying the dataflow actors and networks, but rather based on a graphical representation of the system. Furthermore, the current state of the work mainly focuses on software code generation for multi-processor system-on-chip (MPSoC) ([74], [65]). As for hardware code generation, it has not been shown to be efficient compared to other design methodologies, and also not been demonstrated to support a wide range of complex design cases.

StreamIt language. StreamIt [118] is a formal dataflow programming language and a compilation infrastructure, specially engineered for modern streaming systems. It is designed to facilitate the programming of large stream applications, as well as their efficient mapping to a wide variety of single- and multi-core processors and clusters of workstations. Similar to the PeaCE framework, it aims to use SDF as its model of computation. In order to increase design flexibility, it introduces novel constructs such as structured streams, parameterized data reordering, and teleport messaging. Recent work in [113] also presents a hybrid static-dynamic scheduler that strikes the right balance between the expressivity in dynamic actors and the performance of static actors. StreamIt however, focuses exclusively on software implementation from high-level stream programs, and currently does not support any hardware implementation.

COMPAAN/LAURA. The tools COMPAAN [71] and LAURA [126] have been developed separately, with the former as a tool to compile imperative programs in MATLAB into a concurrent representation in KPN, and the latter as a tool that takes in a KPN representation and compiles it into a synthesizable RTL. COMPAAN starts the transformation process by converting a MATLAB specification into a *single-assignment-code* (SAC) specification that describes all parallelism available in the original program. This results in a data structure representing a graph with dependencies, used to create a KPN representation that can be analyzed and simulated using the Ptolemy II framework. The KPN to RTL synthesis tool LAURA also supports the use of IP cores for MATLAB functions defined in the original program. The design methodology is illustrated in Figure 1.7 with the three step process from specification to implementation using COMPAAN, LAURA, and commercial RTL synthesis tools. The methodology however, have only been verified for simple design cases such as the DCT and the QR factorization algorithms, and similar to the PeaCE framework, does not guarantee efficient implementation for substantially complex design cases. Furthermore, this approach still uses a variation of the imperative languages, which have been discussed has severe limitation in generating efficient RTL codes.

1.3 Conclusion

In this chapter, several state-of-the-art systems design methodology have been discussed, where for each one, their strengths and limitations were presented. We started with the design methodology using imperative languages of C/C++ and Verilog/VHDL, and pointed out its main drawback of implementing very complex applications, among others. Although this methodology allows full range of details and controls, design productivity greatly diminishes as applications become more and more complex. For this reason, there is a trend of using high-level of abstraction design methodology, where several variations and approach were discussed, including the use of SystemC and C, synchronous languages, pre-configured blocks and templates, and dataflow programming models. The C programming language approach exploits common specification for both software and hardware parts, but suffers from the difficulty of synthesizing to efficient RTL codes for hardware implementation. The synchronous language approach defines new language semantics, but more suited to a control-dominated signal processing applications. Using pre-configured IP blocks may result in efficient implementation, but suffers from the constraints of the architecture to a given class. Finally,

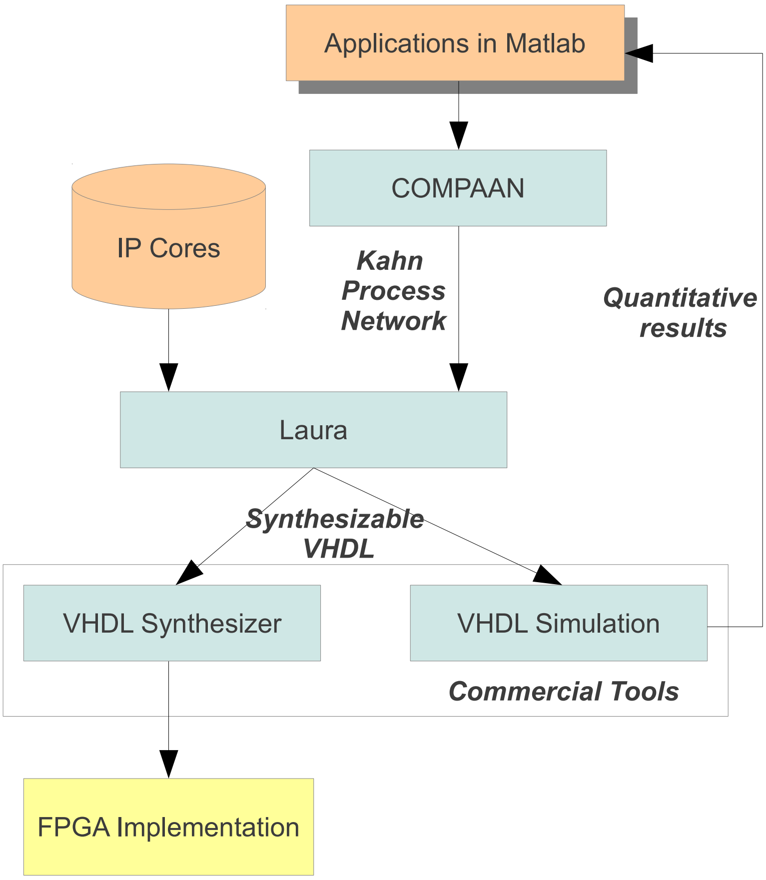


Figure 1.7: Systems design methodology using the tools COMPAAN and LAURA.

dataflow programming models are seen to be efficient for a large class of data-dominated signal processing applications, but the currently available tools and techniques have some limitations as follows: the Ptolemy II framework is mainly developed to explore different models of computation and not used for implementation; the PeaCE framework lacks a formal language, focuses more on MPSoC implementation, and is still lacking in hardware synthesis and implementation; the StreamIt language focuses solely on single- and multi-core processors; and the COMPAAN/LAURA have issues in generating efficient hardware code due to the use of a variation of an imperative language.

The CAL dataflow language used in this work was created at around the same time as the other dataflow programming tools and languages: PeaCE, StreamIt, and COMPAAN/LAURA, but offers a different approach to the design and implementation of DSP systems. While others support only SDF model of computation, or the generation of KPN from another representation, CAL allows the explicit specification of the three major dataflow model of computation: SDF, CSDF, and DPN. In the next chapter, a new systems design methodology using CAL is presented for the complete software/hardware co-design flow from specification to implementation, with analysis, synthesis, optimization, and space exploration.

2 Design methodology with CAL dataflow programming

CAL is one of the domain-specific dataflow programming languages, designed to raise the level of abstraction in the design and implementation of DSP systems. It is based on the concept of actors, and was created as part of the Ptolemy project. The final language specification was released at the end of 2003 [33]. Since then, the design methodology, framework, and tools have actively been developed with the purpose of providing a complete software/hardware (SW/HW) co-design flow from specification to implementation, together with program analysis, optimization, and space exploration. This chapter presents such a design flow with developments on the associated tools and techniques at each main stage of the flow.

Figure 2.1 depicts a proposed design flow for SW/HW co-design and implementation with CAL dataflow programming [23]. It is divided into six main stages. The first stage is the specification stage, where the platform architecture is defined (CPU, FPGA, GPU, etc.), and the application is coded using the platform-agnostic CAL language (Section 2.1). This means that the same specification can be used for any target platform. The second stage is where this high-level specification in CAL is functionally verified with a platform-independent simulator. During this stage, programs can be profiled statically for complexity, and/or dynamically for longest path in the network (Section 2.4). This allows the detection of bottlenecks early in the design process. In the third stage, the design can be 1) refined by modifying the architecture of the critical actors and actions obtained from profiling, called refactoring, and 2) mapped and explored for various combinations of software and hardware partitioning. For software part, the design can also be explored for different multi-core or many-core processor scheduling and partitioning (Section 2.2).

The fourth stage is the code generation stage. For a given CAL program, the software part and hardware part of the design respectively can be sent for synthesis to imperative C/C++ and HDL languages (Section 2.3). For co-design and implementation, the relevant interfaces are also required to interconnect the different platforms, typically coded manually using

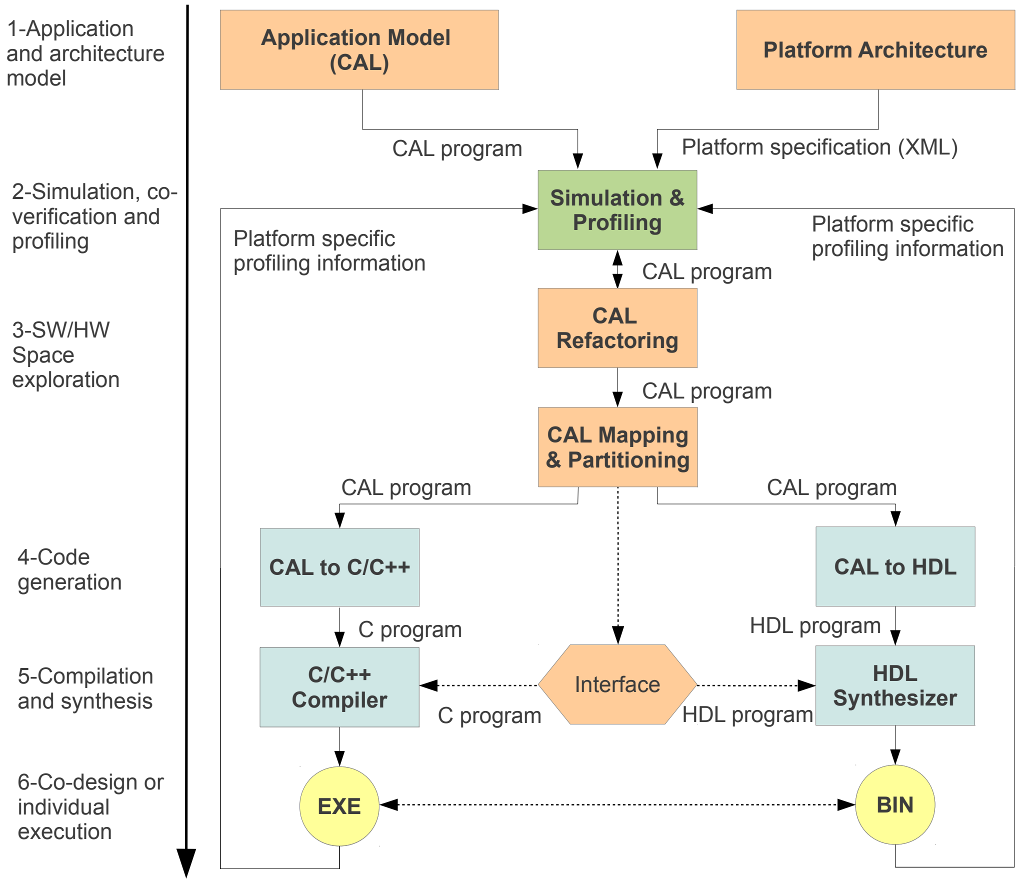


Figure 2.1: Design methodology for SW/HW co-design and implementation using CAL dataflow programming.

imperative languages. In the fifth stage, the imperative language specifications can be compiled/synthesized using standard tools to obtain software executable and hardware binary files for physical implementation (Section 1.1). In the final stage six, the design can be verified both individually and as co-platforms. If the final design implementation behaves correctly with the required performance, then it can be documented and delivered. If not, the design can be refined at the CAL specification stage, with or without platform specific profiling information obtained from stage six (Section 2.5).

There are several notable advantages of using this methodology compared to the classical methodology given in Figure 1.1. First, the use of high-level dataflow specification allows higher degree of program analyzability, explicit parallelism, and fast design cycle compared to coding using imperative languages. Second, design entry is platform *independent* and can be utilized for any implementation platform. The ability to enter a design only *once*[1] undoubtedly would increase design productivity where a design does not have to be coded in its entirety for each platform for exploring the partitioning space. Third, platform dependent design optimizations can be performed at the highest abstraction level, which would result in rapid exploration of design alternatives. Moreover, with optimizations performed mainly on the architectural level of the design, larger improvement gains can typically be achieved compared to low-level optimizations.

2.1 CAL dataflow language

The CAL dataflow language supports dataflow specification using the generic DPN, as well as its subset of SDF and CSDF, thus presenting a wide range of design trade-off between flexibility and analyzability. It directly captures the description of actors as making discrete steps enabled by several conditions on the state of the actor, and the availability of input tokens. As depicted in Figure 2.2, a CAL dataflow network can be represented by a directed graph $G(A,F)$, where A and F respectively are the actors and channels in the network. Each actor $a \in A$ contains a set of actions T_a, where each action $\tau \in T_a$ consists of a *firing rule* and a *firing function*. The firing rule determines *when* the action could fire, which depends on the following conditions:

1. **Availability of input tokens**. If an action has input port(s), then the action is fired if and only if there are sufficient data tokens on the input channel(s)
2. **Guard conditions**. If an action has a guard condition, then the action is fired if and only if the guard condition satisfied.
3. **Actor scheduler**. If an actor contains a finite state machine scheduler, then an action is fired if and only if the action is in the current firing state of the actor.
4. **Priorities**. If an actor contains a priority definition for a given action, then the action is fired if and only if the priority condition is satisfied.
5. **Availability of output space**[2]. If an action has output port(s), then the action is fired if and only if there are sufficient space on the output channel(s).

It should be noted that action firing can be non-deterministic, i.e. more than one action can be enabled at a given time. This can be resolved by appropriate specification of guards and/or priority conditions. The firing function determines *how* action(s) are executed when they are satisfied by the firing rule, by performing the following steps:

---

[1] different platforms may require different coding style

[2] This condition is not defined in the language specification, but required for practical design simulation.

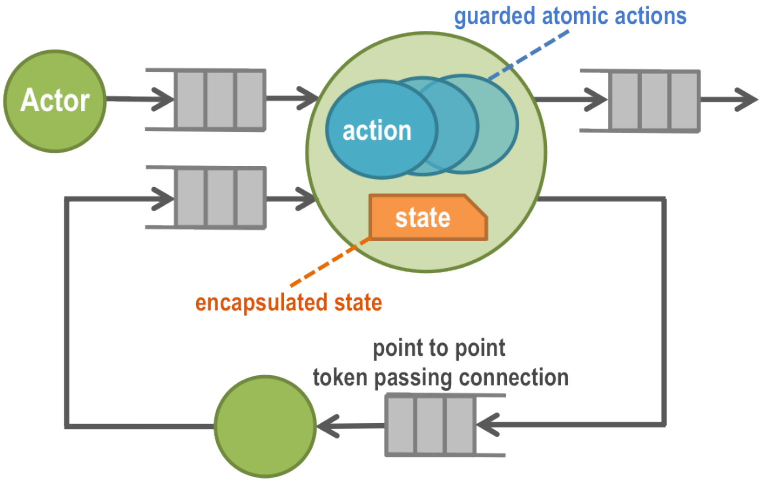


Figure 2.2: CAL dataflow network example. Actors are interconnected by FIFO buffers. Each actor contains a guarded atomic action and an encapsulated state.

1. If the action has input port(s), then it first reads and consumes the input tokens from the input channel(s).
2. If available, the algorithm in the action body is executed. If the algorithm contains access to state variables, then the state variables are updated during this step.
3. If the action has output port(s), then it produces the output tokens on the output channel(s).
4. If the actor contains a finite state machine scheduler, then it modifies the state of the actor to the next action firing state.

A dataflow network in CAL typically consists of several interconnected actors, and is represented by a network file that specifies the list of actors, parameters, and interconnections with their corresponding sizes. Each actor is described by the formal CAL dataflow language, with an example given in the next subsection.

2.1.1 CAL actor example: inverse quantization

In order to further illustrate actor modeling with CAL, Figure 2.3 shows an example of a CAL actor for the inverse quantization function used in the MPEG-4 simple profile decoder. The structure is as follows. It contains three input ports `DC`, `AC`, and `QP`, and one output port `OUT`. It also contains three actions `dc`, `ac`, and `done`, with the latter containing a guard condition of `count = 63`. The actor also contains a finite state machine definition that begins with the state `st_dc` that first verifies that the firing rule for the action `dc` is satisfied, and then fires the action using its firing function. The actor then goes to the next firing state `st_ac`, where action selection between `done` and `ac` is selected based on the priority condition `done > ac`. This means that if the firing rules for both `done` and `ac` are satisfied, then the scheduler would select the `done` action due to the higher priority. In this case, the action `ac` is fired when the value of the state variable `count` is less than 64. The firing of the `ac` and `done` actions respectively cause the next firing state to be `st_ac` and `st_dc`. This process of selecting and firing an action at every discrete step repeats forever as long as the firing rule of at least one action is satisfied.

It is clear from this example that the actor is specified at a high-level of abstraction; the algorithm in the body of the action `ac` is mainly concerned with the mathematical manipulation of the values of the variables for processing. There is no register inference or gate-level control and operation as typically required for low-level RTL specification. This simplifies the design of complex systems such that the focus can be made solely on processing the algorithm without having to worry about the low level implementation details.

2.2 Mapping and partitioning CAL programs

The task of mapping and partitioning CAL programs for heterogeneous implementation can be divided into two phases: 1) SW/HW partitioning for determining the optimum mapping of a CAL program to software and hardware parts of the design for a given set of objectives and constraints; and 2) SW partitioning and scheduling respectively for assigning actors to processor cores and determining the order of execution of the actors assigned to the cores. Note that for HW implementation, partitioning and scheduling are not relevant since each action is assigned to its own dedicated resource with a self-scheduling based on the input tokens and the actor scheduler. In other words, the way hardware implementation is “partitioned and scheduled” depend entirely on the design architecture. The following reviews the two phases of SW/HW and SW mapping and partitioning.

2.2.1 SW and HW partitioning

The choice of selecting SW or HW platforms for different parts of the design specification mainly depends on the design requirements. This includes criteria such as throughput, resource, power, temperature, etc. Different SW/HW partitioning schemes may result in different evaluation of the criteria; the problem then reduces to finding the best SW/HW partitioning scheme for a given design that optimizes one or more of these criteria. For substantially large and complex designs, there could be many design alternatives that can be explored and evaluated for all possible partitioning schemes. Finding the global optimum solution is certainly not trivial, and in some cases, may not even be possible.

```
actor inverse_quantization (int Q_SZ, int S_SZ)
 int(size=S_SZ) DC, int(size=S_SZ) AC, int(size=Q_SZ) QP //inputs
  ⇒
 int(size=S_SZ) OUT : //output

 //state variables
 int quant;
 int round;
 int count;

 //action-1
 dc: action QP:[q], DC:[i] ⇒ OUT:[i]
 do
  quant := q;
  round := ((q & 1) ^ 1);
  count := 0;
 end

 //action-2
 ac: action AC:[i] ⇒ OUT:[outp]
 var
  int v, int o, int outp
 do
  if (i < 0) then i := -i; end;
  v := ( quant * ((i << 1) + 1)) - round;
  if (i = 0) then o := 0;
  else
   if (i < 0) then o := -v;
   else o := v;
   end
  end
  if (o < -2048) then outp := -2048;
  else
   if (o > 2047) then outp := 2047;
   else outp := o;
   end
  end
  count := count + 1;
 end

 //action-3
 done: action ⇒
  guard count = 63
 end

 schedule fsm st_dc :
  st_dc  (dc)   --> st_ac;
  st_ac  (done) --> st_dc;
  st_ac  (ac)   --> st_ac;
 end

 priority
  done > ac;
 end
end
```

Figure 2.3: The `inverse_quantization` actor used in the MPEG-4 simple profile decoder.

There have been numerous attempts to explore design alternatives with different SW/HW partitioning. For example in Vulcan [48], the idea was to start with a hardware-only solution and then migrate as many tasks as possible to software while satisfying the performance criteria. The objective here is to reduce design cost by reducing the required hardware resource. In contrast, the Cosyma [36] design system starts with a software-only solution and the subsequent migration of tasks to hardware in order to satisfy performance constraints. It should be noted that these are early works where it is assumed that the CPU and ASIC worked mutually exclusively. In a more recent approach using high-level synthesis (SystemCoDesigner [69], SoCDAL [14]), the approach is to synthesize a high-level specification of the design to both SW and HW as component "library" blocks that can be mixed and matched for implementation. This library also contains other IP cores such as CPU and memory elements as well. By having all these components in the library, a specific exploration method searches the design space for the optimum solution based on the required design criteria. As for the SW/HW interface, these works provide an automated interface generation based on the selected partitioning. However, despite these efforts, manual partitioning of the application on SW and HW remains a practical approach for embedded systems based on the expertise and knowledge of the designer in comparison to the automated tools [107]. This is the approach taken in this thesis.

2.2.2 SW partitioning and scheduling

The problem of partitioning and scheduling a dataflow graph onto architecture with multiple processing elements (i.e. processor cores) is known to be NP-complete. Therefore, heuristics with polynomial-time complexity are widely used when dealing with large-scale dataflow graphs. One of the approaches is based on *static assignment* [75] where partitioning is defined at compile-time, whereas scheduling is performed at run-time. Partitioning is defined using metrics extracted during the profiling stage using the so-called *execution trace* (defined in Section 2.4). Once partitioning is defined by assigning each actor to its corresponding processing core, the actors are then scheduled dynamically since CAL actors are in general based on the DPN MoC. This may result in an unnecessary runtime overhead. However, static actors in the network can be scheduled statically with less runtime overhead, such as using the approach in [45], where it detects the so-called Statically Schedulable Region (SSR) for a static scheduling.

2.3 Synthesizing CAL programs

The CAL dataflow language was originally used to design and simulate a DPN model of computation. At the beginning of its inception in late 2003, it was supported by a portable interpreter infrastructure called Moses [60], which features a graphical network editor, and allows the user to monitor actor execution including states and token values. CAL models were also possible to be simulated in the Ptolemy environment. It was not until 2008 with the development of the OpenDF framework that it became possible to generate code from CAL programs for implementation. For hardware code generation, OpenDF acts as a front end tool to generate the so-called XLIM code (XML representation for describing a language independent model) from a CAL model. The generated XLIM code is then used with another tool called OpenForge (in the OpenDF framework) that is capable of translating XLIM models to VHDL and Verilog for implementation on Xilinx FPGAs (more on OpenForge in the next section). The tool flow from CAL to HDL in OpenDF is also known as CAL2HDL [64]. Apart from HDL code generation, OpenDF was also designed to generate C code for integration with SystemC [106], and an embedded C code for ARM11 [119].

The main issue with the OpenDF framework is that simulation and code generation are not efficient (i.e. very slow) for substantially large and complex designs. This is mainly due to the architecture that uses XML based processing with XSLT transformations. For this reason, the ORCC framework [2] has been developed (completed in 2009) with the goal of overcoming this issue. Essentially, it intends to move away from XSLT transformations and process CAL actors and networks directly using programming objects. For hardware code generation, the initial work was to port the OpenForge tool into the ORCC framework, which entails generating an XLIM model from CAL. The code generator is called Orc2HDL [24], but was found to be not quite as effective as expected due to XSLT transformations in the OpenForge. Naturally, the following work involves eliminating the slow transformations altogether by processing the objects from ORCC directly into the objects in OpenForge. The new code generator, called Xronos [22] (pronounced chronos) is efficient, and supports new language constructs that were not previously supported in the OpenDF framework. The only current limitation of the HDL code generators using OpenForge (CAL2HDL, Orc2HDL, and Xronos) is the memory element (i.e. block RAM) restriction to Xilinx FPGAs. Designs that utilize memory elements are therefore, only synthesizable to Xilinx devices. Further work is expected to be done in the near future to allow synthesis of generic (or other vendor specific) memory elements in the HDL code generator. It should also be mentioned that apart from HDL code generation, the ORCC framework also supports code generation for the following implementation languages (some are experimental): C, C++, embedded C, Jade, Java, LLVM, OpenCL, PROMELA, and TTA.

2.3.1 CAL to C synthesis

The general overview of the CAL to C synthesis tool is given in Figure 2.4, where it takes as input CAL actors and the network file, and generates one C file per actor, and a single C file for the top level network. A hierarchical network composed of several networks is first flattened, and transformed to a C scheduler. The action scheduler of each actor checks the presence and values of tokens on its input ports, gets the tokens from the ports, and puts the tokens on its output ports. For this, several well-defined functions are declared in both the actor and action

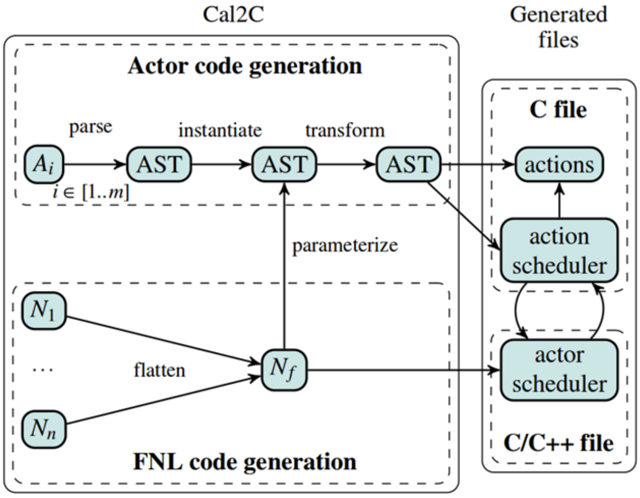


Figure 2.4: Steps and transformations for translating CAL programs to C in the ORCC framework.

schedulers, for example `hasTokens` checks if tokens are present, `peek` peeks tokens from a port, and `hasRoom` checks if the port has rooms to write tokens. The scheduler is designed to switch to another actor only when none of the actions satisfy the firing rule.

For each actor, the first stage of the compilation process is to parse each CAL actor to an Abstract Syntax Tree (AST). The AST uses the LL(k) parser bundled with OCaml called Camlp4. The grammar parsed is embedded in the code where a Syntax Directed Translation generates a node for each grammar rule or group of rules. The AST is then modified during instantiation of the actor, i.e. it removes parameters and replaces them with local variable declarations whose values are specified in the parent network. The next step is a series of transformations to the AST, beginning with type checking. Every expression in the AST is annotated with the type according to the inference rules. If an expression cannot be typed and invalid, then it is not possible to generate code for the actor. If all expressions are well typed, the tool then checks that expressions assigned to variables have types that are compatible with the type of the variables. Once the AST has proven to be correct, the next step is the application of the constant propagation algorithm that finds constant values for all executions, and propagate them through the program. Actor variables are also translated to C global declarations during this step. The last transformation converts the correct AST to an intermediate representation that is closer to C. For this, the C Intermediate Language (CIL) is used for high degree of readability. Before converting CAL to CIL, the names of the variables, actions, etc. are altered to be valid C identifiers. CAL functional expressions are also transformed into CIL imperative constructs, for example the `if` conditions and `for` loops. In compound expressions, temporary variables are used to hold the results.

The final structure of the CAL actor in C program is as follows. Each action becomes a functionally equivalent C function, with the action scheduler implemented in a separate function. The action scheduler determines which action should fire, and calls the relevant function that contains the action to fire. Note that the action scheduler is called every time an action is fired, therefore it is crucial that they are designed to be as fast as possible. This is achieved by carefully checking the actions that are eligible to be fired (according to the state machine), and select a fireable action based on the firing rule.

2.3.2 CAL to HDL synthesis

As mentioned, the CAL to HDL synthesis tool has gone through various evolution from the OpenDF framework (CAL2HDL) to the ORCC framework (Orc2HDL and Xronos). The one common property among all the tools is the use of the OpenForge backend, which is at the heart of the code generation that defines the architecture of the generated hardware. OpenForge was formerly known as Forge, that was developed by Lavalogic for synthesizing Java programs to Verilog. It was then acquired by Xilinx in the year 2000 to bring support for synthesizing C programs to HDL. Xilinx further developed the tool to support HDL code generation from XLIM for use with the CAL dataflow language. Forge is in fact a complex tool with various optimization features such as loop unrolling, dead code elimination, base address uniquifier, memory splitter, reducer, and trimmer, ROM replicated, nested block optimizer, loop variable resizer, etc. Since 2008, Forge became available as an opensource tool called OpenForge.

The work in this thesis utilizes two synthesis tools: CAL2HDL and Xronos. The following first describes how CAL2HDL generates HDL codes from a CAL program. Figure 2.5 depicts the flow of steps and transformations for translating CAL programs to HDL. In essence, it is an XML processing and transformation engine implemented using Java. The tool is divided into three phases: Elaborate, Generate Network HDL, and Generate Instance HDL. In the elaboration phase, CAL actors are first checked for syntax and validity, and then parsed into an initial XML representation called *calml*. The top-level network file (called XDF) is flattened in the case of a hierarchical network. During this phase, the actors are also instantiated in the top level network, where the actors' ID is included, variables annotated, operators canonicalized, and dependencies resolved. The parsed calml actor is then transformed to another XML representation called *pcalml*, which now includes dead code elimination, constant and network expression evaluation, and actor parameter setting.

The following *Generate Network HDL* phase takes in the flattened XDF network and transforms it to a top-level VHDL file. Some of the operations include port type evaluation, data width, fanout, and buffer size annotation, and instance name addition. The top-level network is generated as a VHDL file that simply instantiates and connects all actors in the dataflow

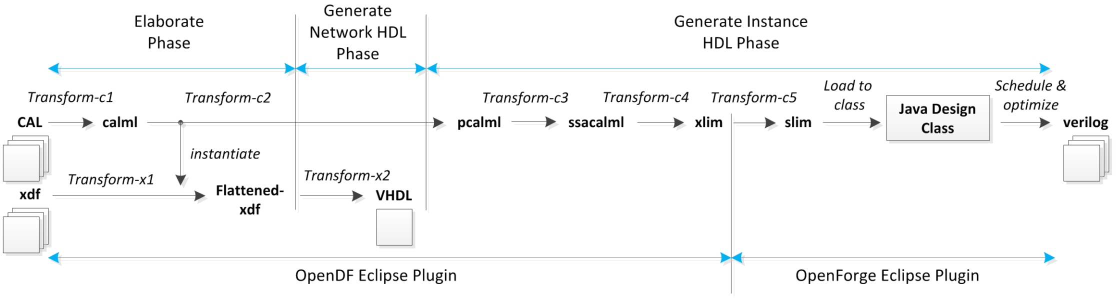


Figure 2.5: Steps and transformations for translating CAL programs to HDL using CAL2HDL.

network using FIFO buffers with size specified in the CAL program.

The final *Generate Instance HDL* phase is the generation of a Verilog file for each actor. The first step is the transformation of pcalml to another representation called *ssacalml*, which is a representation in a single static assignment (SSA) form. This is finally transformed to XLIM to be used with OpenForge. The final XML format called *SLIM* is generated from XLIM, which adds information for Verilog code generation, SSA ϕ function processing, name, loop, and selector fix, casts insertion, and ports routing. The SLIM representation is loaded into a Java *Design* class, which is a class for representing top-level hardware implementation. First, the Java object representing the actor is analyzed and optimized for hardware. This includes operator constant rule, loop unrolling, and variable re-sizer. Memory element is also allocated (BRAM, ROM, register, etc.) and optimized which includes memory reducer, splitter, and trimmer. Following this, a hardware scheduler is also generated based on the specification in the SLIM representation. Finally, a completed Design object for an actor is mapped and written as a Verilog file. The process of HDL code generation from CAL actors is quite complex and consists of many XML transformations using XSLT, the major source for the very slow code generation. However, it has been proven to generate very efficient HDL codes for implementation, such as the work in [64].

The Xronos synthesizer, which was completed as recent as 2012, was developed with three main objectives: 1) to integrate the HDL code generator into the ORCC framework to facilitate co-design with ORCC C code generator, 2) to speed up the code generation process by eliminating all XML transformation process, and 3) to support *all* features of the RVC-CAL language that was not supported before in CAL2HDL which include unsigned integer types, procedures, and multi-token input and output. Essentially, Xronos is implemented in the ORCC framework that already supports all features in RVC-CAL in its Intermediate Representation (IR) in AST. This representation (after several transformation) is fed directly into the OpenForge Design class via the IR representation, thus bypassing all computationally extensive XML transformations from XLIM to SLIM.

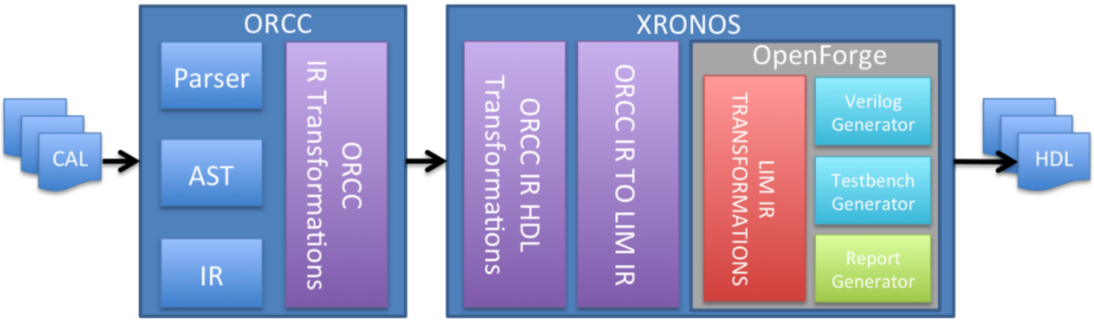


Figure 2.6: Overview of the architecture for synthesizing CAL to HDL using Xronos.

Figure 2.6 presents an overview of the Xronos architecture. It uses the ORCC compiler as its front end, by first parsing the CAL actors and generating the IR for each one. The IR is then serialized to create an actor object, which is transformed to a SLIM representation used in OpenForge. The necessary transformation from SLIM to OpenForge Design class is performed during this step, which include the SSA transformation, 3AC for each operation with 4-tuples of (operator, operand1, operand2, result), casting for data type and size, repeat pattern for multi-token input and output, function and procedure inlining, etc. The two IR from ORCC and SLIM are used to build the final OpenForge Design class. The ORCC IR builds the input/output of the design, and allocates the necessary memory element. The SLIM IR builds the main scheduler of the actor by visiting all the actions firing rules and state machine if available. It then constructs the scheduling and firing policy for the actor. Xronos is also designed to automatically generate testbenches to aid in hardware simulation (used with the C code generator for input stimulus), and a performance report for each action, which includes the complexity (called gate depth), and the estimated latency (for static actions).

2.4 Analyzing CAL programs

The purpose of analyzing dataflow programs is to discover design bottlenecks. In large and complex applications, the main problem is often in finding functions or regions that need to be optimized in order to have the best speed-up for a minimum effort. In this context, the first tool that was developed to profile CAL programs is known as the *CAL Design Suite* [83], where the analysis is mainly performed on the executable software code from the generated C code since the target platform is on multicore processors. For hardware platform analysis, the tool had been extended to include hardware specific profiling information, i.e. the firing latency of each action. Since CAL Design Suite was to some extent specific to a software platform, a new framework called TURNUS was developed to profile directly at the dataflow program level via a functional simulation. At this level, evaluation and validation are performed in a completely platform agnostic environment. Figure 2.7 presents the TURNUS framework in relation to the design methodology. It takes as input, the application model (CAL) and the architecture, and also implementation specific information such as bit-accurate clock cycles, execution times, etc. The TURNUS profiler generates the so-called causation trace (explained in the next chapter), which is used as the main tool for program analysis of the following profiling features: computational load, critical path measurement, buffer size minimization and optimization, and multi-clock domain (MCD) partitioning. Based on these information, a design can be mapped and explored for various partitioning, scheduling, and buffer dimensioning. The following presents a formal description of the causation trace and its analysis for critical path and computational load reduction. The TURNUS approach for minimizing and optimizing the buffer size is given in chapter 5, while the MCD partitioning is not considered in this work, but briefly described in Section 2.5.2.

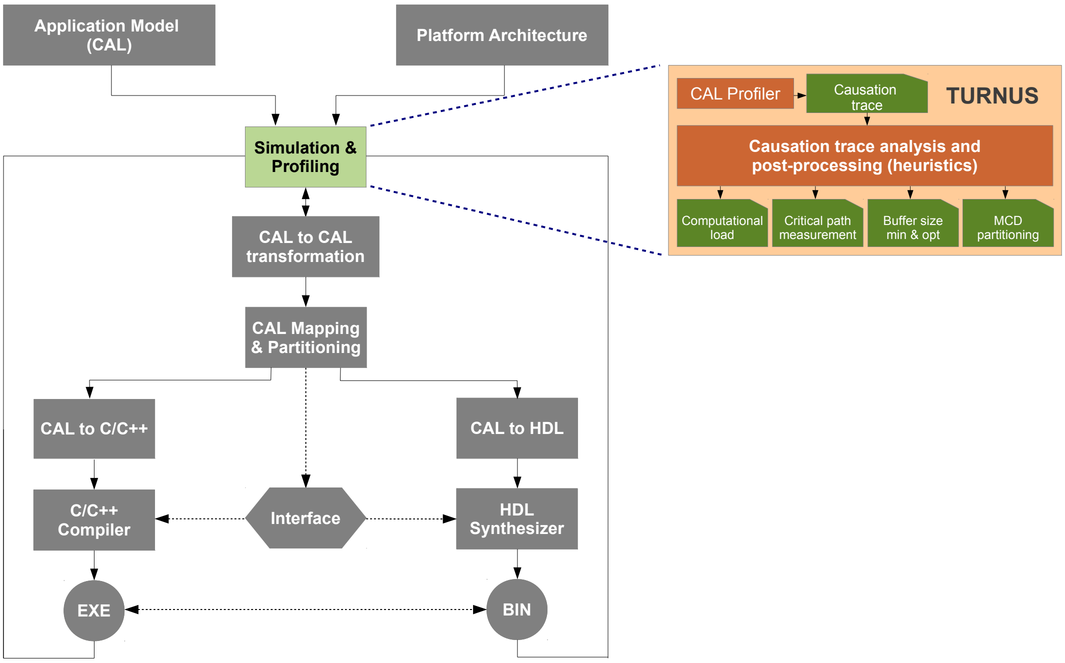


Figure 2.7: The TURNUS framework within the design flow for profiling dataflow programs.

2.4.1 Causation trace

The results from profiling dataflow programs is the creation of all executed actions with their dependencies that fully describe the program behaviour (with implication of relevant input data), known as the *causation trace* (or simply trace). Figure 2.8 shows an example of a trace. As given in [63], a causation trace, is a multi-directed acyclic graph $\mathbf{G}(V,E)$. Each single firing

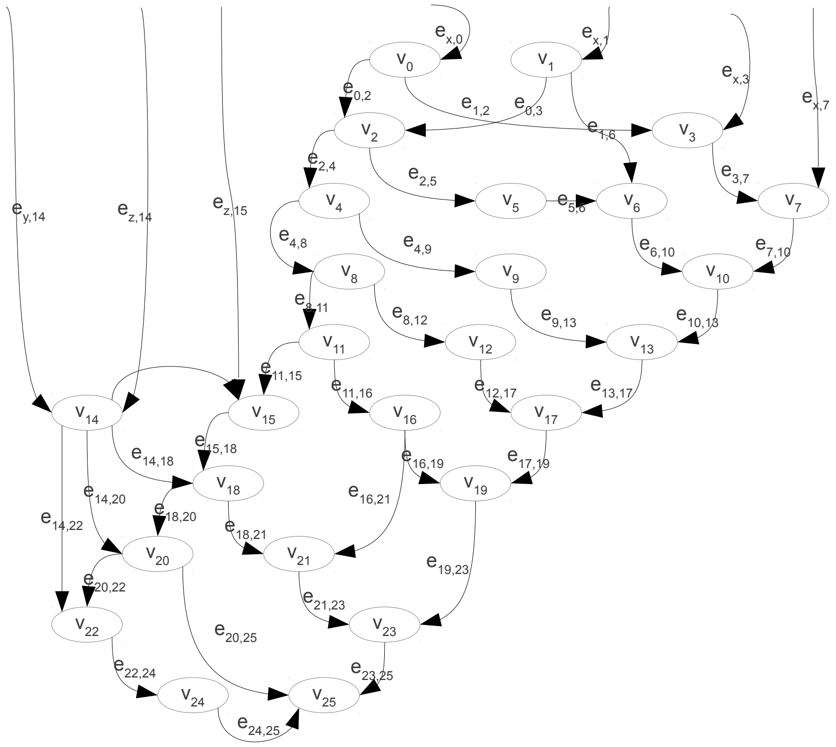


Figure 2.8: Example of a causation trace, where a node represents a single action firing, and an edge represents a firing dependency.

of an action $\tau \in \mathcal{T}$ is represented by a node $v_i \in$ V. Thus, the set $\mathrm{V}^{\tau} \subseteq \mathrm{V}$ contains all the firings of the action τ. Moreover, each single dependence between two fired actions is represented by a directed arc $e^{n}_{i,j} \equiv (v_i, v_j)_n \in$ E. The latter defines an execution order $v_i \prec v_j$, meaning that the execution of v_j depends on the execution of v_i. It follows that V can be considered as a partially ordered set of executed actions. Indeed constructing a consistent dependencies set E is fundamental in order to define constraints on the execution order between any couple of fired actions describing a platform-independent design behavior.

Weighted execution trace

The trace can be extended to a weighted execution trace, where the weights are defined as w_{v_i} and $w_{e_{i,j}}$ for each executed action $v_i \in V$ and dependency $e_{i,j} \in E$ according to the architecture model where each actor is supposed to be mapped. Weights can be estimated with two different levels of abstraction and accuracy: a) abstract profiling counts how often a set of basic operators (arithmetic-logic operation, flow control, memory access) are used during each step; b) platform-specific profiling extracts information from an HDL simulation tool for hardware implementations or by using standard software profiling tools.

Once weights have been evaluated, the total computational load of an action τ is defined as:

$$\mathrm{cl}_\tau = \sum\{w_{v_i} | v_i \in \mathrm{V}^\tau\} \tag{2.1}$$

Likewise, the computational load of an actor $a \in \mathscr{A}$ is defined as:

$$\mathrm{cl}_a = \sum\{\mathrm{cl}_{\tau_i} | \tau_i \in \mathscr{T}_a\} \tag{2.2}$$

And finally, the overall computational load of the actors set $\mathscr{A}$ is defined as:

$$\mathrm{cl}_\mathscr{A} = \sum\{\mathrm{cl}_{a_i} | a_i \in \mathscr{A}\} \tag{2.3}$$

For the purpose of profiling dataflow programs for hardware implementation, a script has been developed to automatically extract the weights (i.e. action firing latency in terms of the number of clock cycles) of each executed action from a Modelsim simulator. The list of all executed action with their associated weights are then organized to create an XML file with mean weight $\overline{w^\tau}$ for action τ. This file is given to TURNUS for further hardware implementation specific analysis and evaluation.

2.4.2 Critical path analysis and evaluation

The critical path (CP) is essentially the longest weighted path from source to sink nodes in the causation trace. For example, the CP in the trace given in Figure 2.8 is shown in Figure 2.9, where the CP is defined from node v_0 to v_{25}. The CP length can be evaluated by post-processing the weighted execution trace where the design configuration has been taken into account. for an augmented graph is defined as $\mathbf{G}(\tilde{\mathrm{V}}, \mathrm{E})$ such that $\tilde{\mathrm{V}} \supset \mathrm{V}$ and $\tilde{\mathrm{E}} \supset \mathrm{E}$, where two fictitious nodes are added. These are respectively the source node v_S and sink node v_T (both with weight $w_{v_S} = 0$ and $w_{v_T} = 0$. All the nodes v_i without incoming edges (i.e. $\delta^-_{v_i} = \emptyset$) are connected to v_S with a fictitious connection $e_{s,i}$ with weight $w_{e_{s,i}} = 0$. The same is done for all the nodes v_i without outgoing edges (i.e. $\delta^+_{v_i} = \emptyset$) where they are connected to v_T with a fictitious connection $e_{i,T}$ with weight $w_{e_{i,T}} = 0$. The topological order of the nodes

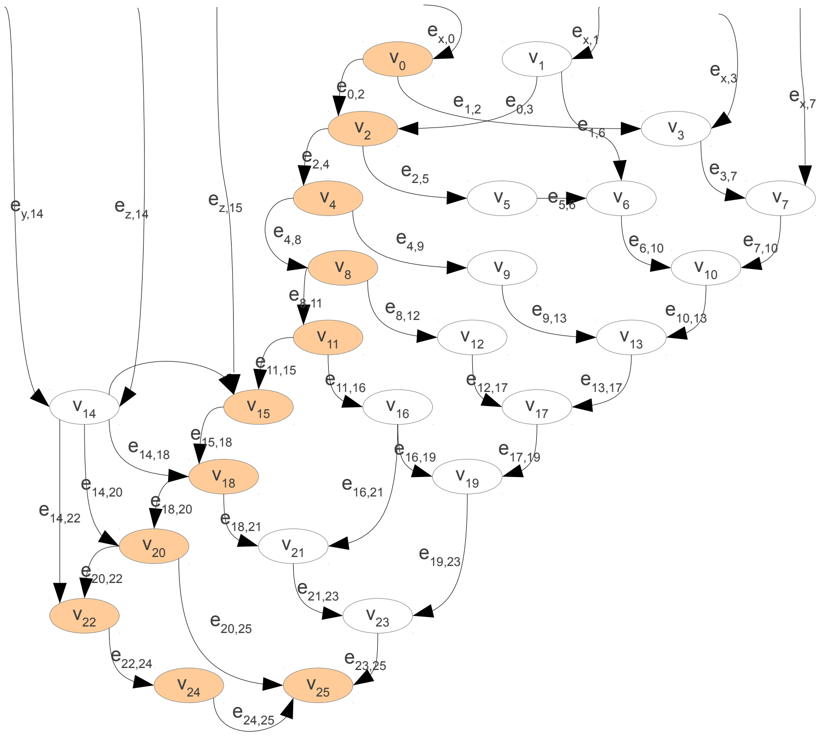


Figure 2.9: Example of a critical path in the causation trace, shown by the orange/shaded nodes from v_0 to v_{25}.

$v_i \in \mathrm{V}$ remains the same by assigning to v_S and v_T respectively by the lowest and the highest topological index of $\tilde{\mathrm{V}}$ such that $v_S \prec v_i \prec v_T$.

2.4.3 CP evaluation technique

The critical path can be evaluated in different ways [122, 16, 83]. Indeed the technique proposed in [16] seems to be the more convenient both for the reduced complexity of the algorithm and for the additional profiling information that could be retrieved. For each node v_i of the causation trace, four new parameters are defined:

- The **Early Start time** ES_{v_i} defines its earliest possible starting execution time.
- The **Latest Start time** LS_{v_i} defines its latest possible starting execution time without extending the overall program completion time.
- The **Early Finish time** EF_{v_i} defines its earliest possible ending execution time.
- The **Latest Finish time** LF_{v_i} defines its latest possible ending execution time without extending the overall program completion time.

Moreover an additional parameter called **Slack** is introduced both for nodes and dependencies represented respectively by S_{v_i} and $\mathrm{S}_{e_{i,j}}$. This is used in order to define the maximum delay that an executed action or a dependence can tolerate without impacting the overall completion time. In order to extract the critical path, the graph is walked back starting from v_T : at each iteration a new node $v_j^* \in \mathrm{V_c}$ is reached by following one of the incoming critical edges $\delta_{v_i}^{*-} = \{e_{j,i}^* \in \delta_{v_i}^- \cap \mathrm{E_c}\}$. The sets $\mathrm{V_{CP}} \subseteq \mathrm{V_c}$ and $\mathrm{E_{CP}} \subseteq \mathrm{E_c}$ contain respectively the executed actions and dependencies along this path. It follows that, for each action $\tau \in \mathscr{T}$, the set of its critical executions can be defined as $\mathrm{V}_C^\tau = \{v_i \in \mathrm{V}^\tau \cap \mathrm{V_C}\}$. Similarly, the set of its executions along the critical path as $\mathrm{V}_{CP}^\tau = \{v_i \in \mathrm{V}^\tau \cap \mathrm{V_{CP}}\}$. So as, the set of critical actions is defined as $\mathscr{T}_C = \{\tau \in \mathscr{T} | \mathrm{V}_C^\tau \neq \emptyset\}$, and the set of actions along the critical path as $\mathscr{T}_{CP} = \{\tau \in \mathscr{T} | \mathrm{V}_{CP}^\tau \neq \emptyset\}$. The critical path can be considered completely determinate only when v_S is reached. As a result, the critical path length is defined as $\mathrm{CP} = \sum\{w_{v_i} | v_i \in \mathrm{V}_{CP}\} + \sum\{w_{e_{i,j}} | e_{i,j} \in \mathrm{E}_{CP}\}$. It also must be noted that $\mathrm{CP} = \mathrm{LF}_{v_T}$.

2.4.4 Computational load reduction

CP analysis only provides a list of actions and actors that are in the most serial path of the design; Here the analysis has been extended to determine the impact of reducing the computational load of an action or a group of actions on the overall design performance, i.e. the reduction in $\Delta\mathrm{CP}$ that could be obtained by reducing the computational load of τ_*.

The computational load ratio of an action τ is defined by:

$$\ell_\tau = \frac{\mathrm{cl}_\tau}{\mathrm{cl}_\tau^0} \tag{2.4}$$

where cl_τ and cl_τ^0 represent respectively the current and the initial computational load values. Hence,the computational load reduction ratio can be simply defined as $\Delta\mathrm{cl}_\tau = 1 - \ell_\tau$. The problem is then to find a configuration $\ell_{\tau_{\mathrm{CP}}} = \{\ell_\tau | \tau \in \mathscr{T}_{\mathrm{CP}}\}$.

For this purpose, the *Logical Zeroing* algorithm has been used, proposed in [54]. This method is based on an iterative heuristic algorithm where the most critical action τ_k is computed and its computational load is neglected at each step k. For every executed action $v_i^{\tau_k}$, the weight $w_{v_i^{\tau_k}}$ is reduced by the factor $\alpha < 1$ which is found using a binary search algorithm on the CP while maintaining τ_* as the most critical action.

2.5 Optimizing CAL programs

After identifying the critical actors and actions via a dataflow program analysis, the next step is to improve/optimize these actors and actions. One such way can be performed by modifying the architecture of actors or actions such that an improvement versus an objective function is achieved, called *refactoring*. Apart from this, the results from dataflow program analysis can also be used to improve other factors such as the buffer interconnection sizes, clock frequency for each actor, SW/HW partitioning, actor scheduling policy, etc. These factors when applied for optimization, do not change the architectural structure of the design, i.e. actor specifications remain the same, but improves design implementation by some *external* factors. It is important to note that the optimization techniques for dataflow programs are to some extent, platform dependent. An optimization technique applied for software implementation target may not result in any gain (or in some cases, inferior performance) for hardware implementation target and vice versa. This is due to the way that different platforms are structured in terms of their processing elements, and the way programs are executed. The following reviews some techniques for optimizing CAL dataflow programs for software and hardware implementation.

2.5.1 Optimizing for software implementation

Refactoring and optimization of dataflow programs was first introduced in [83], mainly for single and multi-core processor platform target. One of the most effective techniques to improve performance in single-core processors is by actor merging. Although the degree of parallelism is often reduced, performance is improved by the reduction in the communication cost associated with the FIFO buffer interconnections. More actors imply more interconnections, and higher cost of locking the FIFO for reading and writing operations. Merging actors together reduces the number of FIFO interconnections which reduces the communication cost. Actor merging technique is particularly suited for actors that are simple and at a low granularity level, when the communication cost dominates the processing cost.

Actor merging however, is found to be less effective for multi-core processor implementation since the degree of parallelism is reduced and the available processor cores could not be exploited. Therefore, refactoring for increasing the level of task and/or data parallelism was also proposed. The strategy was to expose parallelism at the highest granularity level, so as to exploit the parallel processing cores as efficiently as possible. In the case of MPEG-4 decoders, the serial decoding is refactored into separate parallel decoding for the Y, U, and V color spaces. The different parallel components can be partitioned into the different processor cores and executed in a parallel fashion. Fine-grain granularity parallelism in multi-core processor on the other hand, is not quite effective due to the difficulty in finding the optimum partitioning and scheduling of the parallel blocks on various processor cores.

Another strategy being proposed was to merge actions within an actor such that the total number of action firing is reduced. The result is that actions become larger and more complex, but the number of states and action calls can be reduced significantly. As mentioned in Section 2.3.1, the action scheduler for a given actor is called every time an action fires. If the scheduling evaluation condition in the actor is complex, for example with many different states, multiple guard checks, and priority conditions, then the scheduling may become the bottleneck of the actor. By simplifying the scheduling conditions with less number of action firing, performance can be improved significantly for both single and multi-core processor implementation.

2.5.2 Optimizing for hardware implementation

Dataflow program refactoring and optimization techniques for hardware implementation target has not been analyzed before in detail with experiments on complex design cases. It should be clear that some of the techniques proposed for software implementation are not suitable for the virtually unlimited processing units available in a hardware platform. This is the primary objective of this thesis: to provide some effective CAL dataflow program refactoring and optimization techniques for hardware implementation target. Ultimately, the final result in hardware is dependent upon the CAL design specification, and is affected by both the HDL code generated and the results of conventional synthesis, place and route in the hardware tool flow. An understanding of the way OpenForge generates RTL codes with its hardware structure can provide a good starting point to perform effective program optimizations. Such report on recommended coding practices is given in [97].

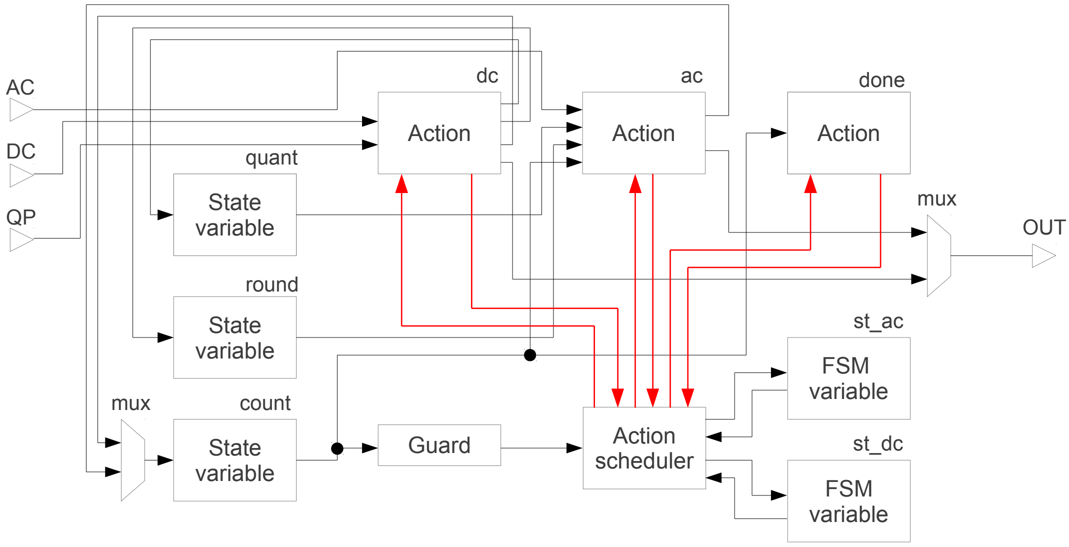


Figure 2.10: RTL architecture of the inverse quantization actor in Figure 2.3 using the OpenForge synthesizer. The red/bold interconnections are the control signals to signify the start and end of actions execution.

Figure 2.10 presents the RTL architecture for the `inverse_quantization` actor example given in Figure 2.3 using the OpenForge synthesis tool. State variables are mapped to registers with the corresponding size; actions and the scheduler to dedicated combinatorial logic. The action scheduler also takes as input the guard evaluation and the FSM states (with one register allocated per state). If an output port is used by more than one action (output `OUT` for action `dc` and `ac`), then a multiplexer is required to select the correct output at a given time. A multiplexer is also required for state variables that are updated in more than one action (variable count for action `dc` and `ac`). The multiplexers are controlled by the action scheduler. The key feature in this architecture is the interconnections shown in red/bold lines between the action scheduler and the actions. They are in fact the *go* and *done* signal pair for each action, where at a given time, the action scheduler determines *one* action to fire (based on FSM states, guards, and priorities) by sending the go signal for the action. Once the action has completed executing the algorithm in the action body, it sends the *done* signal to the action scheduler. The action scheduler in turn sends the new go signal for the next action to fire, and so on.

The resulting performance of the actor largely depends on this cycle, called the *scheduler-action* cycle; In order to obtain a high performance design, both the scheduler and the actions execution need to be as fast as possible. This depends on two factors: the clock cycle latency for the actions, and the operating frequency of the actor. In general, the clock cycle latency for action execution is one. It is also possible to obtain a zero latency action, for example an action without a body such as the action `done`. Actions execute for more than one clock cycle in the following cases:

- **Memory read/write**. This typically requires an additional two clock cycles for every access to a memory element.
- **Multitoken read/write**. This requires one additional clock cycle for every token that needs to be read/write.
- **Division**. Division algorithm that utilizes loops require additional clock cycles that depend on the operands of the divisor.
- **while/for each loops**. This requires additional clock cycles that depend on the number iterations for the loop.
- **Guard evaluation with the above**. Any complex guard evaluation increases the latency for the action scheduler to select an action to fire.

Another important point to note here is that for a given actor, actions execute in series, one after another as determined by the scheduler. If an actor contains many complex actions, then an effective strategy is to re-factor complex actions into separate actors so that the complex actions could now execute in parallel with actions from the original actor. This is the key concept in our methodology for exploiting data and task parallelism in chapter 3. Another effective strategy concerns with actions that perform multiple access to a memory element. Memory optimization techniques can be applied to these actions so that the clock cycle latency for the actions can be reduced.

Moreover for memory element utilization, resource can be traded-off with performance by defining memory size/length in the power of two so that address calculation can be streamlined. If memory size is not defined as such, additional arithmetic operations may be required to calculate the addresses, which could reduce performance, but allows smaller memory size. Similarly, OpenForge currently does not optimize memory and variable bit width. The allocation of arithmetic resource (adder, multiplier, etc.) depend on the bit width sizes; If a variable is defined for sizes that are larger than required, then the larger arithmetic resource will be allocated that not only increase the implementation area, but may also reduce performance by using larger units.

The second factor that impacts the performance of the scheduler-action cycle is the operating frequency. The maximum operating frequency for a hardware design is defined by the longest combinatorial path. For a given actor, this path is typically found on the most complex

```
//original - action body with common subexpression b * 13
a = b * 13 + c;
...
d = e - b * 13;

//improved - local variable to hold the common subexpression
x = b * 13;
a = x + c;
...
d = e - x;
```

Figure 2.11: Example to hold common subexpression in a local variable to reduce hardware overhead. The improved implementation requires only a single multiplier, as opposed to two multipliers in the original implementation.

action in terms of the combinatorial logic. In order to obtain a high maximum operating frequency, actions should be kept as small as possible. This can typically be achieved by refactoring/partitioning the complex action into several smaller actions. This is the key concept in our pipelining methodology in Chapter 4.

There are also other design strategies that can be applied to improve the performance of an action execution. For instance, by using a local variable for common subexpressions. Currently, common subexpressions are not searched and optimized. The strategy is to use local variables to hold the common subexpression results for reuse when an action has a common subexpressions that appear in more than one place. Unlike state variables, introducing local variables causes no hardware overhead. An example of this is given in Figure 2.11. Another recommendation based on the report in [97] is to use native CAL constructs such as FSM specifications, guards, and priority statements over standard programming constructs such as `if` and `while`. This is due to the architecture of the code generator that is designed to process these language specific statements more efficiently. In this case, the gain is on the resulting hardware resource.

Another strategy to reduce hardware resource is to introduce and apply resource sharing techniques. One of the approaches is as follows. In the current implementation, any function or procedure calls within an action are in-lined, i.e. a dedicated resource is allocated for the function or procedure. If two actions in the actor calls the same function or procedure, then the implementation would result in a redundant duplicated resource, since these actions never execute in parallel. The strategy here is to implement the function or procedure as an action that is called by the finite state machine, thereby creating only one instance of the function or procedure. This approach is quite trivial and is applied in the present work, although only a minor reduction in resource is observed. Another direction for resource sharing is at the lowest granularity level within an action. In this case, the objective is to share arithmetic operators. This approach is more challenging to implement and currently not possible to be implemented at the dataflow program level, but only at the code generation level. It is currently not being considered since it falls outside the scope of this work.

At the higher granularity level than an individual actor, the following optimizations can be made. First is FIFO buffer interconnection size for each communication channel. For hardware designs, this is a very important aspect in that it impacts not only the resource usage, but also functionality and performance. Buffer size selection that are too small may cause the system to deadlock, while those that are too large may be inefficient in terms of resource. The methodology to minimize and optimize buffer interconnection size is one of the objectives of this thesis, given in Chapter 5.

Secondly, another effective strategy is to analyze and optimize the execution of adjacent actors. This is sometimes referred to as cross-actor optimizations. It involves analyzing the behavior of two adjacent actors, and determining if actions can be re-factored by merging or partitioning some or all of the algorithm in the action body into different or existing actors, with the objective of improving one or more of the design criteria. This strategy has been identified before in [64] as possible optimization strategy, but has never been implemented, analyzed and explored in detail.

The final strategy that is discussed here is the multi-clock domain implementation. This is another powerful technique in which different actors are assigned different clock domain with the purpose of reducing the average clock cycle speed. This is a very important objective for reducing the dynamic power consumption. The hardware code generators CAL2HDL and Xronos already supports this implementation where adjacent actors with different clock domains are interconnected with an asynchronous FIFOs, instead of the usual synchronous one. Indeed, several works have explored this possibility, with [103] presents an architecture to exploit the so-called Globally Asynchronous Locally Synchronous (GALS) design by introducing a clock manager to each clock domain. Another work in [9] aims to find the optimum clock frequency for each actor such that the makespan of the overall design remains constant, compared to the design with a single clock domain. It should be noted that the two optimization strategies (cross-actor and multi-clock optimizations) are not implemented in the present work due to the scope and time limitation, but remains an interesting future direction.

2.6 Conclusion

In this chapter, a design flow for a complete software/hardware co-design using CAL dataflow programming has been presented, from specification to implementation, with program mapping and partitioning, high-level synthesis, analysis, and optimization. The design flow is constructed in the ORCC framework, that allows dataflow program specification and simulation using the CAL language, automatic synthesis of *all* language constructs to various implementation languages including for software and hardware, and program analysis for profiling and bottleneck detection. These tools within ORCC have been validated in this work, and have proven to be effective, and work seamlessly even for very complex applications. However, what is lacking from the design flow is the optimization tools and techniques that can be applied on the dataflow program, especially for hardware implementation target. This is a very important aspect in the design flow, where a design most likely has to be iterated several times in order to achieve the desired performance.

The following is the primary contribution of this thesis: new optimization tools and techniques that a CAL designer could use in order to improve and optimize designs at the dataflow program level. The first one is given in the next chapter, in Chapter 3, a refactoring technique to minimize system latency by exploiting task and data parallelism, and memory access reduction. This is followed in Chapter 4 for the refactoring technique to maximize operating frequency by action partitioning with a timing constraint and a minimum resource objective. The final technique in Chapter 5 aims to minimize resource by reducing and optimizing the buffer interconnection sizes between actors, which include two different approaches: on the hardware execution level, and on the dataflow program level. In Chapter 6, complex design case studies are described, including two MPEG-4 video decoders: the MPEG-4 Part 2 Visual Simple Profile (SP) and the MPEG-4 Advanced Video Coding (AVC)/H.264 Constrained Baseline Profile (CBP) decoders. In Chapter 7, these video decoders are used as case studies to explore and evaluate design alternatives, obtained by appropriate combinations of the refactoring and optimization techniques.

3 Minimizing system latency with refactoring

Minimizing system latency is a very important objective in systems design. Latency is defined as the number of clock cycles to process a given set of computational elements. This chapter presents some refactoring techniques and design architectures to achieve this objective of low latency, as applied on high-level dataflow programs for hardware implementation target. Two strategies are proposed to reduce system latency: 1) by exploiting data and task parallelism, and 2) by reducing the number of access to memory. We first describe how these methods can be applied for CAL programs in the general case, and present some remarks on automating the refactoring process. We then apply the TURNUS profiling tool to find the critical actors in the MPEG-4 AVC/H.264 decoder case study such that the refactoring techniques can be applied most effectively. These techniques are shown analytically and experimentally to be very effective for improving system throughput, and could be applied seamlessly on the high level program.

It should be noted that in this chapter, the refactoring techniques make a fairly extensive use of some of the actors in the MPEG-4 AVC/H.264 decoder case study as examples. Since the main focus of this chapter is on the refactoring techniques, we defer the description of the MPEG-4 decoder case study and its CAL specification in chapter 6.

3.1 Background and related works

Techniques for reducing system latency by exploiting parallelism and optimizing memory access in dataflow programs have been reported in literature. For example in [17] and [35] where CAL has been used to design and implement motion estimation algorithms in video encoder and decoder respectively. Both designs utilize algorithms that are naturally parallel (full-search and diamond-search), therefore allows explicit specification of data and task parallelism. In terms of memory usage, both use a cache-based approach, where a frequently used search window is stored in a local buffer so that it does not have to be fetched from memory each time. The problem in these approaches is that they are algorithm-specific and could not be generalized to any complex algorithms, i.e. a non-search based algorithm. The work in [110] proposes an automated technique to determine the possible parallelism factor of a given algorithm, and then generates RTL code for a given resource constraint. However, the implementation is restricted to synchronous dataflow (SDF). Furthermore, the case studies were only made for very small examples such as an adder, multiplier and second order FIR and IIR filters. The technique has no guarantee that it will work for dynamic dataflow actors with complex control and structures. The work in [46] describes several ways of how concurrency can be exploited in dataflow programs, including how system latency can be improved by reducing the number of access to memory. However, the case study was mainly targeted to multi-core platforms where this issue has only minimal effects on the overall performance.

Since the techniques and architectures presented in this chapter are to some extent, specific to MPEG decoders, it is worthwhile to review some of the related works in parallel architectures and memory optimizations in the decoders. Most of the works in literature focus on optimizing the decoding process itself where there are high potential for parallelism, and where most of memory access are made, i.e. intra and inter predictions for reducing spatial and temporal redundancies. Since predictions in video decoders are processed in blocks, various parallelism techniques can be exploited in the algorithm. Moreover, for inter prediction that utilizes past frames to decode a current frame, large amount of memory bandwidth is required to store and retrieve the frames.

The following presents some techniques on exploiting data and task parallelism on MPEG decoders as reported in literature. The work in [111] presents a coarse-grain parallel processing of luma and chroma intra blocks with group-based write-back approach to fully utilize the parallel blocks. In a slight contrast, the approach in [79] attempts to rearrange the equations in the intra prediction algorithm so as to obtain efficient parallel hardware implementation. This is achieved by simplifying the equations in the standard reference software, and instantiating multiple parallel prediction mode blocks. This approach is also quite similar to the work in [112], for optimizing the inter prediction process where the 6-taps half pixel interpolation filter equation is simplified to facilitate parallelism. Furthermore, the level of parallelism in inter prediction can be increased by having parallel fractional pixel filtering, such as the work in [81] with four parallel units. Besides prediction units, there are also works on increasing the parallelism of the deblocking filter unit, such as the works in [72] and [92] where six and two parallel filters are used respectively. In [72], the six filters implement independent memory for each line of 4x4 block to achieve a latency of 49 clock cycles per macroblock, while in [92], the two parallel units process horizontal and vertical edges simultaneously with overall latency of 112 clock cycles per macroblock.

In terms of memory optimizations, the objective is mainly to minimize the bandwidth such that the required throughput is guaranteed; this is achieved by making sure that no redundant data is loaded for macroblock processing. In [80], the following strategy is proposed: data for pixel interpolation is loaded specifically for a particular block size and fractional position type, and data is reused whenever possible for vertical and horizontal filtering. This results in an average of 70%-80% reduction in the required bandwidth. The work in [125] on the other hand, proposes a novel 3-D cache where the memory index is composed of vertical and horizontal positions of the requested area within a frame, with 3 concatenated tags of picture order count number, row, and line positions. The authors reported a bandwidth reduction of roughly 62%.

It should be noted that most of the work in implementing parallelism in video decoders for hardware implementation utilize low-level RTL languages. Although effective, the task of minimizing system latency can be difficult and time-consuming, and often very hard to find system bottlenecks, especially for complex systems. In contrast, dataflow programming language is designed for having higher degree of analyzability by focusing more on the algorithm by means of a dataflow process, and hiding low-level hardware implementation details such as register allocation, bit-accurate memory access, and gate-level control. This greatly simplifies the process of identifying critical parts in the system, and simplifies the way to implement data and task parallelism and memory optimizations.

3.2 Minimizing system latency in CAL programs

3.2.1 Task and data parallelism

Possibly the most common method for minimizing latency in hardware designs is by exploiting task and data parallelism. In this work, *task* parallelism refers to the execution of a given task in a concurrent fashion, where the task is *partitioned* across several parallel subtasks. In order for the subtasks to execute in parallel, they should not have any precedence relations, i.e. each subtask should have no dependencies with any other subtask. If there are dependencies between the subtasks, then it refers to pipeline parallelism, which is described in chapter 4. In contrast, processing with *data* parallelism refers to the execution of several similar tasks in a concurrent fashion, where the task is *replicated* several times. In this case, input data is partitioned accordingly and sent to each of the replicated task. Similar to task parallelism, in order for the tasks to execute in parallel, they should not exhibit any precedence relations. Figure 3.1 shows timing diagram comparisons between processing sequentially, with task parallelism, and with data parallelism. The latency to process task *B* can be reduced by up to 3x when exploiting data or task parallelism.

Naturally, task and data parallelism are most effective to be applied on parts of the program that are long, complex, and are processed in serial. In dataflow programs, these can be

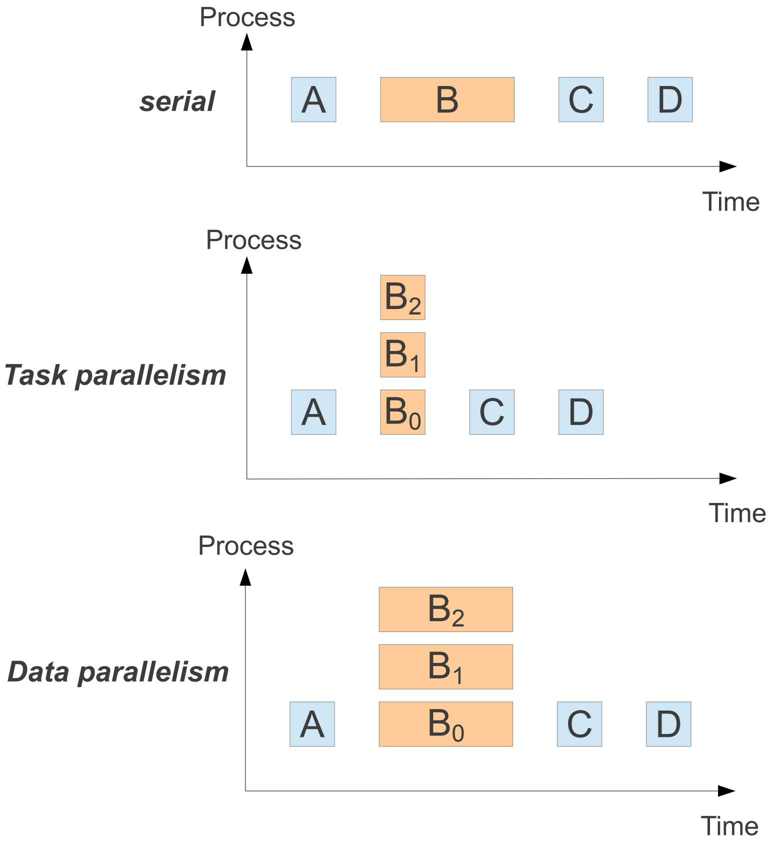


Figure 3.1: Comparison between processing with sequential, task, and data parallelism. In task parallelism, the task B is partitioned into distinct tasks B_0, B_1, and B_2, while in data parallelism, B is replicated 3-times to form B_0, B_1, and B_2

.

easily identified in actions where extensive computations with many clock cycles are required to complete the firing. In a complex dataflow network, the relevant actions can be found automatically using a dataflow program analysis tool TURNUS as explained in Section 2.4. Using this tool, actions that are in the most serial path of the program can be found. This serial path can of course be reduced by increasing the degree of parallelism of the critical actions, i.e. by reducing the number of clock cycles required to fire critical actions.

The choice of performing task or data parallelism largely depends on the algorithm in the action body. Task parallelism is more suited to algorithms that perform *irregular* computations, i.e. extensive computational requirements that are not repetitive and not fired continuously. Partitioning this type of action into subtasks results in latency improvement in a single action firing. On the other hand, data parallelism is effective when the action is being fired continuously with *regular* computations. In this case, replicating the complex action into separate actors such that multiple actor instances could execute in parallel would result in significant latency reduction because of the continuous firing. It is interesting to note that task parallelism is also effective to be performed for actions with regular computations, but not the other way around. If the computation is irregular, the replicated action in another actor instance most likely would not be fired in parallel with another actor instance, therefore only results in a redundant additional resource. The detection for the type of critical action for regularity is quite trivial and could easily be integrated with TURNUS.

Once the critical action is found and the parallelism technique has been chosen, the next step is to apply the technique on the action. Given a critical actor $a \in A$ with critical action τ_{a_i} where $i \subset \{1 \dots k\}$ for k actions in actor a, the following describes the necessary steps for performing task and data parallelism:

1. ***Partition/replicate critical action.*** For task parallelism, the task in action τ_{a_i} is partitioned into M subtasks with actions $\tau_{a_{i1}} \dots \tau_{a_{iM}}$. The technique to partition the action can be performed manually, or automatically with an optimization objective, such as the one given in chapter 4. In order to obtain a true task parallelism, the subtasks should not have any dependencies among each other. For data parallelism, the action τ_{a_i} is replicated N times to obtain actions $\tau_{a_{i1}} \dots \tau_{a_{iN}}$, without a feedback dependencies among the replicated actions.

2. ***Create parallel actors.*** The partitioned/replicated actions $\tau_{a_{i1}} \dots \tau_{a_{iM}} / \tau_{a_{iN}}$ are extracted and transformed into actors by appropriate wrapping with the relevant inputs and outputs. The key idea here is to instantiate the actors $a_0 \dots a_M / a_N$ such that they can be executed in parallel for as much time as possible. Since actions are fired if there are sufficient input tokens, maximum parallelism among the actors can be obtained by sending the input tokens in parallel to the actors.

3. ***Manage the split and merge actors***. Another key feature when applying task and data parallelism is the split and merge actors. The split actor is used for splitting the original serial input data to actor a into parallel input data to actors $a_0 \ldots a_M/a_N$. The merge actor is used for merging the output from the parallel actors such that the correct output sequence is obtained. In some cases, these actors do not have to be explicitly defined, but can be integrated in the adjacent actors in the dataflow network.

Figure 3.2 summarizes the steps for implementing task and data parallelism starting from a critical action. In the current implementation, the detection of a critical action, its suitability for task and/or data parallelism, and action partitioning and replicating are performed automatically, while the actual process of implementing task and/or data parallelism from creating parallel actors, and managing the split and merge actors are performed manually. In Section 3.2.3, we present some remarks on fully automating the refactoring process.

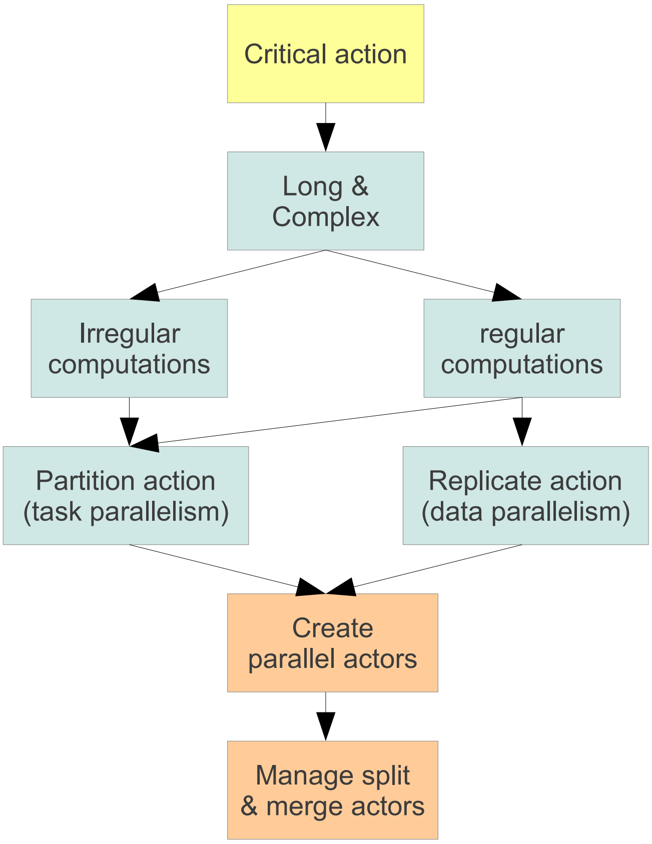


Figure 3.2: Methodology to implement task and data parallelism for a critical action.

.

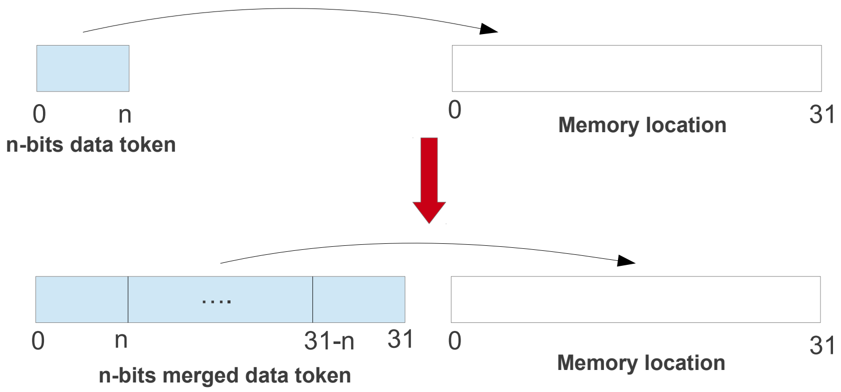


Figure 3.3: With (bottom) and without (top) token merging before a write access. The number of access to memory can be reduced significantly with merging the data tokens before accessing the memory.

3.2.2 Reducing number of memory access

Another very effective method at reducing system latency at the architectural level is by reducing the number of access to memory. Given a task C that needs to access memory for M number of times, by reducing the number of access by N, the latency to process task C can now be reduced proportionally to $M - N$. There are several ways that memory access can be optimized. In this work, we aim to reduce the access by 1) packing data tokens before a memory access, which we call *Data-packing*, and 2) eliminating any redundant memory access in the program, which we call *Redundancy-elimination*.

The data-packing technique aims to pack/merge data tokens before a write access, with the corresponding split of the merged tokens when performing a read access. The main idea here is to exploit the lower latency to pack tokens together compared to accessing the memory for every token. Moreover, the latency to access memory is typically independent of the word size, i.e. writing 1 byte or n bytes to a single memory location requires the same latency. As shown in Figure 3.3, n-bits data token can be packed into a 32-bit word typically used in a memory element such as an external RAM. The number of packed n-bits data tokens in a 32-bit word can be computed as $floor(\frac{32}{n})$. If $n = \{1, 2, 4, 8, 16, 32\}$, we fully utilize the bits in the memory location, while all others for $n < 32$ would result in a storage of packed data tokens of less than 32-bit. When individual tokens of size n needs to be read, the memory location and its sub-location has to be accessed and masked correctly.

The redundancy-elimination technique aims to reduce the number of access to memory by removing any unnecessary intermediate access. This is particularly useful for actors that require sending output results after processing. The results after processing an algorithm

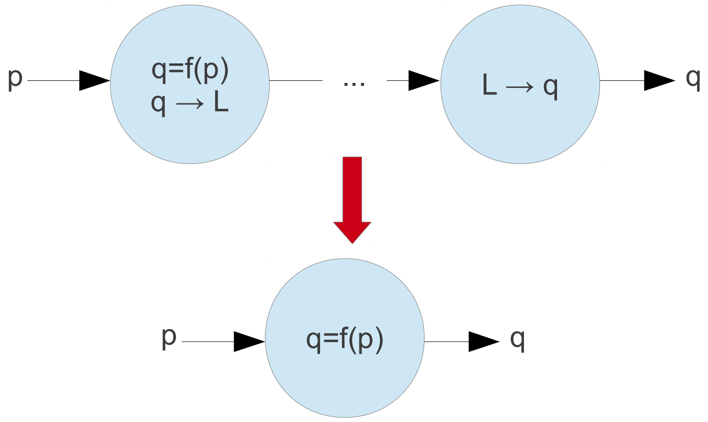


Figure 3.4: With (bottom) and without (top) the redundancy-elimination technique. The intermediate storage buffer L can be eliminated for a significant reduction in latency.

in a given action is sometimes stored in a temporary buffer, and later sent to the output port by a different action that simply outputs the content from the temporary buffer. If the sequence of data storage *after* processing is the same as the sequence of data output from the storage element, then the results could be sent directly during processing, without requiring an intermediate buffer. This technique is very effective in that large number of memory access can potentially be eliminated, and therefore reduces the latency significantly. Figure 3.4 shows the technique with ovals as dataflow actions. The function $q = f(p)$ generates a set of data q from a set of input data p. In the implementation without redundancy-elimination, q is stored in a temporary memory element L, where the following action then sends q by reading from L. In the implementation with redundancy-elimination, q is sent directly to the output without storing to and reading from L.

Figure 3.5 shows the steps for applying the memory optimization techniques, starting from a critical action τ_{a1} obtained for example using TURNUS. If the action takes an n-bit data and stores each one to a memory element L, and $n < 32$, then the data-packing technique can be applied. The *floor*$(\frac{32}{n})$ data are first packed together and stored as a 32-bit word in a new memory element L_1 with a width of 32-bit. Because the data tokens are now packed together, each time that these tokens need to be accessed, L_1 has to be split, masked, and indexed correctly. As for the redundancy-elimination technique, it can be applied to a critical action τ_{a1} that produces data tokens from a memory element L. The first step is to detect if L is being written in another action, say for example τ_{a2}. The action τ_{a2} is analyzed if the writing/update sequence is similar to the output sequence in the action τ_{a1}. If so, then the output can be sent directly during the execution of action τ_{a2}, without requiring an intermediate memory element L. In this case, the action τ_{a1} can also be eliminated.

It should be noted that the techniques are mostly being performed manually, with only the detection part being done automatically. In the next section, we present some remarks on fully automating the refactoring process.

3.2.3 Automating the refactoring techniques

The refactoring techniques for parallelism and memory optimizations involve major modifications to the design architecture. For CAL programs that follows a DPN model of computation, the flexibility afforded by the model makes the task of automatically generating a parallel or a memory optimized architecture a very difficult, if not impossible task. This is because of a large number of design styles, parameters, and controls that could be implemented in the program that makes no guarantee if an automatic solution can be found. Even if an automatic structure can somehow be found, the efficiency is also not guaranteed, compared to a manual intervention where the most important feature can be controlled for maximum efficiency. Some of the specific issues are summarized below:

- For task and data parallelism, the difficult part is on creating an actor automatically for a partitioned/replicated action. This includes generating the relevant interfaces and controls, that also depends on adjacent actors. Finding an a generic automatic technique for generating these features with maximum efficiency is also a very difficult task.
- For the data-packing technique, the difficult part is when the merged data needs to be loaded for a read access. There are many different ways that a read access can be performed, either sequentially or randomly, and in any order. Finding a generic automated technique with high efficiency for all cases is also a very difficult task.
- For the redundancy-elimination technique, the difficult part to be performed automatically is analyzing the update sequence of the *processing* and *output* actions, since they can be implemented in a variety of ways for example using FSM, guards, or loop in the action body. Furthermore, with an almost infinite way of updating the sequence, it is very difficult to find a generic and efficient implementation that outputs the correct sequence from an arbitrary sequence.

The problem of automatically finding a program structure can be related to the Kolmogorov complexity theory [80], which mainly concerns with finding a minimum description for a given string of output values, given by:

$$C_f(x) = \min\{|p| : f(p) = x\} \tag{3.1}$$

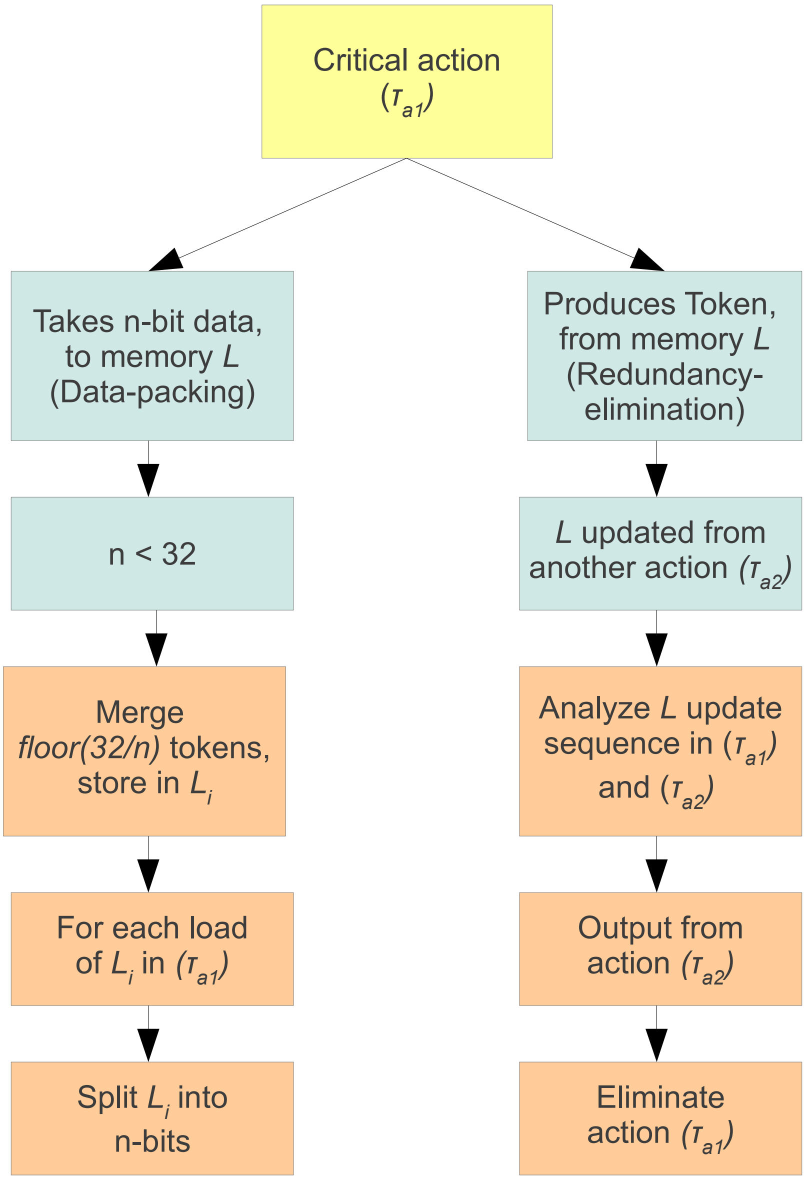


Figure 3.5: Methodology to implement the data-packing and the redundancy-elimination techniques for a critical action.

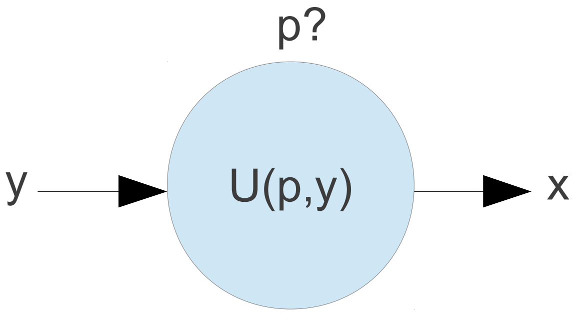


Figure 3.6: Finding a minimum program/structure p for a given input y and output x using a Universal Turing Machine U, based on the Kolmogorov complexity theory.

where the objective is to find the smallest $|p|$ to represent the output string x using a computable function f. For programs with input specifications, this can be extended to a *conditional* Kolmogorov complexity with a Universal Turing Machine U [70],

$$C_u(x/y) = \min\{|p| : U(p, y) = x\} \tag{3.2}$$

where the objective now includes the input y, as depicted in Figure 3.6. The key idea here is that if x can be described as a function, then the minimum p can be found. However, in systems design where x typically has the notion of being *random* (or incompressible), x could not be expressed as a function, where in this case, p can be proven to be incomputable [121]. Therefore, the program structure p could not be found automatically, but has to be implemented manually, similar to some parts of the refactoring techniques that have been presented.

3.3 Analyzing the MPEG-4 AVC/H.264 decoder using TURNUS

Before the refactoring techniques are applied, the first step is to analyze the dataflow program to find regions that are critical. For this purpose, TURNUS framework (described in Section 2.4) has been used to 1) determine the critical path (CP) in a complex dataflow network, and 2) to rank the critical actors. The computational load reduction technique is also applied to find the right refactoring directions. As case study, TURNUS has been applied on the MPEG-4 AVC/H.264 decoder design (Section 6.4).

Table 3.1: Results from profiling the original CAL description of the MPEG-4 AVC/H.264 decoder. The most critical actor is found to be the `half_quarter_interpolation` with almost 70% on the CP executions, followed by the `picture_buffer_y` with roughly 23%.

Critical actor	$\|\mathbf{V^a_{CP}}\|$	$\|\mathbf{E^a_{CP}}\|$	**CP contributions (%)**
half_quarter_interpolation	385669	1925548	69.7
picture_buffer_y	128306	506601	23.2
parser	37913	613065	6.9
blocks_reorder_y	1304	1298	0.2
deblocking_filter_y	27	31	0.0

Table 3.2: Logical Zeroing results for the intermediate version of the MPEG-4 AVC/H.264 decoder for reducing the overall CP by 15%. It consists of actors, actions, and the required computational load (CL) reductions.

Actor	**Action**	**Required CL reduction (%)**
half_quarter_interpolation_launch1	getpixval_done	30.0
half_quarter_interpolation_launch3	getpixval_done	30.0
half_quarter_interpolation_launch2	getpixval_done	28.5
half_quarter_interpolation_launch1	getpixvalquarter	15.5
half_quarter_interpolation_launch3	getpixvalquarter	9.8

Table 3.1 shows the top 5 critical actors with the number of critical execution ($|V^a_{CP}|$) and number of dependencies on the CP ($|E^a_{CP}|$) for a given actor a. The analysis was evaluated for 5 *Foreman* QCIF frames with total number of action execution, $|V| = 5251653$ and the total number of executions in the critical path, $|V_{CP}| = 553238$.

The computational load reduction technique has also been applied in order to find the right refactoring directions. Table 3.2 summarizes the results for one of the design alternatives of the MPEG-4 AVC/H.264 decoder. In order to obtain a 15% reduction in the CP, it is necessary to reduce the computational load of the actions in the list by the given amount.

The analysis above is compared to actual refactoring results of the critical actions in terms of the performance obtained on hardware. When the actors `half_quarter_interpolation_launch`**x** (for $\boldsymbol{x} = \{1, 2, 3\}$) are refactored by improving the average latency by 22%, an overall 15% latency reduction is observed. This agrees with the analysis that on average, 23% computational load reduction is required for the actors in order to obtain the same (15%) CP length reduction. Consequently, this also proves the accuracy of the analysis in relation to actual implementation performance.

3.4 Exploiting data and task parallelism on MPEG-4 AVC/H.264 decoder

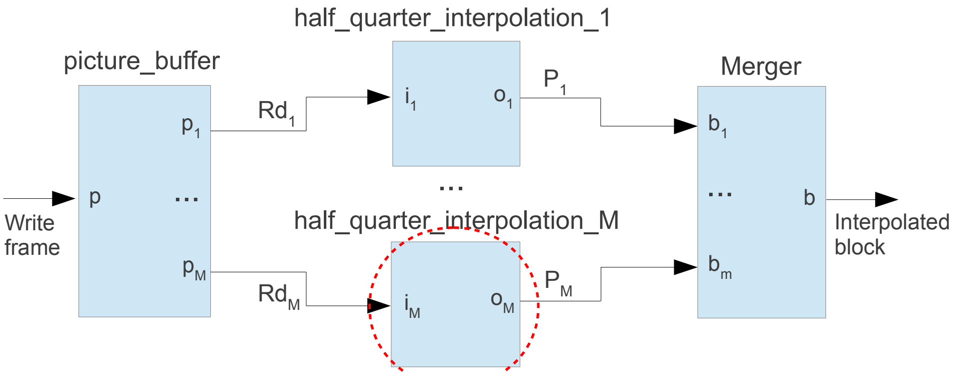
This section presents some examples of applying the task and data parallelism refactoring techniques presented in Section 3.2.1 on the design case study, based on the critical actors and optimization directions given by TURNUS.

3.4.1 The *half_quarter_interpolation* actor

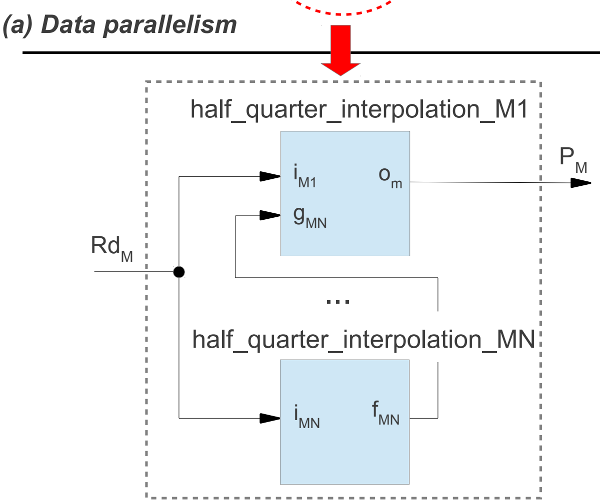
Based on the CP analysis in Table 3.1, this actor contributes the most to the CP, and is the most critical. Figure 3.7 shows how data and task parallelism can be exploited for this actor. In data parallelism (a), the video blocks are sent in an alternate fashion from the `picture_buffer` to the `half_quarter_interpolation` using M dedicated channels. Using this approach, M similar tasks are essentially executed concurrently, where the task (`half_quarter_interpolation`) is *replicated* M times. The input data (from `picture_buffer`) is partitioned accordingly by block and sent to each of the replicated task. In this case, the blocks vary in size where their dimension $W \times H \in \{4\times4, 4\times8, 8\times4, 8\times8, 16\times8, 8\times16, 16\times16\}$.

On the other hand, in task parallelism (b), a given task is executed in a concurrent fashion, where the task (`half_quarter_interpolation`) is *partitioned* across N parallel subtasks. Each subtask performs a different set of operations, which requires the final merging of results as shown by the interconnection between f_{MN} and g_{MN}.

For this specific actor, task parallelism is implemented as follows. The task of computing half and quarter pixels for a given video block $W \times H$ can be partitioned into 4 subtasks s_0, s_1, s_2, and s_3 with computational requirements for the worst case block size of 16×16 given in Table 3.3. Each subtask is also found to be independent of each other except for s_3, i.e. s_0, s_1 and s_2 can all be performed in parallel, followed by task s_3. The execution of subtasks for a given video block depends on the sample location defined in Figure 3.8, with the required subtasks summarized in Table 3.4. Hence, we can see that most of the sample location requires at least two subtasks which can be performed in parallel. Figures 3.9 and 3.10 respectively show algorithms for half and quarter pixel interpolation as implemented in CAL. For half pixel interpolation, the subtasks s_0, s_1, and s_2 are shown respectively at lines 1, 17, and 33. The functions *computeHalfPixel* and *sixTapFilter* in Figure 3.9, and *computeQuarterPixel* in Figure 3.10 respectively are based on equations 6.8, 6.8 without the *Clip* function, and 6.9.



(a) Data parallelism



(b) Task parallelism

Figure 3.7: (a) data parallelism, where the actor `half_quarter_interpolation` is replicated M times, and later merged. For each instantiated actor, it is possible to perform (b) task parallelism, where the task is partitioned into N subtasks with distinct set of operations.

Further analysis of the subtasks in Table 3.3 shows that the required number of memory access and arithmetic operations are still relatively large and being performed serially, especially for subtasks s_0 and s_1. Therefore, a higher degree of task parallelism can be obtained by further partitioning these subtasks. As shown in Figures 3.9 and 3.10, the subtasks contain a loop that performs 6-tap filter operation for x number of times, where $x = \{1312, 592\}$. The strategy here is to partition the loop by h, and create several parallel instances of this operation, such that each of them performs filtering for only x/h times.

The estimated latency reduction for both data and task parallelism will now be derived, based on the example above. Given B video blocks to be processed by the task (`half_quarter_-`

Table 3.3: The subtasks of MPEG-4 AVC/H.264 half (s_0, s_1, s_2) and quarter (s_3) pixel interpolation for worst-case video block and their complexity in terms of memory access and arithmetic operations. s_0, s_1 and s_2 can be performed in parallel, followed by s_3.

Subtask	**# of distinct memory access**	**# of 6-taps FIR filter**	**# of 2-taps FIR filter**
s_0	9184	1312	-
s_1	9184	1312	-
s_2	4144	592	-
s_3	768	-	256

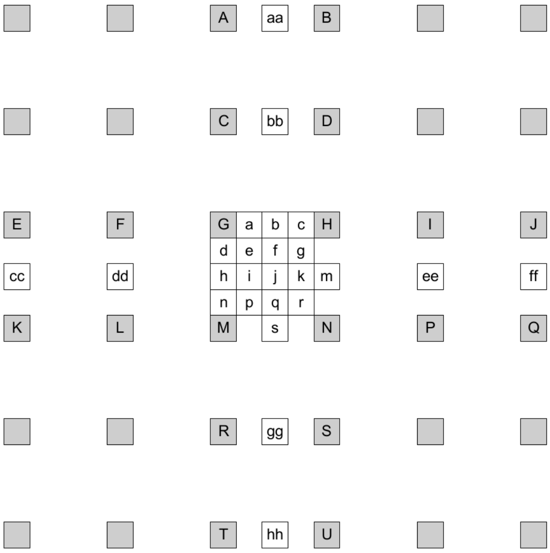


Figure 3.8: Integer sample (shaded blocks with upper-case letters) and fractional sample positions (unshaded blocks with lower-case letters) for quarter sample luma interpolation.

Table 3.4: Fractional sample positions based on Figure 3.8 and their required subtasks given in Table 3.3.

Fractional position(s)	**d,h,n**	**a,c**	**i,k**	**b**	**f,q**	**j**	**e,p,g,r**
Required subtask(s)	s_0	s_0, s_1	s_0, s_2	s_1	s_1, s_2	s_2	s_0, s_1, s_3

```
//subtask s0
if(xFrac = 0 or xFrac = 1 or xFrac = 3) then
 _x = xMin;
 while (_x < xMax+1) do
  _y := 0;
  while (_y < height) do
   tabPix[4*_x][4*(2 + _y) + 2] :=
      computeHalfPixel(tabPix[4*_x][4*_y], tabPix[4*_x][4*(1 + _y)],
            tabPix[4*_x][4*(2 + _y)], tabPix[4*_x][4*(3 + _y)],
            tabPix[4*_x][4*(4 + _y)], tabPix[4*_x][4*(5 + _y)]);
   _y := _y + 1;
   end
  _x := _x + 1;
  end
end

//subtask s1
if(yFrac=0 or (xFrac!=0 and yFrac=1) or (xFrac!=0 and yFrac=3)) then
 _x := 0;
 while (_x < width) do
  _y := yMin;
  while (_y < yMax+1) do
   tabPix[4*(2 + _x) + 2][4*_y] :=
      computeHalfPixel(tabPix[4*_x][4*_y], tabPix[4*(1 +_x)][4*_y],
              tabPix[4*(2 +_x)][4*_y], tabPix[4*(3 + _x)][4*_y],
              tabPix[4*(4+ _x)][4*_y], tabPix[4*(5 + _x)][4*_y]);
   _y := _y + 1;
  end
  _x := _x + 1;
 end
end

//subtask s2
if((xFrac=2 and yFrac != 0) or ((xFrac=1 or xFrac=3) and yFrac = 2)) then
 _x := 0;
 while (_x < width) do
  _y := yMin;
  while (_y < yMax+1) do
   tabPix[4*(2 + _x) + 2][4*_y] :=
       computeHalfPixel(tabPix[4*_x][4*_y], tabPix[4*(1 + _x)][4*_y],
            tabPix[4*(2+ _x)][4*_y], tabPix[4*(3 + _x)][4*_y],
            tabPix[4*(4+ _x)][4*_y], tabPix[4*(5 + _x)][4*_y]);
   _y := _y + 1;
  end
  _x := _x + 1;
 end
 _x := 0;
 while (_x < width) do
  _y := 0;
  while (_y < height + 5) do
   tabIntermPix[4*(2 + _x) + 2][4*_y] :=
    sixTapFilter(tabPix[4*_x][4*_y], tabPix[4*(1 + _x)][4*_y],
              tabPix[4*(2 + _x)][4*_y], tabPix[4*(3 + _x)][4*_y],
              tabPix[4*(4+ _x)][4*_y], tabPix[4*(5 + _x)][4*_y]);
   _y := _y + 1;
  end
  _x := _x + 1;
 end
end
```

Figure 3.9: Half quarter interpolation algorithm as implemented in CAL. The task is divided into three subtasks *s0*, *s1*, and *s2*.

```
//subtask s3
if(not(xFrac=0 and yFrac=2) and not(xFrac=2 and yFrac=0)
   and not(xFrac=2 and yFrac=2)) then
 x_idx := 0;
 while (x_idx < width) do
  y_idx := 0;
  while (y_idx < height) do
   pix1 :=
    if((xFrac=1 and yFrac<2) or(xFrac=2 and yFrac=1) or (xFrac=3 and
        yFrac=0)) then tabPix[4*(2 + x_idx)+2][4*(2 + y_idx)]
    else if(xFrac!= 0 and yFrac=3) then tabPix[4*(2 + x_idx) + 2][4*(3 + y_idx)]
    else if((xFrac=1 and yFrac=2) or (xFrac=3 and yFrac=2))then
     tabPix[4*(2 + x_idx)+2][4*(2 + y_idx) + 2]
    else if(xFrac=0) then
     if(yFrac=1)then tabPix[4*(2 + x_idx)][4*(2 + y_idx)]
     else tabPix[4*(2 + x_idx)][4*(3 + y_idx)] end
    else tabPix[4*(2 + x_idx) + 2][4*(2 + y_idx)]
    end end end end;
   pix2 :=
    if(xFrac=0 or (xFrac=1 and yFrac!=0)) then
     tabPix[4*(2 + x_idx)][4*(2 +y_idx) + 2]
    else if(xFrac=3 and yFrac!=0)then tabPix[4*(3 + x_idx)][4*(2 + y_idx) + 2]
    else if(xFrac=2) then tabPix[4*(2 + x_idx)+2][4*(2 + y_idx) + 2]
    else if(xFrac=1 and yFrac=0) then tabPix[4*(2 + x_idx)][4*(2 + y_idx)]
    else tabPix[4*(3 + x_idx)][4*(2 + y_idx)]
    end end end end;
   tabPix[4*(2 + x_idx) + xFrac][4*(2 + y_idx) + yFrac] :=
    computeQuarterPixel(pix1,pix2);
   y_idx := y_idx + 1;
  end
  x_idx := x_idx + 1;
 end
end
```

Figure 3.10: Quarter pixel interpolation algorithm as implemented in CAL. This task has to be performed after the completion of half quarter interpolation.

`interpolation`), the estimated latency for serial implementation is simply given by:

$$L = \sum_{b=0}^{B} (t_r^b + t_p^b + t_s^b) \tag{3.3}$$

where t_r^b, t_p^b, and t_s^b are the latency to receive, process, and send a single block $b \in B$. By implementing data parallelism with M replicated tasks, the latency is taken as the last instance to finish executing all of its given blocks, i.e. $max(L'_1, \ldots, L'_M)$, where L'_m, $m \in M$ is the estimated latency of instance m to process all its blocks B/M, i.e.

$$L'_m = \sum_{b=0}^{B/M} (t_r^{Mb+m} + t_p^{Mb+m} + t_s^{Mb+m}) \tag{3.4}$$

The superscript $Mb + m$ refers to a specific block for an instantiated task that depends on the number of instantiated tasks M, the current block b, and the instance sequence m. For example if $M = 2$, the instance $m = 1$ and $m = 2$ respectively process all odd and even numbered blocks. The latency reduction is therefore, $L - max(L'_1, \ldots, L'_M)$, where it is expected that $L'_m < L$.

The latency reduction for task parallelism will now be estimated. Given a block $b_{mn} \in B$ that is to be processed by instance m and subtask n, the latency for serial implementation is given by:

$$L = t_{b_{m1}} + t_{b_{m2}} + \ldots + t_{b_{mN}} \tag{3.5}$$

where $t_{b_{m1}}, \ldots, t_{b_{mN}}$ are the composition of latencies for processing the block with N sequential subtasks. By performing the N subtasks in parallel with the merge task in subtask $n = 1$, the new latency is reduced to:

$$L' = t_{b_{m1}} + t^c_{b_{m2}} + \ldots + t^c_{b_{mN}} \tag{3.6}$$

The superscript c refers to the latency to receive the results from subtask n, where $t^c_{b_{mn}} < t_{b_{mn}}$

is expected due to the parallel computations. The latency reduction is therefore,

$$L - L' = (t_{b_{m2}} + \ldots + t_{b_{mN}}) - (t^{c}_{b_{m2}} + \ldots + t^{c}_{b_{mN}}) \tag{3.7}$$

3.4.2 The *blocks_reorder* actor

This actor is also one of the actors in the CP. The following describes a strategy to increase its level of data parallelism. Figure 3.11 shows an implementation of the `blocks_reorder` actor, where originally, the output is sent per byte to the `add` actor to produce the predicted pixel. By analyzing the design of the `blocks_reorder` actor, all bytes for a single macroblock (256 bytes altogether) are available immediately after combining the variable blocks from the output of the `half_quarter_interpolation` actor. Therefore, all the pixels for one macroblock can be sent to the output in parallel to 256 parallel adders. However, this approach is not very efficient due to the very large additional resource that would be required for the adders and the buffer interconnections. The better approach here is to send n where $n < 256$ parallel pixels from the `blocks_reorder` and instantiate n parallel adders, where the value of n can be found by incrementing the value continuously until this actor no longer appears in the CP of the top-level network. In our case, $n = 16$ is found to be the optimum number of parallel output bytes that is required to remove this actor from the CP.

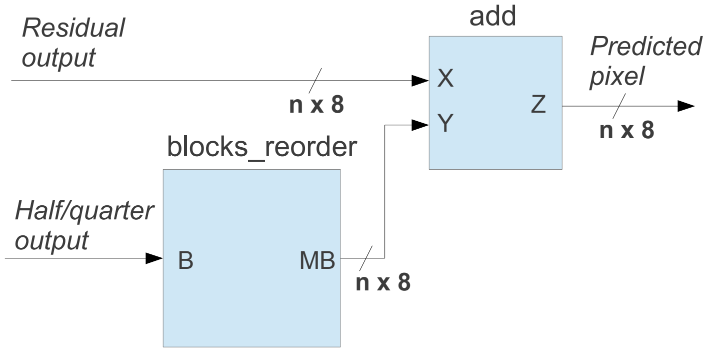


Figure 3.11: Actors `blocks_reorder` and `add` for producing inter-prediction pixels. In the original implementation, n is set to 1, where a single byte is sent serially for addition. Since the whole macroblock is available immediately from `blocks_reorder`, the value of of n can be set up to 256.

The following presents an estimated latency reduction for n parallel output bytes from the `blocks_reorder` actor, compared to the original implementation. The latency to obtain a single pixel from the output of the `add` actor is given by $L = t_1 + t_{add}$, where t_1 is the latency to send one byte from the `blocks_reorder` to the `add` actor, and t_{add} is the latency of the `add` operation. By sending n parallel bytes from the `blocks_reorder` actor, the latency to obtain a single pixel from the `add` actor is given by

$$L' = \frac{t_1}{n} + t_{add} \tag{3.8}$$

The latency for addition remains the same due to the implementation of n parallel adders. Note that the other adder operand from the residual output is not taken into account in the analysis since the residual component is not in the CP, i.e. the bottleneck comes from the output of the `blocks_reorder` actor. The estimated reduction in latency is given by:

$$L - L' = t_1 \times \frac{(n-1)}{n} \tag{3.9}$$

3.5 Reducing number of memory access on MPEG-4 AVC/H.264 decoder

This section presents some examples of applying the data-packing and redundancy elimination techniques presented in Section 3.2.2 on the design case study, based on the critical actors and optimization directions given by TURNUS.

3.5.1 The *picture_buffer* actor

Data-packing technique. The main functionality of the `picture_buffer` actor is to receive and store the current decoded pixels into memory, and send them as a reference frame in terms of variable blocks for predicting the next frame. The actor first stores the whole frame before proceeding to load the reference frame. Hence, large amount of memory bandwidth is required for both the store and load operations. Here, the estimated latency reduction is derived for this operation in the case of merging 4 bytes before a write access in a single macroblock. Let t_r be the latency to receive one token and t_e the latency to write a token to memory, both in terms of clock cycles per byte. In the implementation without data-packing, the total latency to write one macroblock to memory is $L = (t_r + t_e) \times MBSZ$ where $MBSZ = 16$ is the size of one macroblock in terms of bytes. In the implementation with data-packing, it would take $4 \times t_r$ clock cycles to receive 4 tokens and merge, and t_e clock cycles to write the 4 bytes into memory. However, since the action takes 4 bytes per firing, the number of firing is now $\frac{16}{4} = 4$. Therefore, the new latency is $L' = (4 \times t_r + t_e) \times 4$. The potential reduction in

```
writeData.Launch: action WD:[wd] ⇒
guard i < MBSZ
do
  pictureBuffer[picIdx][xAddr][yAddr] := wd;
  i := i + 1;
  ...
end
```

Figure 3.12: Original implementation of the action `writeData.Launch` that takes in a single pixel and stores into memory `pictureBuffer`.

```
writeData_new.Launch: action WD:[wd0,wd1,wd2,wd3] ⇒
guard i < (MBSZ >> 2)
do
  pictureBuffer[picIdx][xAddr][yAddr] :=
    (wd0<<24) or (wd1<<16) or (wd2<<8) or (wd3);
  i := i + 1;
  ...
end
```

Figure 3.13: Improved implementation of the action in Figure 3.12, that takes in 4 pixels in a single firing, merge the pixels into a 32-bit word, before storing into memory.

latency per macroblock is given by:

$$L - L' = \frac{3}{4} \times t_e \times 16 = 12 \times t_e \tag{3.10}$$

which is an estimated 12 times less memory access required per macroblock compared to the original implementation without data-packing.

The technique to merge a group of pixels for reducing the number of memory access is simple and effective. The slightly tricky part is when the stored pixels need to be loaded from memory, and sent as variable blocks to the inter prediction unit. The pixels are stored in a *raster-scan* fashion with each memory location containing 4 pixels, but they need to be read as blocks with individual pixels for a given dimension and location. The basic methodology is as follows. Given a block *B* with location *{RaOffx,RaOffy}* and dimension *{RaWidth,RaHeight}* that needs to be extracted from a frame *F*, the first step is to determine the correct memory location that contains the first pixel on the top-left of the block. With the row obtained directly with the value of *RaOffy*, the column is obtained by performing a modulo by 4 of *RaOffx* for a particular pixel location in memory. *RaOffx % 4=(0,1,2,3)* respectively means that it is the first, second, third, and fourth pixel. The adjacent pixels are taken accordingly until *RaWidth* has been reached. This is repeated for the other rows until *RaHeight* has also been reached.

Redundancy-elimination technique. By analyzing the CAL design of this actor given in Figure 3.14, there is a redundant intermediate storage called `ReadTable`, where on line 27, the extracted block from the main buffer called `pictureBuffer` is copied into another buffer called `ReadTable`. In the action `readData.launch` at line 36, the buffer `ReadTable` is accessed serially to send each pixel in the block. Here we can see that there is a redundant memory access when extracting the block, when in fact, the block can be sent directly *during* extraction. The improved implementation is given in Figure 3.15. The action `getReadAddrY` at line 27 gets the current y-position of the block, and the action `getReadAddrX` at line 34 directly extracts and sends the block for one line directly from the main buffer. Since the blocks are sent directly during extraction, the latency to store and load data to and from an intermediate memory is eliminated. This is a memory access reduction of $2 \times ((W+5) \times (H+5))$ per block for a block size of $W \times H$. Note the additional “5” term during extraction, which is due to the block that requires an offset by 5 pixels for interpolation.

3.5.2 The *half_quarter_interpolation* actor

Redundancy-elimination technique. Using a slightly different approach, the redundancy-elimination technique can also be applied on the most critical actor, the `half_quarter_interpolation`. The interpolation for half and/or quarter pixel is performed based on the values of the motion vectors, *Mvx* and *Mvy*. If both values are zero, then it is not necessary to perform half and/or quarter pixel interpolations, so the block can be sent directly for reordering. In the original implementation, the blocks are stored regardless if half and/or quarter pixel interpolation are required. The check for the motion vectors are performed *after* the blocks are stored. The implementation can be improved by performing the check *before* the blocks are stored, therefore eliminating the latency for storing blocks that are not required to be processed. This is shown in Figure 3.16. In the improved implementation (dashed arrow), the next action firing after `getMvSz` is selected to be `getBlock` or `sendBlock`, depending on the values of the motion vectors. If both are zero, then the block is not written into memory, but gets sent directly. However, this technique would only result in significant latency reduction if there are many blocks that do not require any interpolation or only integer interpolation.

3.6 Experimental results

Table 3.5 summarizes the results of applying the refactoring techniques on the MPEG-4 AVC/H.264 components case studies presented in the previous section. The re-factored CAL actors are synthesized to HDL for implementation on Xilinx Virtex-5 FPGA. Xilinx XST synthesis tool and Modelsim hardware simulator have been used to evaluate the designs in terms of resource and performance.

```
1  getReadAddr: action
2   RA:[RaOffX, RaOffY, RAWidth, RAHeight],
3   FRAME_TO_READ:[FrameNumToRead],
4   ENABLE_READ:[ReadEnabled] ⇒
5  guard
6   ReadEnabled
7  var
8   bool found := false,
9   int(size=SZ_NBPIC+1) refIdx := 0,
10  int yAddr := 0, int xAddr := 0
11 do
12  if FrameNumToRead=listFrameNum[0] then
13   refIdx := 0;
14   found  := true;
15  end
16  if (not found and FrameNumToRead=listFrameNum[1]) then
17   refIdx := 1;
18   found  := true;
19  end
20  RaOffX := clip_i32(RaOffX, −2*MB_WIDTH, picWidthInMacroB*MB_WIDTH);
21  RaOffY := clip_i32(RaOffY, −2*MB_WIDTH, picHeightInMacroB*MB_WIDTH);
22  readIdxMax := 0;
23  yAddr := RaOffY + 2*MB_WIDTH;
24  while (yAddr < RaOffY + RAHeight + 2*MB_WIDTH) do
25   xAddr := RaOffX + 2*MB_WIDTH;
26   while (xAddr < RaOffX + RAWidth + 2*MB_WIDTH) do
27    ReadTable[readIdxMax] := pictureBuffer[refIdx][xAddr][yAddr];
28    readIdxMax            := readIdxMax + 1;
29    xAddr := xAddr + 1;
30   end
31   yAddr := yAddr + 1;
32  end
33  readIdx    := 0;
34 end
35
36 readData.launch: action ⇒ RD:[rd]
37 guard
38  readIdx < readIdxMax
39 var
40  int(size=9) rd
41 do
42  rd      := ReadTable[readIdx];
43  readIdx := readIdx + 1;
44 end
45
46 readData.done: action ⇒
47 guard
48  readIdx = readIdxMax
49 end
```

Figure 3.14: The original implementation of extracting and sending blocks for half/quarter interpolation. The extracted block is stored in a redundant buffer `ReadTable` at line 27.

```
1  getReadAddr: action
2   RA :[RaOffX , RaOffY , RAWidth , RAHeight],
3   FRAME_TO_READ :[FrameNumToRead],
4   ENABLE_READ  :[ReadEnabled] ⇒
5  guard
6   ReadEnabled
7  var
8   bool found := false
9  do
10   if FrameNumToRead=listFrameNum[0] then
11    refIdx := 0;
12    found  := true;
13   end
14   if (not found and FrameNumToRead=listFrameNum[1]) then
15    refIdx := 1;
16    found  := true;
17   end
18   RaOffX := clip_i32(RaOffX, −2*MB_WIDTH, picWidthInMacroB*MB_WIDTH);
19   RaOffY := clip_i32(RaOffY, −2*MB_WIDTH, picHeightInMacroB*MB_WIDTH);
20   yAddr := RaOffY + 2*MB_WIDTH;
21   _RaOffY := RaOffY;
22   _RaOffX := RaOffX;
23   _RAHeight := RAHeight;
24   _RAWidth := RAWidth;
25  end
26
27  getReadAddrY: action ⇒
28  guard
29   (yAddr < _RaOffY + _RAHeight + 2*MB_WIDTH)
30  do
31   xAddr := _RaOffX + 2*MB_WIDTH;
32  end
33
34  getReadAddrX: action ⇒ RD:[rd]
35  guard
36   (xAddr < _RaOffX + _RAWidth + 2*MB_WIDTH)
37  var
38   int rd := 0
39  do
40   rd := pictureBuffer[refIdx][xAddr][yAddr];
41   xAddr := xAddr + 1;
42  end
43
44  doneGetReadAddrX: action ⇒
45  guard
46   not (xAddr < _RaOffX + _RAWidth + 2*MB_WIDTH)
47  do
48   yAddr := yAddr + 1;
49  end
50
51  doneGetReadAddrY: action ⇒
52  guard
53   not (yAddr < _RaOffY + _RAHeight + 2*MB_WIDTH)
54  end
```

Figure 3.15: The improved implementation of extracting and sending blocks for half/quarter interpolation. The extract and send processes are performed simultaneously by the action `getReadAddrX` at line 34.

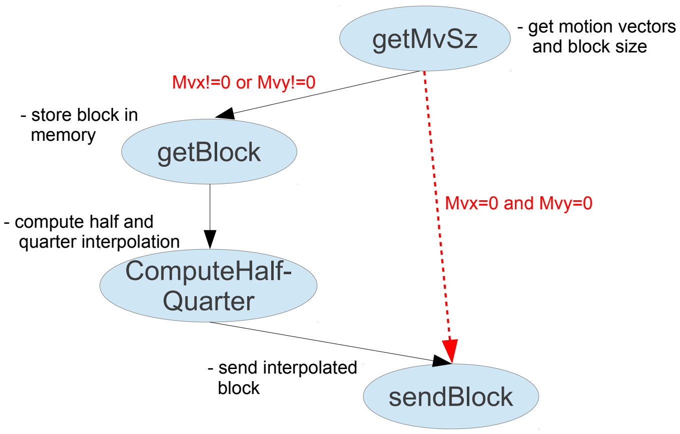


Figure 3.16: Part of the `half_quarter_interpolation` actor, where ovals are actions and arrows are transitions. The dashed arrow and the Mvx and Mvy checks represent an improved implementation where the block is not stored if half and/or quarter interpolation are not required.

For the `half_quarter_interpolation` actor, data parallelism is applied based on the method described in Section 3.4 by replicating the process 4 times. For this, a latency reduction by 3.6x is achieved, from 566,097 to 156,445 clock cycles per frame. Task parallelism is also applied by partitioning the task into 3 subtasks where a further latency reduction of 53853 clock cycles is obtained. In terms of resource, these techniques require significantly more slice and block RAM by up to 4.3x for both of these criteria. In terms of frequency, all designs result in the same value since the longest combinatorial path remains the same.

For the `blocks_reorder` actor, we instantiate 4 parallel 8-bit adders with 4 parallel input coming from the `blocks_reorder` actor instead of 1 in the original implementation, where a small gain of around 1000 clock cycles per frame is observed. However, when 16 parallel 8-bit adders are instantiated, a significant gain of about 90,000 clock cycles per frame is obtained, which corresponds to 3x improvement. The resource also increased significantly in terms of block RAM where an additional 78 is required, although the required occupied slice increased only by slightly. In terms of frequency, all designs result in the same value since the longest combinatorial path remains the same.

For the `picture_buffer` actor, memory optimizations are applied where for data-packing, the number of clock cycles is reduced by almost 40,000 per frame. For redundancy-elimination technique where it is applied after data-packing, a further latency reduction of about 100,000 clock cycles is observed. In terms of resource, data-packing requires slightly more due to extra logic to merge tokens and different ways to access memory, while the redundancy-elimination technique results in less slice, due to elimination of intermediate memory.

Table 3.5: Results of applying the refactoring and memory optimization techniques on several actors in the MPEG-4 AVC/H.264 decoder.

Actor	**Technique**	**Latency (c.c./frame)**	**f_{max} (MHz)**	**Occupied slice**	**BRAM**
half_quarter_-interpolation	*original*	566097	56	1794	13
	Data parallelism (4x)	156445	56	6882	48
	Task parallelism (3x)	102592	56	7733	57
blocks_reorder	*original*	134131	146	585	3
	Data parallelism (4x)	132930	146	653	5
	Data parallelism (16x)	44015	146	884	83
picture_buffer	*original*	499460	74	851	37
	Data-packing	460765	74	991	37
	Redundancy-elimination	360465	74	828	36

Here, the gain in latency is shown only for a given actor. The gain in the overall network when the actors are instantiated depends on the refactoring order and its contribution to the execution trace critical path. Different ordering of the techniques applied to the complex network would result in different design points in the exploration space. Theoretically, the best ordering is the one applied to the higher rank actor in the current critical path. In practice however, the possible computational load reduction of the critical actors are unknown. Therefore, finding the best refactoring techniques and directions at each subsequent stage can be reduced to a combinatorial problem, where the selection can be made using heuristics at each refactoring stage.

3.7 Summary

In this chapter, we have presented several refactoring techniques to minimize system latency which include data and task parallelism, and memory access reductions. First, background and related works on dataflow program refactoring, parallelism, and memory optimizations in video decoders were given. This is followed by a generic method to apply the refactoring techniques in dataflow programs. The MPEG-4 AVC/H.264 decoder case study was first analyzed and profiled using TURNUS where the critical path and computational load information were obtained. Based on these information, we then presented and applied specific refactoring techniques for some of the critical actors. The techniques have been proven analytically and experimentally to be very effective with significant reduction in system latency.

4 Maximizing system frequency with refactoring

In hardware designs, circuit pipelining is typically implemented to allow designs to operate at a higher maximum frequency by partitioning its data processing elements that are in the combinatorial critical path. By partitioning the relevant parts, the length of the critical path can be reduced, thus allowing a higher maximum operating frequency, and consequently, the design throughput as well.

This chapter presents such techniques that are applied at the level of dataflow programs, as opposed to traditional low-level RTL programs. First, background and related works on circuit pipelining are given, followed by our novel pipelining techniques for dataflow programs. This includes the mathematical modeling of the synthesis and optimization tasks by dataflow graph relations, its corresponding algorithms, and a methodology to pipeline complex designs. The final part of the chapter presents results on pipelining a single-actor design, and a more complex MPEG-4 decoder.

4.1 Background and related works

In computing, a pipeline is a set of data processing elements connected in series, so that the output of one element is the input of the next one. The elements of a pipeline are executed in parallel or in time-sliced fashion; in this case, some amount of buffer storage (pipeline registers) is inserted in between elements. The time between each clock signal is set to be greater than the longest delay between pipeline stages, so that when the registers are clocked, the data that is written to the following registers is the final result of the previous stage. A pipelined system typically requires more resource (circuit elements, processing units, computer memory, etc.) than one that executes one batch at a time, because each pipeline stage cannot reuse the resource of the other stages.

Key pipeline parameters are number of pipeline stages, latency, clock cycle time, delay, turnaround time, and throughput. A pipeline synthesis problem can be constrained either by resource or time, or a combination of both [61]. A resource-constraint pipeline synthesis limits the area of a chip or the available number of functional units of each type. In this case, the objective of the scheduler is to find a schedule with maximum performance, given available resource. On the other hand, a time-constraint pipeline synthesis specifies the required throughput and turnaround time, with the objective of the scheduler is to find a schedule that consume minimum resource.

Sehwa [95] is the first pipeline synthesis program. For a given constraint on the number of resource, it implements a pipelined datapath with minimum latency. Sehwa minimizes time delay using a modified list scheduling algorithm with a resource allocation Table. HAL [98] performs a time-constrained, functional pipelining scheduling using the force directed method which is modified in [62]. The loop winding method was proposed in the Elf [42] system. A loop iteration is partitioned horizontally into several pieces, which are then arranged in parallel to achieve a higher throughput. The percolation based scheduling [102] deals with loop winding by starting with an optimal schedule [15] which is obtained without considering resource constraints. Spaid [52] finds a maximally parallel pattern using a linear programming formulation. ATOMICS [43] performs loop optimization starting with estimating a latency and inter-iteration precedence. Operations which cannot be scheduled within the latency are folded to the next iteration, the latency is decreased and the folding is applied again. The above listed tools support resource sharing during pipeline optimization.

SODAS [66] is a pipelined datapath synthesis system targeted for application-specific DSP chip design. Taking signal flow graphs (SFG) as input, SODAS-DSP generates pipelined datapaths through iteratively constructive variation of the list scheduling and module allocation processes that iteratively improves the interconnection cost, where the measure of equidistribution of operations among pipeline partitions is adopted as the objective function. Area and performance trade-off in pipeline designs can be achieved by changing the synthesis parameters, data initiation interval, clock cycle time, and number of pipeline stages. Through careful scheduling of operations to pipeline stages and allocation of hardware modules, high utilization of hardware modules can be achieved.

Pipelining is an effective method to optimize the execution of a loop with or without loop carried dependencies, especially for digital signal processing [62]. Highly concurrent implementations can be obtained by overlapping the execution of consecutive iterations. Forward and backward scheduling is iteratively used to minimize the delay in order to have more silicon area for allocating additional resource which in turn will increase throughput.

Another important concept in circuit pipelining is Retiming, which exploits the ability to move registers in the circuit in order to decrease the length of the longest path while preserving its functional behavior [78]. A sequential circuit is an interconnection of logic gates and memory elements which communicate with its environment through primary inputs and primary outputs. The performance optimization problem of pipelined circuits is to maximize the clocking rate or equivalently minimize the cycle time of the circuit. The aim of constrained min-area retiming is to constrain the number of registers for a target clock period, under the assumption that all registers have the same area, the min-area retiming problem reduces to seeking a solution with the minimum number of registers in the circuit. In the retiming problem the objective function and constraints are linear, so linear programming techniques can be used to solve this problem. The basic version of retiming can be solved in polynomial time. The concept of retiming proposed by Leiserson et al [78] was extended to peripheral retiming in [85] by introducing the concept of a "negative" register. These works assume that the degree of functional pipelining has already been fixed and consider only the problem of adding pipeline buffers to improve performance of an asynchronous circuit.

The works discussed are mainly targeted at the generation and optimization of hardware resource from behavioral register transfer level (RTL) descriptions. As to our knowledge, there is no available tool that performs these functions at the level of a dataflow program. The development of the CAL dataflow language allows the application of these techniques at a higher abstraction level, thus provides the advantages of rapid design space exploration to explore pipeline throughput and area trade-off, and simpler transformation of a non-pipelined to a pipelined behavioral description.

4.2 Pipeline synthesis and optimization for CAL programs

The design abstraction of CAL dataflow programs can be loosely defined as pipelined implementations, where actors are the processing elements that are connected to adjacent actors by FIFO buffers. The key difference however, is that actors may contain many actions that are controlled by a local scheduler. In this case, data may not necessarily be written at FIFO buffers at every clock cycle as in true pipelined circuits. Furthermore, an action may require to execute for more than a single clock cycle, for example when a state variable access is performed. A dataflow program only becomes a fully pipelined implementation if every actor in the network contains just a single action that executes in a single clock cycle.

The idea behind our pipeline synthesis is to partition a single action that is contained in an actor, into several actions in separate actors that are interconnected by pipeline registers. The optimization task is to find the best way to partition the action with the objective of minimum pipeline resource.

The first step is to make the action body (i.e. sequence of operations) more analyzable. This is achieved by limiting each arithmetic operator to two operands, and assigning a unique output variable for each operator, essentially transforming each operator to a two-operands-single-assignment form. Figure 4.1 shows the ISO/IEC 23002-2 1D IDCT algorithm, with each operation represented as such. The algorithm uses 25 subtractors, 19 adders, and 52 variables. It also takes in eight inputs in parallel (*x0* to *x7*), and produces eight outputs in parallel (*o0* to *o7*). The transformed algorithm can be represented as a dataflow graph (DFG) as shown in Figure 4.2, which can then be analyzed using mathematical relations.

4.2.1 Dataflow graph relations

Let $N = \{1, \dots, n\}$ be a set of algorithm operators and $M = \{1, \dots, m\}$ be a set of algorithm variables. In the example of Figures 4.1 and 4.2, $N = 44$ and $M = 52$. The following matrices describe operator-variable and precedence relations.

1. The *operators/input variables relation.* The operators / input variables relation is described with the $F(n, m)$ matrix:

$$F = \begin{bmatrix} f_{1,1} & \cdots & f_{1,m} \\ \vdots & \ddots & \vdots \\ f_{n,1} & \cdots & f_{n,m} \end{bmatrix}$$

   where $f_{i,j} \in \{0,1\}$ for $i \in N$ and $j \in M$. If $f_{i,j} = 1$, then the *j* variable is an input for the i operator, otherwise it is not. For example in the sequence of operations in Figure 4.1, if $x1$ is variable $0 \in M$, and the + operator on line 1 is operator $0 \in N$, then $f_{0,0} = 1$ since $x1$ is the input for the given operator.

2. The *operators/output variables relation.* This relation describes which variables are outputs of the operators. It is represented with the *H(n,m)* matrix:

$$H = \begin{bmatrix} h_{1,1} & \cdots & h_{1,m} \\ \vdots & \ddots & \vdots \\ h_{n,1} & \cdots & h_{n,m} \end{bmatrix}$$

   where $h_{i,j} \in \{0,1\}$ for $i \in N$ and $j \in M$. If $h_{i,j} = 1$, then the *j* variable is an output for the i operator, otherwise it is not. For example in Figure 4.1, if xa is variable $8 \in M$ and the + operator on line 1 is operator $0 \in N$, then $f_{0,8} = 1$ since xa is the output for the given operator.

3. The *operator direct precedence relation.* This relation describes a partial order on the set of operators derived from analysis of the data dependencies between operators on the dataflow graph. The relation is represented with the $P_{direct}(n, n)$ matrix:

```
xa := x1 + x7;
xb := x1 - x7;
x11 := xa + x3;
x31 := xa - x3;
x71 := xb + x5;
x51 := xb - x5;
y2 := (x31 >> 3) - (x31 >> 7);
y3 := y2 - (x31 >> 11);
xa1 := y2 + (y3 >> 1);
x32 := x31 - y2;
y21 := (x51 >> 3) - (x51 >> 7);
y31 := y21 - (x51 >> 11);
xb1 := y21 + (y31 >> 1);
x52 := x51 - y21;
x32 := x33 - xb1;
x53 := x52 + xa1;
y22 := (x11 >> 9) - x11;
x12 := (y22 >> 22) - y2;
y23 := (x71 >> 9) - x71;
x72 := (y23 >> 2) - y23;
x13 := x12 + x71;
x73 := x11 - x72;
y24 := x2 + (x2 >> 5);
x21 := (y24 >> 2) + (x2 >> 4);
xa3 := y24 - (y24 >> 2);
y25 := x6 + (x6 >> 5);
x61 := (y25 >> 2) + (x6 >> 4);
xb3 := y25 - (y25 >> 2);
x22 := xb3 - x21;
x62 := x61 + xa3;
xa4 := x0 + x4;
xb4 := x0 - x4;
x01 := xa4 + x62;
x63 := xa4 - x62;
x41 := xb4 + x22;
x23 := xb4 - x22;
o0 := x01 + x13;
o1 := x41 + x53;
o2 := x23 + x33;
o3 := x63 + x73;
o4 := x63 - x73;
o5 := x23 - x33;
o6 := x41 - x53;
o7 := x01 - x13;
```

Figure 4.1: The ISO/IEC 23002-2 1D IDCT algorithm in the two-operands-single-assignment form. It consists of 25 subtractors, 19 adders, and 52 variables. Shifters assume no cost in hardware implementation.

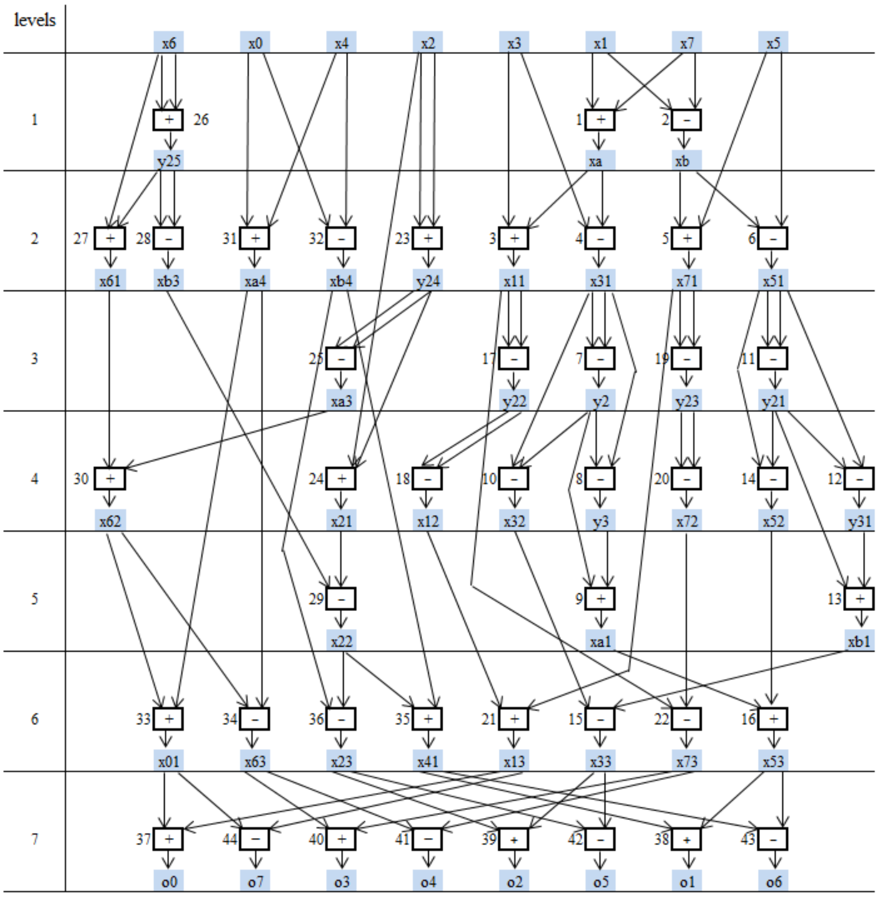


Figure 4.2: Dataflow graph of the ISO/IEC 23002-2 1D IDCT algorithm in the two-operands-single-assignment form. There are a maximum of 7 stages for minimum granularity.

Table 4.1: CAL operator relative delays. The "+/-" operator is selected as the reference with delay of 1.00.

No.	CAL operator type	Time delay
1	+/-	1.00
3	*	3.00
4	> / <	0.10
6	bitand/bitor	0.02
8	not	0.01
11	if	0.05
12	other	...

$$P_{direct} = \begin{bmatrix} p_{1,1} & \cdots & p_{1,n} \\ \vdots & \ddots & \vdots \\ p_{n,1} & \cdots & p_{n,n} \end{bmatrix}$$

where $p_{i,j} \in \{0,1\}$ for $i, j \in N$. If $p_{i,j} = 1$, then the i operator is a direct predecessor for the j operator, otherwise it is not. For example in Figure 4.1, the operations on lines 1 and 3 are direct predecessors, therefore, $p_{0,2} = 1$.

4. The *operator precedence relation.* The direct/indirect precedence P_{total} relation between operators can be inferred by applying the transitive closure operation to the $P_{direct}(n,n)$ matrix:

$$P_{total} = P_{direct} \cup P^2_{direct} \cup \ldots \cup P^i_{direct} \cup \ldots \cup P^n_{direct} \tag{4.1}$$

where P^i_{direct} is P_{direct} in power of *i*. P_{direct} defines the *direct* precedence relation and P_{total} defines the precedence relation. For example in Figure 4.1, the operations on lines 1 and 7 are indirect predecessors, i.e. the two operations are linked by another operation on line 4. Therefore, $p_{total_{0,6}} = 1$, but $p_{0,6} = 0$.

The operators/variables precedence relations guarantee that the dependencies between operators and variables are respected when creating pipeline schedules. In order to meet timing requirement, the timing delays for all operators need to be estimated. In this case, the *relative* delay of an adder is used, where an adder is assigned a time delay of 1.00. All other operators are assigned time delays relative to the delay of an adder, as shown in Table 4.1.

With this information, the P_{total} matrix can be extended to include the maximum delay between two operators on the dataflow graph. This is defined by the G matrix:

$$G = \begin{bmatrix} g_{1,1} & \cdots & g_{1,n} \\ \vdots & \ddots & \vdots \\ g_{n,1} & \cdots & g_{n,n} \end{bmatrix}$$

where $g_{i,j}$ at $i, j \in N$ is a real value . If $g_{i,j} = 0$ then there exist no path between *i* and *j* operators on the dataflow graph, and the corresponding element of the P_{total} matrix is also equal to zero. If $g_{i,j} > 0$ then there is a path between the operators. Essentially, the maximum value t_{max} in the G matrix defines the longest path in the dataflow network, i.e. the maximum time delay to execute the action. For a single stage, the minimum time delay t_{min} is the maximum delay for a single operator. Therefore, the range of stage-time-delay (T_{stage}) that can be set is between t_{max} and t_{min}.

Based on the G matrix and T_{stage}, the *mobility* of each operator can be determined, i.e. the possibility of a given operator to be scheduled to various pipeline stages. The earliest stage that an operator *i* may be scheduled as is called *asap(i)*, and the latest as *alap(i)*. Hence the mobility of operator *i* is given by *alap(i)-asap(i)*. If an operator may be scheduled to only one stage, then mobility equals to zero. Table 4.2 shows the mobility all 44 operators with $T_{stage} = 4$ for the IDCT example.

Table 4.2: Operator mobility for the IDCT with T_{stage}=4. Operator with mobility 0 means that it can only be scheduled to a single stage, and 1 means that it can be scheduled to 2 different stages.

Mobility	**Operators**
0	1,2,3,5,6,7,8,9,12,13,15,16,21,22,23,25,35,36,37,38,39,40,41,42,43,44
1	10,11,14,17,18,19,20,24,26,27,28,29,30,31,32,33,34

Given T_{stage}, the number of required pipeline stages can be also determined, i.e. the ceiling of t_{max}/T_{stage}. Let $K = \{1, \ldots, k\}$ be a set of pipeline stages. The distribution of operators onto pipeline stages is described with the X matrix:

$$X = \begin{bmatrix} x_{1,1} & \cdots & x_{1,n} \\ \vdots & \ddots & \vdots \\ x_{k,1} & \cdots & x_{k,n} \end{bmatrix}$$

In the matrix, the number of rows is equal to the number k of pipeline stages and the number of columns is equal to the number n of operators. A $x_{i,j} \in \{0, 1\}$ variable for $i \in N$ and $j \in K$ takes one of two possible values. If $x_{i,j} = 1$ then the i operator is scheduled to the j stage, otherwise it is not scheduled to the stage. The X matrix essentially describes a distribution of the operators on the stages.

Searching from all possible X matrices in the optimization space is inefficient, since not all X matrices would result in valid pipeline schedules. For example, two adjacent operators with direct precedence relation may not be assigned to the same pipeline stage due to the T_{stage} constraint. The operator conflict relation is as follows,

$$C = \begin{bmatrix} c_{1,1} & \cdots & c_{1,n} \\ \vdots & \ddots & \vdots \\ c_{n,1} & \cdots & c_{n,n} \end{bmatrix}$$

where $c_{i,j} \in \{0,1\}$ for $i, j \in N$. If $c_{i,j} = 1$, then the *i* operator conflicts with the *j* operator (i.e. cannot be assigned on same pipeline stage), otherwise it is not.

and the operator non-conflict relation,

$$nC = \begin{bmatrix} nc_{1,1} & \cdots & nc_{1,n} \\ \vdots & \ddots & \vdots \\ nc_{n,1} & \cdots & nc_{n,n} \end{bmatrix}$$

where $nc_{i,j} \in \{0,1\}$ for $i, j \in N$. If $nc_{i,j} = 1$, then the *i* operator does not conflict with the *j* operator, otherwise it is.

Note that the matrices C and nC are not necessarily their complement due to the T_{stage} requirement and the precedence relations between the operators.

4.2.2 Optimization tasks

Since one possible and valid X matrix describes a particular pipeline schedule, a global optimum schedule can be found by searching for a schedule that meets a required optimization objective among *all* possible and valid X matrices. In our case, we would like to find a pipeline schedule X with minimum pipeline register resource for a required T_{stage}.

Let Ω be a set of all possible and valid X matrix. The objective function as follows finds the matrix X with minimum total pipeline register width:

$$\min_{X\in\Omega} \sum_{s=1}^{k} \Big\{ \sum_{j=1}^{m} [\max_{i\in N}(f_{i,j} \times x_{s,i}) - \max_{i\in N}(h_{i,j} \times x_{s,i})] \times width(j) + \sum_{j=1}^{m} [\max(\tau_j, \max_{e=s+1,\ldots,k,i\in N}(f_{i,j} \times x_{e,i})) - \max_{e=s,\ldots,k,i\in N}(h_{i,j} \times x_{e,i})] \times width(j) \Big\} \tag{4.2}$$

where $\tau_j = 1$ if the j variable is an output token and $\tau_j = 0$ otherwise; × is the arithmetic multiplication operation.

There are two parts in equation 4.2. The first one estimates for each stage *s* the width of registers inserted in between the stage and the previous neighboring stage. The second one estimates for each stage the width of transmission registers.

There are three constraints related to our optimization tasks – operator scheduling, time, and precedence constraints.

The operator scheduling constraint describes the requirement that each operator should belong to only one pipeline stage:

$$\sum_{s=asap(i)}^{alap(i)} x_{s,i} = 1\, for\ i \in N, \tag{4.3}$$

where *s* is a pipeline stage from the range *asap(i)* to *alap(i)*, i.e the earliest and latest that operator *i* can be scheduled to respectively.

The time constraint describes the requirement that the time delay between two operators *i* and *j* must not be larger than $t_{required}$ if the operators are scheduled to one pipeline stage *s*:

$$x_{s,i} \times x_{s,j} \times g_{i,j} \le t_{required}\, for\ i, j \in N and\, s \in K, \tag{4.4}$$

where $g_{i,j}$ is the longest path between *i* and *j* operators on the algorithm dataflow graph. It is easy to see that if the operators are in the same stage and $x_{s,i} = x_{s,j} = 1$ then the inequality as follows must hold: $g_{i,j} \le t_{required}$. If the operators are not in the same stage the longest path length may be larger than the stage delay.

The operator precedence constraint describes the requirement that if the i operator is a predecessor of the j operator on a dataflow graph then i must be scheduled to a stage whose number is not greater than the number of stage which j operator is scheduled to:

$$\sum_{s=asap(i)}^{alap(i)} (s \times x_{s,i}) - \sum_{s=asap(j)}^{alap(j)} (s \times x_{s,j}) \leq 0$$
$$for\ (i, j) \in PrecedenceRelation, \tag{4.5}$$

where $PrecedenceRelation \subseteq N \times N$ is described by the P_{total} matrix. Constraints 4.3, 4.4, and 4.5 together define the structure of the optimization space.

Given an objective function with a set of constraints, an exact solution can be found and is typically implemented using the techniques of integer linear programming (ILP). However, ILP formulation is known to exhibit high computational complexity for relatively large functions such as that given in our case. The following section describes our algorithm and approach to finding the optimum pipeline schedule for a given T_{stage} constraint.

4.2.3 Synthesis and optimization algorithm

The general overview is given in Figure 4.3. Starting from a non-pipelined CAL actor and the pipeline stage-time requirement, ASAP and ALAP schedules are generated based on the $F, H, P_{direct}, P_{total}$, and G matrices. From this, operator mobility is determined and operators are arranged in order of mobility. This is then used in the *coloring* algorithm [89] that generates all possible and valid pipeline schedules X based on the C and nC relations. For each pipeline schedule, total register width is estimated, and the least among all schedules is taken as the optimal solution, which is finally used to generate pipelined CAL actors.

The algorithm to select possible and valid pipeline schedule is based on the coloring algorithm that colors the nodes such that no edge $(i, j) \in E, i, j \in V$ has two end-points with the same color. For any two adjacent nodes i and j, the inequality as follows holds: $color(i) \neq color(j)$. A chromatic number $\chi(G)$ of the undirected graph G is the minimum number of colors over all possible colorings.

However, since our conflict and non-conflict graphs are directed graphs, coloring on *directed* graphs is presented using the following additional requirement: for directed edge $(i, j) \in E$ the inequality as follows should hold: $color(i) < color(j)$. In the pipeline optimization task, if the directed operator conflict graph has a chromatic number $\chi(G)$ then the pipeline can be constructed on $\chi(G)$ stages. The problem of purely directed graph chromatic number can be

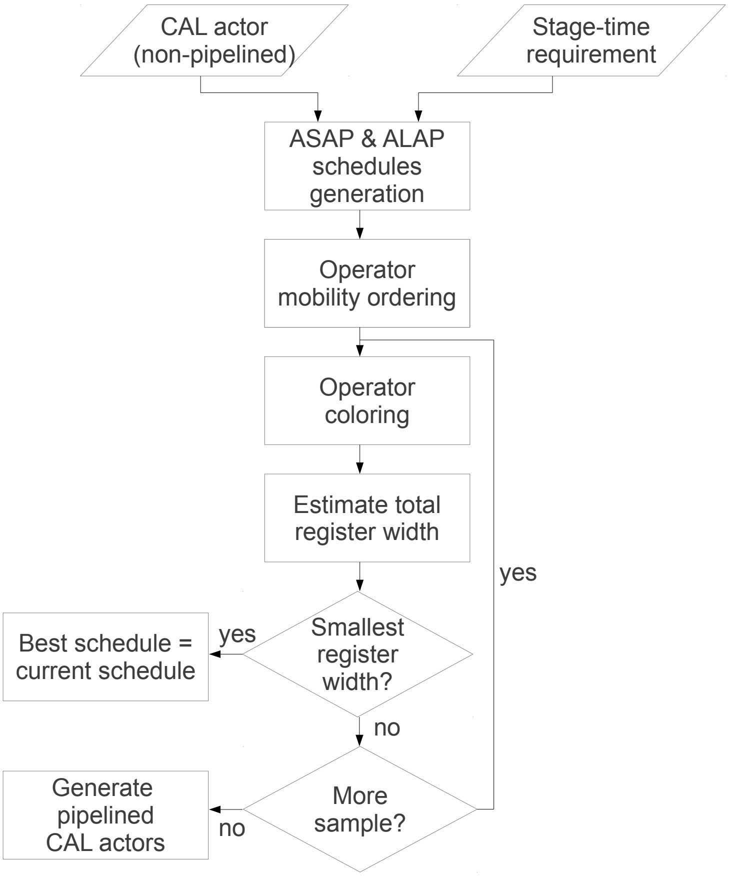


Figure 4.3: Methodology to synthesize and optimize non-pipelined CAL actors to pipelined CAL actors.

```
RegWidthColoringStep(top) begin
  if top >= n then
    completeColorings := completeColorings + 1;
    regWidth := totalRegisterWidth(ColorStack);
    if minRegWidth > regWidth then
      optimalColors := colorStack; minRegWidth := regWidth;
      if not MeetOptimizationTimeConstraint() then exit;
    end if
    return;
  end if
  for c in minColor(top) to maxColor(top) do
    colorStack(top) :=c;
    if RegWidthLowerBound(colorStack,asap,alap) >= minRegWidth then
      cutBranches := cutBranches + 1; continue;
    end if
    if top < n-1 then
      oper := order(top+1);
      minC := estimateMinConflictColor(ColorStack,top,oper,ConflictRelation);
      maxC := estimateMaxConflictColor(ColorStack,top,oper,ConflictRelation);
      minP := estimateMinNonConflictColor(ColorStack,top,oper,
                NonConflictRelation);
      maxP := estimateMaxNonConflictColor(ColorStack,top,oper,
                NonConflictRelation);
      minColor(top+1) := maximum(asap(order(top+1)),minC+1,minP);
      maxColor(top+1) := minimum(alap(order(top+1)),maxC-1,maxP);
      if minColor(top+1) > maxColor(top+1) then continue; end if
    end if
    coloringStep(top+1);
  end for
end
```

Figure 4.4: The algorithm for register width minimization on set of operator colorings

reduced to the problem of longest directed path length in the operator conflict graph. This problem has polynomial complexity.

The algorithm in Figure 4.4 works as follows. The recursive function takes in an input parameter *top*, which indicates the top record in the stack of operators. Depending on the *top* value, the function can return the control, generate the next complete coloring solution and compare it with the best current one, choose the next correct color of the current operator and generate the next record in the stack for procedure recursive call. In the next *top*+1 record, the minimum and maximum colors of the next operator is determined. If the minimum color is larger than the maximum color, then recoloring of the current operator is performed. The computation of minimum and maximum colors for operators are performed for both the conflict and non-conflict relations.

Figure 4.5 shows an algorithm to estimate minimum colors from a conflict relation. Among all

```
estimateMinConflictColor (ColorStack,top,op,ConflictRelation) begin
 minC := 0;
 for i in 0 to top do
  c := colorStack(i);
  nd := order(i);
  if (nd, op) is in ConflictRelation then
   if minC < c then minC := c; end if
  end if
 end for
 return minC;
end
```

Figure 4.5: The algorithm for estimating minimum color from conflict relation

operators that are recorded in the stack as predecessors and are in conflict relation with the given operator *op*, the operator with maximum color gives the value of *minC* that is returned by the algorithm as minimum color of *op* operator. The computations of maximum color from a conflict relation, minimum color from a non-conflict relation, and maximum color from a non-conflict relation are performed in a similar way.

Once all operators have been colored and a valid pipeline schedule is generated, the total register width is estimated to evaluate the efficiency of the schedule. From all possible pipeline schedules, the one with the smallest total register width is saved and selected as the best schedule.

The final step is to generate CAL actors from the optimal coloring. This is done by taking the best pipeline schedule, partition the operators according to the scheduled stage, and print the required operations, variables, registers, inputs and outputs declarations according to the syntax of the CAL dataflow language. The top level *XDF* network of pipelined CAL actors is also automatically generated based on the required number of pipeline stages.

It should be noted that our program is designed to generate potentially all possible and valid pipeline schedules for a given T_{stage} constraint, therefore results in a global optimum solution. The number of possible schedules depends on the mobility of operators; an algorithm with many operators that can be moved among various stages would generate many possible schedules, therefore could potentially take a long time to find a global optimum. In this case, the basic coloring algorithm can be extended to include a branch-and-bound algorithm by means of introducing a lower bound function on the register width using partial operator coloring that is recorded in the stack.

4.3 Pipeline methodology for complex dataflow network

The previous section presents a technique to efficiently partition a given algorithm in the action body. For complex dataflow network where each actor may contain more than a single action, loop operations, and access to state variables and/or memory elements, some preliminary steps are required before the pipeline synthesis and optimization tool can be used.

The proposed methodology to pipeline complex CAL dataflow network is given in Figure 4.6. Starting from a CAL program, it is first synthesized to HDL, and then to RTL, where the information on the combinatorial critical path can be obtained. The action that dominates this critical path is called the *critical action*. If the critical action is not in the trace critical path, then partitioning this action into the same actor will only increase the latency for this particular actor, but not the whole network. On the other hand, if the action is in the trace critical path, then the action should be first extracted into its own actor (if the actor contains other actions) before partitioning the action into other dedicated actors. By partitioning the action into separate actors, we can guarantee that the different pipeline stages are executed in parallel without reducing the latency. For example, the actor *sample* with a single action *a* in Figure 4.7 can be partitioned into two actors *sample1* and *sample2* with a buffer interconnection between ports *Out_a1* and *In_a1*.

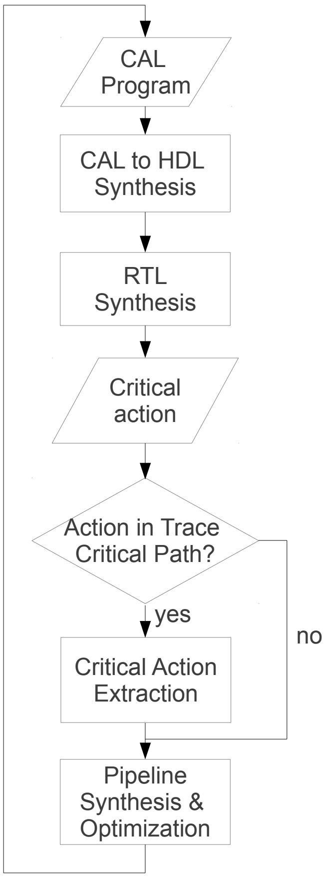


Figure 4.6: Action pipelining methodology for complex CAL dataflow network. If the action with the longest combinatorial path is in the trace critical path, then the action needs to be extracted first before applying the methodology in Figure 4.3.

.

```
actor sample()
 int(size=SZ) Input ⇒ int(size=SZ) Output :
 a: action Input:[in] ⇒ Output:[out]
 do
   ...
   ...
   ...
 end
end
```

Figure 4.7: Actor *sample* with a single action *a*.

```
actor sample1()
 int(size=SZ) Input ⇒ int(size=SZ) Out_a1, .. :
 a1: action Input:[in] ⇒ Out_a1:[out_a1], ..
 do
  ...
 end
end

actor sample2()
 int(size=SZ) In_a1, .. ⇒ int(size=SZ) Output :
 a2: action In_a1:[in_a1], .. ⇒ Output:[out]
 do
  ...
 end
end
```

Figure 4.8: 2-stage pipeline of actor *sample* with two actions *a1* and *a2*. FIFO interconnections between the two actors are equivalent to pipeline registers.

The technique to extract an action from an actor that contains other actions, into its own actor is as follows. First, the action has to be analyzed for its main input and output ports. Along with its original input port, the state variables read/used by the action has to be received also via input ports. If the action modifies a state variable, then the value has to be sent as feedback via an output port to the other actions that require this variable. In the case when the input port of the critical action is also used by other actions in the original actor, the consumption of the token from the port has to be properly controlled by guard conditions such that the two actions do not consume the tokens at the same time. As for the output port, in the case when other actions in the original actor are also using the same port as in the critical action, the port has to be multiplexed correctly such that only a single output token from the port is taken at a time. The final implementation of the critical action with its own dedicated actor contains new definition of input and output ports, and the original action body. It is interfaced with other actors that contain other actions from the original actor, and may consists of additional actors that provide the necessary input and output control.

The above methodology works well for simple actions that consist of only basic arithmetic and logical operations, and conditional statements. When an action also contains memory accesses and loop operations, the problem becomes slightly tricky. If a memory element is shared among several actions, then it will be inefficient to send the values in the memory element of the original actor to the partitioned action in another actor, especially if the memory size is large. A solution can be found if only a few memory location is read and write, where they can be received and sent respectively using input and output ports. However, if this is not the case, then it is necessary to resort to same actor partitioning where frequency and latency trade-off has to be analyzed.

As for loop operations, the challenge is how to perform pipelining for an iterative process. If the loop operation is static, i.e. the number of iterations can be determined statically, then loop unrolling can be performed to obtain only basic operations in the action body, which can then be pipelined. However, this has to be done carefully since unrolling a loop with large iteration count can be inefficient in terms of resource. If this is the case, then it is again necessary to resort to same actor partitioning where frequency and latency trade-off has to be analyzed.

In order to illustrate the action extraction methodology, the actor example in Figure 2.3 is used. The action `ac` is found to be the critical action that needs to be extracted due to its complexity. The action does not perform any memory access and does not contain loop operations. The state variables `quant`, `round` and `count` are sent to the extracted action as ports, along with the input port `AC`. Since it is the only action that is using the input port `AC`, no further control is required on this port. However, the output `OUT` is also used by the action `dc`, therefore a new actor would need to be created to select the correct output at a given time. The top level network after action extraction is shown in Figure 4.9. The critical actor `AC` can now be automatically pipelined using the methodology in Figure 4.3, with the resulting top level network shown in Figure 4.10.

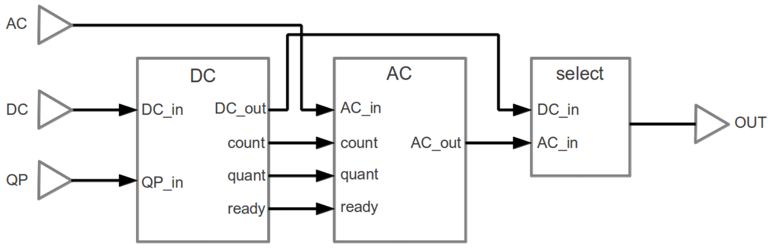


Figure 4.9: Multi-actor implementation of the `inverse_quantization` actor in Figure 2.3. The action `ac` is now contained in the actor `AC` with a latency of 1. The action can now be automatically pipelined using the methodology given in Figure 4.3.

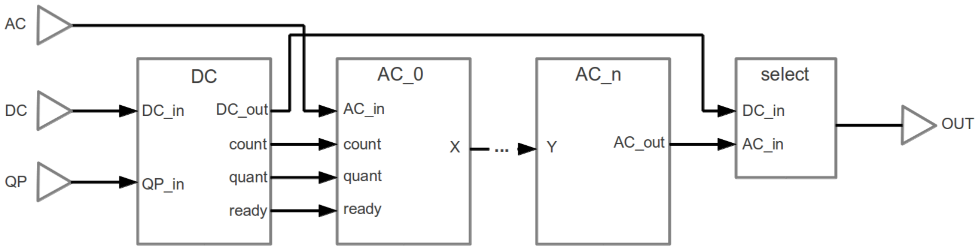


Figure 4.10: Top level network after pipelining the critical actor `AC` using the automated pipeline synthesis and optimization tool. The actor `AC` is now contained in actors `AC_0` to `AC_n`.

4.4 Experimental results

This section is divided into two parts: results for pipelining the ISO/IEC 23002-2 1D-IDCT algorithm (used in the MPEG-4 SP decoder) given in Figure 4.1 to demonstrate the automatic synthesis and optimization techniques for various pipeline configurations, and the more complex MPEG-4 SP decoder (described in chapter 6) by pipelining various actions in different actors to demonstrate the methodology for pipelining complex dataflow network. All designs have been written in CAL, pipelined using our automated tool, and synthesized to HDL for Xilinx Virtex-5 FPGA implementation. Xilinx XST synthesis tool and Modelsim hardware simulator have been used to evaluate the designs in terms of resource and performance.

4.4.1 ISO/IEC 23002-2 1D-IDCT

The first step is to determine the range of the number of possible pipeline stages for implementation. From the DFG in Figure 4.2, the relative time delay of each adder and subtractor are assigned to 1.00, the longest path length from source to sink is found to be 7.00. Therefore, $T_{stage} = 1.00$ synthesizes to a 7-stage pipeline, T_{stage}=2.00 to a 4-stage pipeline, T_{stage}=3.00 to a 3-stage pipeline, T_{stage}=4.00 to a 2-stage pipeline, and $T_{stage} \geq 7$ to a non-pipelined implementation.

For each of the n-stage pipeline for $n = \{2, 3, 4, 7\}$, ASAP, ALAP, best and worst schedules are generated. Table 4.3 summarizes the result. For a 2-stage pipeline of T_{stage}=4.00, the highest total register width is the worst-case with 494, followed by ASAP with 364, ALAP with 312, and the best case with only 260. This results in a register-width reduction of 90% compared to the worst-case. The optimization space for this pipeline configuration is 24,336. For a 3-stage pipeline, register width reduction between best and worst cases is almost similar, with 88.9%. However, the optimization space is significantly more with 29,555,604 possible pipeline schedules. The 4-stage design shows the highest number of optimization space with more than 63 million schedules, with register width reduction of 43.8%. The smallest reduction is in the 7-stage pipeline with only 21.9%. This configuration also results in the most total register width with up to 2028 in the worst case. It is interesting to note that for 7-stage pipeline, the schedules generated are the lowest among all n-stage pipeline configurations due to the lower mobility of operators.

The result for FPGA implementation is shown in Figure 4.11. Non-pipeline implementation results in 1650 total slice (sum of slice register and slice LUT) with throughput of 764 Mpixels/s (with maximum operating frequency of 115 MHz, mean latency of 6.7 pixels per clock cycle). As pipeline stages is increased from 2-stages to 4-stages, a roughly linear increase in throughput and resource is observed. However, for 7-stage pipeline, the throughput is saturated to that of 3-stages and 4-stages pipeline. The maximum throughput obtained is 1654 Mpixels/s (at

Table 4.3: The 8x8 1D IDCT: Exploration of pipeline optimization space for 2, 3, 4, and 7 stage pipeline with asap, alap, best and worst case pipeline schedules.

N_{stage}	T_{stage}	Total Register-Width				Register-Width Reduction (%)	Schedules Generated	Feasible Schedules
		asap	**alap**	**best**	**worst**			
2	4.00	364	312	260	494	90.0	24336	24336
3	3.00	520	624	468	884	88.9	29555604	29555604
4	2.00	832	832	832	1196	43.8	63002926	63002926
7	1.00	1664	1716	1664	2028	21.9	4505752	4505752

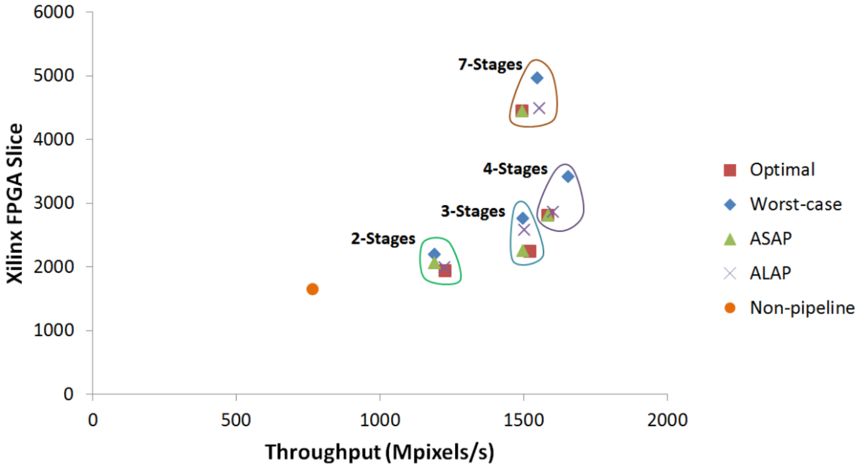


Figure 4.11: Slice versus throughput for all implementations of the 8x8 1D IDCT.

roughly 200 MHz) with total slice of 3419 for the 4-stages pipeline, which corresponds to 2.07x more slice and 2.16x higher throughput compared to non-pipeline implementation. However, for the best case (resource optimized) 4-stages pipeline, it utilizes only 70% more slice with a throughput increase of 2.08x. The reason for throughput saturation in the 7-stages pipeline is due to the path that is now dominated by interconnect and registers, rather than the operators.

4.4.2 MPEG-4 SP decoder

The strategy employed here is to iteratively find the critical action in this complex network, and synthesize the action into 2-stage pipeline implementation at every iteration. Eventually, the combinatorial path will be dominated by the routing delay, in which case, a very fine-grain action descriptions in the network is obtained.

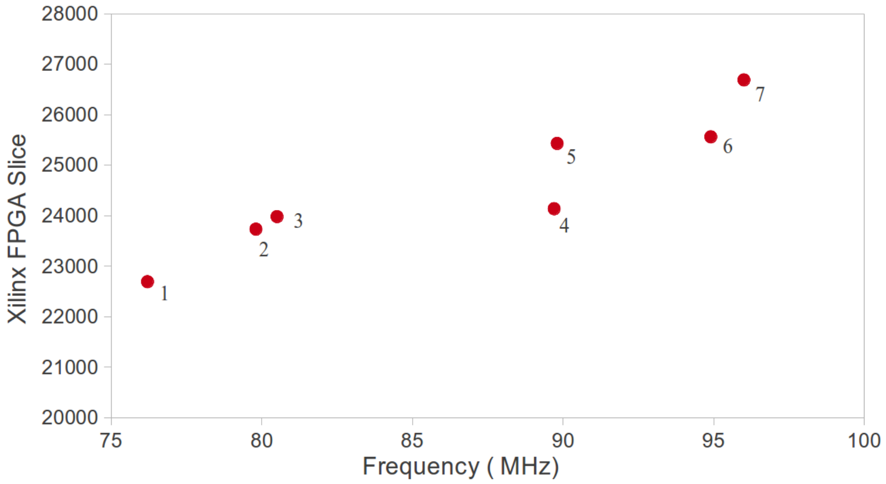


Figure 4.12: Frequency versus slice for various pipeline iterations of the MPEG-4 SP decoder. The details of each iteration is given in Table 4.4.

Figure 4.12 shows a graph of frequency versus resource after seven iterations of pipelining, and Table 4.4 with details for each iteration. From the iterations, it is found that five actions in four actors appear as critical: the actor `inverse_dc_pred` with actions `read_intra` and `getdc_inter`; the `picture_buffer` actor with action `read_address`; the `inverse_quant` actor with action `ac`; and the `idct` actor with action `calc_row`. In the final implementation, all actions are pipelined into 2-stages, except the `inverse_dc_pred` with 4-stages. This actor is also found to be outside of the trace critical path, therefore the critical actions are simply partitioned into the same actor. All other critical actions are extracted and pipelined into separate actors. The final frequency obtained is 96 MHz, which corresponds to 26% increase in frequency compared to the original non-pipelined implementation. The additional resource (in terms of slice LUT) required is around 18% more between these two points.

4.5 Summary

In this chapter, we have presented a novel pipeline synthesis and optimization technique that allows automatic partitioning of dataflow actions into pipeline stages for a given timing constraint using minimum pipeline resource. The technique is designed based on dataflow graph relations and matrices that describe the algorithm in the action body. Using these models, an algorithm has been developed based on the coloring algorithm to efficiently search the optimization space for a pipeline schedule that meets the required objective of minimum resource. Furthermore, a methodology for pipelining complex CAL dataflow network has also been proposed, including a technique to extract a critical action into a separate actor. In the final section, we provided experimental results based on two design cases, the ISO/IEC

Table 4.4: Logical delay, routing delay, total delay, and maximum frequency after seven iterations of pipelining the MPEG-4 SP decoder. Also shown are the corresponding actors and actions at each iteration, along with pipeline type, $Type = 0$ for same actor partitioning, and $Type = 1$ for separate actor partitioning.

Iteration	**Actor**	**Action**	**Type**	**Logical (ns)**	**Routing (ns)**	**Total (ns)**	**F_max (MHz)**
1	inverse_dc_pred	read_intra	0	9.4	3.8	13.2	75.8
2	picture_buffer	read_address	1	9.4	3.3	12.7	78.7
3	inverse_dc_pred	read_intra	0	9.1	3.5	12.6	79.4
4	inverse_dc_pred	read_intra	0	5.3	7.2	12.5	80.0
5	inverse_quant	ac	1	6.5	4.7	11.2	89.3
6	inverse_dc_pred	getdc_inter	0	5.5	5.1	10.6	94.3
7	idct	calc_row	1	2.9	7.5	10.4	96.3

23002-2 1D-IDCT and the MPEG-4 SP decoder, where in both cases, significant throughput improvements have been achieved using minimal additional resource. It should be noted as well that the techniques are generic and applicable to any dataflow networks and actors, including another of our design case study – the MPEG-4 AVC/H.264 decoder. Results are given in chapter 7.

5 Minimizing resource with buffer size optimization

The objective of the refactoring techniques presented in previous chapters is to improve system performance at the cost of higher computing resource. In contrast, this chapter presents techniques to minimize resource by reducing the size of the buffer interconnections between dataflow actors. The selection of the sizes of each FIFO buffer in a dataflow network is crucial as it impacts both the system functionality and performance, not to mention the implementation resource as well.

In this chapter, two novel approaches to optimize the size of the FIFO buffers in CAL programs are presented: First based on a hardware program execution, and second based on TURNUS critical path and execution trace analysis. Both provide methodologies to find the close-to-minimum buffer size for deadlock-free execution, and further extension to explore larger buffer sizes to obtain higher throughput.

This chapter is organized as follows. The next chapter presents background and related works on minimizing buffer sizes for both, a restricted SDF and a generic DPN MoC. This is then followed by our techniques for assigning and minimizing buffer sizes for CAL dataflow programs, using the two different approaches as mentioned. Each technique is also supplemented with experimental results on two design components of the MPEG-4 AVC/H.264 decoder. The final section presents chapter summary.

5.1 Background and related works

At the implementation level, actors can be executed in parallel. Therefore, high throughput system is obtained if as many actors as possible are executed (i.e. fired) at a given time. As mentioned in Section 2.1 regarding action firing rule, an action in an actor fires if enabled by: 1) availability of input tokens, 2) value of input tokens, controlled by guard conditions, 3) the actor scheduler, 4) the action priority, and/or 5) the availability of free space to store output tokens. In order to ensure that actions are enabled and fired as quick as possible (hence results in higher throughput), conditions (1) and/or (5) have to be met as fast as possible. A dataflow network with large buffer sizes between actors would satisfy these conditions (for both actors) at a higher rate since input tokens are rapidly available from the buffers, and output tokens can always be generated due to large output buffers. However, setting all buffers to large values may not result in area-efficient implementation. On the other hand, buffer sizes that are too small between actors may introduce system deadlock. This is a condition when one or more actor stalls while waiting for input tokens that will never arrive, or actions that could not fire due to an insufficient space on the output buffer.

Since CAL programs is based on the DPN MoC, the problem of finding bounded buffer sizes for deadlock-free execution is undecidable [96], which means that it is not possible to find a bounded buffer size for all possible execution order. This is due to the non-determinism of DPN actor execution where the order of action firing depends on the input token. In this case, a given set of bounded buffer sizes for a dataflow network results in a complete execution of only a *subset* of all execution orders; The same buffer size configuration that works on a given execution order may not result in deadlock-free execution for other execution orders.

Due to this limitation of finding a strictly bounded (i.e. bounded for all execution order) buffer memory requirement in DPN, a subset of DPN called the synchronous dataflow graphs (SDF) have been used that allows the construction of a bounded memory for *all* complete execution order at compile-time, if it exists. This is due to the static nature of SDF actors that consume and produce fixed number of tokens. However, it comes at the cost of lower design flexibility where action selection and order of firing could not depend on the value of the input tokens, which is a main feature in video codec design. Since CAL programs allow the implementation of SDF actors, the following reviews some techniques and approaches to finding the minimum buffer size requirements for designs using only SDF actors.

5.1.1 Single appearance scheduling in SDF

Most of the work on minimizing buffer sizes in a SDF network is designed for an embedded software target with an additional requirement of minimum code size. These two constitute the *total* memory requirement for a design implemented on a software platform. The work in [25] presents such optimization techniques, where a scheduling policy called the *Single Appearance Scheduling* (SAS) is introduced that results in minimum code size. The actor execution order based on this scheduling policy is then used to find the minimum buffer size requirement between the actors. Figure 5.1 illustrates an example of SAS scheduling. Here, the actor X produces 2 tokens and actor Y consumes 3 tokens on edge e_{XY} for every firing. Similarly, the actor Y also produces 1 token and actor Z consumes 2 tokens for every firing. In order to obtain the SAS schedule, the balance equations as follows can be solved:

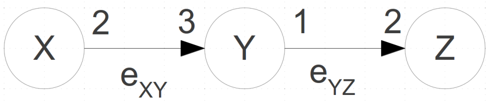


Figure 5.1: SDF graph example with actors X, Y, and Z, together with annotations for token consumption and production.

$$v_X \times p_X = v_Y \times c_Y \tag{5.1}$$

$$v_Y \times p_Y = v_Z \times c_Z \tag{5.2}$$

where v_X, p_X, and c_X respectively are the number of firing, the number of tokens produced, and the number of tokens consumed for actor X (similarly for actors Y and Z). With known number of token consumption and production for every actor, equations 5.1 and 5.2 become,

$$v_X \times 2 = v_Y \times 3 \tag{5.3}$$

$$v_Y \times 1 = v_Z \times 2 \tag{5.4}$$

This can be easily be solved with $v_X = 3$, $v_Y = 2$, and $v_Z = 1$. The SAS schedule with minimum code size is therefore *3X2YZ*. If the actors are fired in the sequence *XXXYYZ*, then the simple network can be executed forever without deadlock. The minimum number of firing of each actor in a SAS schedule is represented by a *repetitions vector* q_G for a graph G. For this example, $q_G = [3,2,1]$. The required buffer size for this scheduling policy can be inferred by the token consumption or production, i.e. $e_{XY} = 6$ since the actor X is fired 3 times (similarly since actor Y is fired 2 times), and $e_{YZ} = 2$ since the actor Y is fired 2 times (similarly since Z is fired once). The total buffer size for this schedule is 8.

The problem with the approach above is that it prioritizes code size over buffer size, which may not lead to a global minimum buffer size. For example, using a non-SAS schedule *2XYXYZ* in Figure 5.1 results in buffer size of only 6, but larger code size. The task of minimizing buffer sizes for any arbitrary schedule is known to be NP-complete, and is the main feature of the works in [40], [47], and [82], where a model-checker has been used to find an exact solution to the problem.

5.1.2 Finding minimum buffer sizes using model-checker for SDF

Model-checker is essentially a tool to automatically verify the correctness properties of finite-state systems. Among others, it has been used to solve NP-complete scheduling problems. The use of model-checker to minimize buffer size requirement in SDF graphs was first reported in [40], using the SPIN [55] model. The operational semantics of SDF is first encoded in the model-checker using the state-space approach. From the derivation of the state-space of the SDF graph $(\Gamma, \rightarrow)$ where Γ is the set of buffer configurations for all edges for a given SDF, it is converted to an equivalent model-checking state-space $(\Gamma \times \Gamma, \Rightarrow)$; the first channel quantity is the current configuration, and the second encodes the storage bounds required for the schedule so far. For a finite schedule, the bounds in the last configuration are the required channel bounds; for infinite schedules, the bounds converge to the required bounds. The model-checking state-space is encoded in the SPIN modeling language *Promela*, with parameters given in table 5.1. The verification challenge is formulated as follows:

"Every schedule will eventually require a storage capacity larger than BOUND."

If this claim is false, SPIN will provide a counter example, which is a schedule within the required bounds.

Table 5.1: Key notations used in the SPIN model.

Parameter	**Details**
sz[i]	Buffer size of edge *i*
ch[i]	Current number of tokens on edge *i*
nf[j]	Maximum allowed number of firings of actor *j* before its repetition count is reached
np[i]	Maximum allowed number of firings of the producer actor of edge i without buffer overflow on edge *i*
nc[i]	Maximum allowed number of firings of the consumer actor of edge *i* without buffer underflow on edge *i*
sl	Current schedule length (sum of all actor firing counts so far)
prev_sl	Schedule length in the previous iteration
ESL	Expected schedule length (sum of repetition counts of all actors)
ls	Loop count of an actor
ACTOR	Number of actors
EDGE	Number of edges

The technique of utilizing the model-checker directly runs the risk of state-space explosion for large graphs. The work in [82] implements the SPIN model-checker with a modified bounded greedy algorithm (BGA) called the *BGA with buffer increase* in order to overcome this issue. Using the same verification challenge for SPIN, if it is proven true, then `BOUND` is a safe lower bound on the buffer size requirement, but it may not be a tight bound, so the value of `BOUND` is incremented for the next iteration. Eventually the verification will be proven false, where SPIN has found a schedule with total buffer size <=`BOUND` as a counter example. The minimum buffer size requirement is the value `BOUND` such that it is true for `BOUND-1`, but false for `BOUND`. Binary search is used to narrow down the range of minimum buffer size requirement.

The buffer size increment technique is based on actor firing order using the BGA algorithm, where each actor fires for the maximum number of times consecutively while respecting the initial `BOUND`. If running BGA on the SDF graph leads to deadlock, then the buffer sizes on the edges for which the producer actor could not fire due to insufficient buffer space (called the constraining edges) is incremented. The process is repeated until deadlock is resolved permanently, and a feasible buffer size distribution is obtained. For optimal buffer sizes, the size on the constraining edges is incremented exhaustively in turn, each forming one branch of the search tree. Following different branches along the search tree leads to different total buffer size, and the least among the set is taken as the optimal solution.

The approach in [47] uses the NuSMV [30] model-checker to obtain minimum buffer size requirement for SDF graphs. Similar to [82], it also aims to address the state-space explosion problem in the original work in [40], but using a symbolic model-checker where the size of the data structure does not increase with the increasing state-space size. Furthermore, in order to optimize the code size due to a non-SAS scheduler, the authors also proposed a technique called *dynamic SAS*. The schedule obtained with minimum buffer configuration is dynamically modified to obtain an optimized code representation. For example, given a non-SAS schedule *ABCDBCDC*, the technique to obtain a SAS a schedule is by introducing runtime decisions, such as the one given in figure 5.2.

```
for (i=0; i<8; i++){
 if (i==0) A();
 else if (i==1 || i==4) B();
 else if (i==3 || i==6) D();
 else C();
```

Figure 5.2: SAS scheduling using runtime decisions from a non-SAS schedule.

This naive way of generating a SAS schedule may incur large runtime performance overheads. The overheads can be reduced by using bit-shifting and bitwise comparisons instead of integer and boolean comparisons. The final result is the possibility to obtain both an optimized code size and buffer size.

5.1.3 Buffer size minimization for DPN

As mentioned, the task of finding minimum buffer size for a deadlock-free DPN is undecidable, i.e. could not be determined statically. Therefore, it is necessary to resort to dynamic analysis where the task of finding buffer size configurations would need to depend on the input stream. Similar to the approach for SDF graphs, the methodology to obtain feasible buffer size configurations is done by determining the required scheduling. Dynamic scheduling policies can generally be classified as data driven, demand driven, or some combination of the two [124].

Data Driven Scheduling

In this scheduling policy, a process is activated as soon as sufficient data is available. This results in a complete execution of the program because an execution stops if and only if all of the processes are blocked from reading an empty channel. However, this requires that the buffer sizes are unbounded to guarantee a complete execution.

Data driven scheduling can be described as follows. Given a generic DPN described by a connected graph $G(V,E)$ where V is the set of vertices corresponding to processes, and E the set of edges corresponding to the communication channels. At a given execution time, find the set $V_e \subseteq V$ of all enabled processes in G, where a process can either be blocked due to empty input channel, or enabled. By assuming unbounded buffer sizes for all communication channels, the graph G would execute forever as long as there are sufficient input data, whereby $V_e \neq 0$ at any given time.

Figure 5.3 shows an example of a process network using data driven scheduling that would require an unbounded buffer size configuration in order to obtain a complete execution. The source processes $g(2)$ and $g(5)$ generates data forever from zero, and incrementally in multiples of 2 and 5 respectively, i.e. the outputs of $g(2) = (0,2,4,\ldots)$ and $g(5) = (0,5,10,\ldots)$. The process m merges the data on the inputs such that a monotonically increasing integer sequence is obtained, with the algorithm given in figure 5.4, i.e. $m = (0,2,4,5,6,8,10,\ldots)$. The process p simply consumes the result from process m and performs the print function.

Using data driven scheduling where it is assumed that the two generators $g(2)$ and $g(5)$ outputs data at *every* time step, it is easy to see that tokens would eventually accumulate without bound at edge y, since process m consumes data from edge x and y at different rates. In this case, process m takes one data from edge y for every 2 data from edge x, as illustrated in table 5.2. t is the time step of execution, and $t \to \infty$ if the network is to execute forever. After step 3 of execution, two data accumulate on edge y, i.e. $D_y = (10,15)$; after 6 steps, $D_y = (15,20,25,30)$, and after 9 steps, $D_y = (20,25,30,35,40,45)$. As time steps are increased, more data is accumulated on edge y, which results in an unbounded execution.

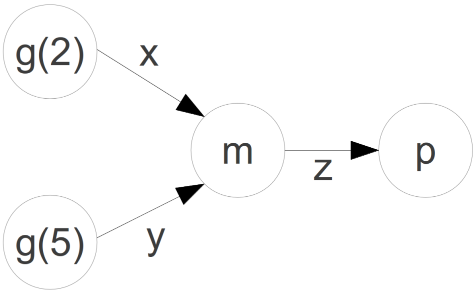


Figure 5.3: Dataflow process network for generating a monotonically increasing sequence. The result is an unbounded execution if data driven scheduling is used with different output rates for $g(2)$ and $g(5)$.

```
int u = get (x); //get data from edge x
int v = get (y); //get data from edge y
do
{
 if (u < v) {
  put (u); //output from edge x
  u = get (x); //get data from edge x
 }
 if (u > v) {
  put (v); //output from edge y
  v = get (y); //get data from edge y
 }
 if (u == v) {
  put (u); //output from either edges
  u = get (x) //get data from edge x
  v = get (y) //get data from edge y
 }
} forever;
```

Figure 5.4: Example of a dataflow process that merges the data on its inputs such that a monotonically increasing integer sequence is obtained.

Table 5.2: Time steps from 0 to 9 and the corresponding data written to edges x, y, and z. Result is based on the process network in figure 5.3.

Time step, t	0	1	2	3	4	5	6	7	8	9
Edge, x	0	2	4	6	8	10	12	14	16	18
Edge, y	0	5	10	15	20	25	30	35	40	45
Edge, z	0	2	4	5	6	8	10	12	14	15

Demand Driven Scheduling

In contrast to data driven scheduling where processes are enabled as long as data is available, demand driven scheduling enables a process based on the demand of the consumer. The demand typically starts with the process that produces the ultimate output of the program. When a process attempts to consume data from and empty channel, it is suspended, and the channel is marked as *hungry*, and the producer process for that channel is activated. When this new process is activated, it may attempt to consume from an empty input channel, which would cause another process to be activated in turn. When a process produces data on a hungry channel, it is suspended, and the waiting consumer process is activated. In essence, it is the reverse of data driven scheduling, where processes are activated from sink to source. With this scheduling policy, the process network in Figure 5.3 could execute with bounded buffer sizes, since the generators $g(2)$ and $g(5)$ are not enabled at every time step, but only when they are in demand.

However, there are instances when demand driven scheduling could also result in an unbounded execution, as illustrated in Figure 5.5. The process $n(5)$ directs its input to edge y if it is a multiple of 5, else to edge z. The process $g(1)$ generates an output incrementally by 1, i.e. $g = (0,1,2,3,\ldots)$, while the processes p_0 and p_1 simply performs the print function. If p_0 and p_1 generate demands at the same rate, then tokens will accumulate on the edge z since every time p_0 consumes a token on y, 4 tokens will accumulate on edge z. In general, after k token are consumed by process p_0, $(i-1)\times k$ tokens will be consumed by process p_1, where i is the parameter of process n. The token accumulation on edge z is due to the scheduling policy that is based on the simultaneous demands of the sink processes p_0 and p_1.

Combined data and demand driven scheduling

A pure data or demand driven scheduling could result in unbounded buffer sizes for deadlock-free execution, respectively for multiple source and sink processes. One of the earliest work in combining the scheduling policies is given in [104], called Eazyflow. The execution of the processes alternates between data and demand driven, where data driven execution begins if there is a token deficit, and continues until there is a token surplus, at which point demand driven execution resumes. The approach is based on classifying data streams as either *eager* or

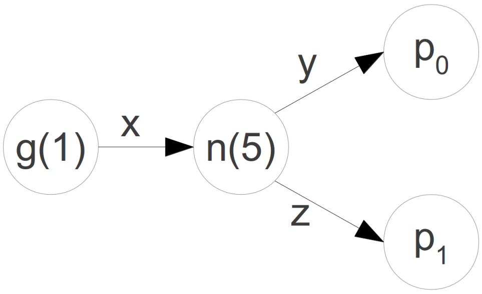


Figure 5.5: Dataflow process network example for which a demand driven scheduling results in an unbounded buffer size configurations for a deadlock-free execution.

lazy; eager streams are associated with processes that consume and/or produce fixed amount of data (i.e. SDF), while lazy streams are associated with processes that consume and produce unpredictable amounts of data (i.e. DPN). Processes that produce eager streams would be executed until it reaches a certain surplus to lazy streams. In this way, data driven scheduling is partially used to supply data, and an unbounded accumulation of tokens would not occur since the scheduling would then switch to a demand driven one. Another interesting approach is presented in [99] and [100] where a DPN graph is transformed so that a data driven execution of the new graph is equivalent to a demand driven execution of the original graph. For each edge in a graph, a new edge is added to carry demands in the opposite direction. This edge essentially acts as a control for a hybrid data and demand driven scheduling.

5.2 Buffer size assignment and reduction for CAL programs

There are relatively few works in literature concerning the assignment and reduction of buffer sizes for CAL programs, although the importance of the task has been pointed out (Section 5.1). One related work is given in [37], but focuses only on finding an optimum scheduler for DPN actors executed on a general purpose CPU. This work does not provide anything on the buffer size configurations. In this section, our approaches to finding an optimum buffer size configuration for CAL programs are presented. First, based on the hardware program execution, and second, based on the dataflow program analysis.

The simplest way to assign buffer sizes to the communication channels in a CAL network is by transforming the network G to produce a semantically equivalent network G^0 that is bounded by b^0 (for all interconnections). This transformation may introduce deadlock such that the network G^0 represents only a partial execution of the original network G. However, if the execution of the network G^0 never stops and results in a complete execution, then we have succeeded in implementing a complete and bounded execution. If the execution of G^0 stops and it represents only a partial execution of the original network G, then the bound b^0 have been chosen to be too small. One or more of the channels require a buffer size more than b^0 in order to obtain a complete and deadlock-free execution. Therefore, a larger bound $b^1 > b^0$ must be chosen. The bound can be chosen successively, $b^0 < b^1 < b^2 < \dots$ until eventually, a complete execution is obtained. By definition, there exists a bound b that is finite if the network G is to execute in finite time. After N iterations, a bound $b^N \leq b^0$ is obtained that corresponds to a complete execution of the network G^N.

The problem with this approach is that all communication channels have the same value of capacity, which in most cases, are not required for a complete and deadlock-free execution. The following presents a technique to set different capacity limits for different channels, based on the work in [96]. Essentially, having different capacity on the channels allow a much smaller total buffer size compared to a single bound b, thus meeting our objective of finding a minimum or close-to-minimum total buffer size.

5.2.1 Hardware program execution approach

As explained in Section 5.1, scheduling or the execution order of actors determine the required buffer size for complete and deadlock-free execution of a network. In the approach using hardware program execution, buffer sizes are assigned and optimized based on the execution order given by the synthesized hardware architecture. The generic top-level overview of the interconnection architecture between actors is given in Figure 5.6. Each actor port consists of two forward channels *Data* and *Send*, and two feedback channels *Ack* and *Ready*. The actual data is carried through the Data channel, while the Send channel signifies the validity of the data. Once an actor has consumed this data token, then it asserts the Ack signal. At the same time, the Ready signal is also asserted to signify that it is ready to accept more data tokens.

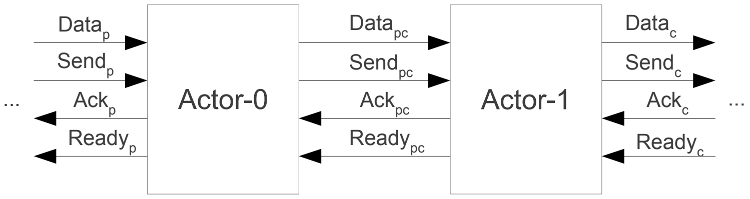


Figure 5.6: Interconnect architecture of actors in hardware. The firing of an actor (producer) is determined by the availability of data, and demand from the consumer.

Based on this architecture, it can be concluded that hardware scheduling is in fact based on a hybrid data and demand driven scheduling, where producer actor(s) are executed not only if there are sufficient input tokens, but also if the corresponding consumer actor(s) are ready to accept data tokens. Typically, a consumer actor is not ready to accept data tokens from a producer actor if the consumer actor could not produce data tokens (on the consumer output) due to an insufficient space on its output channel (i.e. *ready*=0). Once this output space becomes available, then *ready* is asserted to 1 so that the producer can fire and send the token to the consumer.

In a complex network, the execution order of actors largely depends on the set buffer sizes for the interconnections. Having large buffer sizes imply the Ready signals to be asserted more frequently, and therefore, more actor execution at a given time. On the other hand, smaller buffer sizes typically results in less actor execution at a given time, and in cases where the sizes are too small, none of the Ready signal may be asserted, which results in a deadlock condition where no actor can be executed. The following presents two approaches to find 1) close-to-minimum with deadlock free execution and 2) larger buffer sizes for higher throughput requirement.

Finding close-to-minimum buffer size configuration

We call this the ***HEM*** (short for **H**ardware program **E**xecution **M**inimum buffer size) technique, and works as follows. The initial capacities of all channels are set to 1 on the RTL description of the top network and a hardware simulation is performed. If execution stops due to deadlock, it means that one or more actors are blocked writing to a full channel. Increasing the capacity of channels that are not full does not allow execution to continue. It is necessary to increase the capacity of one or more full channels. It is important not to increase the largest buffer because this could lead to unbounded growth of that buffer. Therefore, only the capacity of the *smallest* full channel is increased, because this guarantees that every full channel will eventually be increased if necessary to unlock the program. The reasoning is that if the same channel is increased repeatedly, then eventually it will no longer be the smallest full channel. If some full channel other than the smallest full channel is increased, then some buffer could grow without bound. The way to prevent this from happening is to increase the smallest full channel, and to avoid additional tokens being added to destination actions with already sufficient input tokens.

This technique has been implemented using a tcl script for a fully automated approach, with the algorithm shown in Figure 5.7. The first step is to run a hardware simulation (using Modelsim), and verify the Send signal of the output for deadlock. The verification is done from $t_{sim} - k$ to t_{sim} (t_{sim} is total simulation time and k is a constant), where if the Send signal is not asserted at all during this time, then deadlock occurs. All buffer interconnections and their sizes are then automatically stored in a Modelsim *List* file and analyzed by an external Java program that doubles the smallest full buffers in the top RTL network. Hardware simulation is run again using this new buffer size configuration, and this is repeated until a deadlock-free execution is obtained. Note that the smallest full channel is doubled in capacity every time deadlock occurs, instead of for example, incremented by 1. This is due to the FIFO hardware architecture that is designed to be implemented as memory element with

```
while {$run_again == 1} {
    #run hw simulation
    do run_script_<design_name>.tcl
    set n 0
    set k 100
    #get "send" until value = ’1’ or $x clock cycles reached
    while {$n == 0 && $k < $x} {
        #get "send" value at time $t_sim-$k
        set n [exa -time [expr $t_sim - $k] <output_name>_send]
        set k [expr $k + 100]
    }
    #if "send" value not found for $y clock cycles
    if {$k >= $y} {
        echo "deadlock with k = $k"
        #get all buffers from waveform, save to list file
        do <tcl_<design_name>_add_list.tcl
        #find smallest full buffer, multiply by 2
        java -jar $java_program $design_name
        set iterations [expr $iterations + 1]
    } else {
        echo "done, no deadlock"
        set run_again 0
    }
}
```

Figure 5.7: Tcl script to automatically find the close-to-minimum buffer size configuration using the hardware program execution approach.

addressable locations. Therefore, it is required that the buffer sizes are defined by values that are in the power of 2.

This technique has been applied on two design components of the MPEG-4 AVC/H.264 decoder, with experimental results given in Section 5.3.

Reducing buffer sizes with a throughput constraint

Buffer size configuration that is the minimum or close-to-minimum typically does not yield a design with high throughput. System throughput can be improved by using larger buffer sizes, as explained in Section 5.1. A buffer size bound b for all interconnections (with a given throughput) can be limited either directly ([99, 100]) or indirectly [38] using feedback channels such that a lower total buffer size can be obtained, but with the same throughput as the one with a buffer size configuration with bound b.

Indirect buffer size reduction with feedback channels works as follows. Given a connected graph $G = (V, E)$ with a set of vertices V corresponding to the processes and a set of directed edges E corresponding to the communication channels. For each edge $e_i = (v_m, v_n)$, add a new edge $e'_i = (v_n, v_m)$ in the reverse direction. The channels corresponding to these new edges are called the *feedback channels*. Let $|e_i|$ be the size of an edge e_i, if $b_i - |e_i|$ tokens is placed on edge e'_i, then the total number of tokens for the pair of edges is $b_i = |e_i| + |e'_i|$. Typically there are no tokens initially on the communication channel, with $|e_i| = 0$. The process is modified such that it must consume one token from a feedback channel e'_i for each token that it produces on edge $|e_i|$. Thus the number of tokens on the pair of edges remain constant, i.e. $b_i = |e_i| + |e'_i|$. At the end of a complete execution, the number of data tokens on edge $|e_i|$ can be obtained, where $|e_i| \leq b_i$. Using the new buffer size configuration with this technique does not modify the original order of execution for when the buffer bound b_i is used for all communication channels.

Instead of adding feedback channels, the buffer capacity of the channels can also be limited directly by requiring the processes to be blocked when trying to write to a full channel. We call this technique, ***HEO*** (short for **H**ardware program **E**xecution buffer size **O**ptimization) and works as follows. First a large enough buffer bound b is searched for, such that a reasonably high throughput with practical bounds is obtained, which can be found quickly for example using a binary search. An RTL simulation of the program is performed for duration t based on this buffer size configuration. At each time step of the simulation, the number of tokens present in each buffer interconnection is recorded, where they are expected to fluctuate due to constant token production and consumption. The minimum required size for a given buffer interconnection channel is therefore the *peak capacity* (maximum number of tokens at any given time) of the buffer for the duration t. The new buffer size configuration using this peak capacity is expected to be less than the original bound b, but guarantees the same execution order and therefore, throughput as well. The algorithm can be summarized as follows:

- For each interconnection channel from $i = 0 \ldots m$, initialize to constant C to obtain a given throughput P.
- Run hardware simulation for duration T, and update the *peak-capacity* C_i^{peak} for each FIFO channel at every time step.
- The final values of C_i^{peak} after simulation time T are the minimized buffer size requirements for throughput P.

This technique has also been implemented using a tcl script for an automatic analysis and optimization, and has also been applied on two design components of the MPEG-4 AVC/H.264 decoder, with experimental results given in Section 5.3.

5.2.2 Dataflow program analysis (TURNUS) approach

Rather than performing the analysis on the hardware execution level, finding feasible buffer size configuration can also be made based on the execution trace obtained from dataflow program analysis TURNUS. Using the *makespan* (Section 2.4) metric, the most optimal CAL design for a given architecture is the one with shortest makespan, typically using unbounded buffers. A design that uses a bounded and smaller total buffer size for a deadlock-free execution results in larger makespan (lower throughput), but also achieves our objective of obtaining a close-to-minimum buffer size configuration. In between the two scenarios, intermediate configurations can also be obtained by gradually increasing the relevant buffer sizes in order reduce the makespan. By repeatedly performing this task, the trade-off between total buffer size requirement and throughput can be explored effectively.

Finding close-to-minimum buffer size

Recall that the longest weighted path in an execution trace is called the *Critical Path* (CP). Given a buffer size configuration x_β with the corresponding total buffer size $B(x_\beta)$, the objective is to find the configuration x_β with the minimum $B(x_\beta)$, while also minimizing the CP. Minimizing the total buffer size also entails minimizing the size of every individual buffer interconnection $b_{f_i}, f_i \in F$.

The general idea to obtain the minimum buffer size configuration is based on constructing a schedule using a topological sorting. Instead of using a platform-specific dataflow program execution as in the previous approach, the schedule is obtained using an execution trace where nodes represent action firing, and edges represent dependencies between the firing. The dependencies can either be internal dependencies E_I, or fifo dependencies E_F. For each $e_{i,j} \in E_F$, an additional discrete state variable $st_{e_{i,j}}$ is defined that takes either of the following values: $st_F = \{\texttt{unavailable}, \texttt{requested}, \texttt{available}\}$. For each fifo dependency the initial value is set to $st_{e_{i,j}}$ = `unavailable`. The topological sorting is based on a *graph-walk* method, where at each execution step k, a single node $v_i^k \in V$ is analyzed (starting from the sink node): If the action is fireable, then it is added to the last position of the new topological sorted vector $\mathbf{V} = \{v_i \prec v_{i+1} | v_i \in V, v_{i+1} \in V\}$, its outgoing fifo dependencies are made available one by one and consumer nodes are analyzed during the next execution step. Otherwise, if the action is not fireable yet, the graph is walked back to analyze the parent step. In some cases, dependencies on incoming fifo are not satisfied, therefore they are polled one by one where the corresponding producer nodes are analyzed during the following execution step. The analysis is complete after all input tokens are exhausted, in which case the final action firing schedule is obtained based on this topological sorting. The buffer size configuration for this final action execution ordering is found to be close-to-minimum since the execution is ordered such that the sequence of action firing generates the least amount of token on a given edge for a deadlock-free execution. Details on the implementation of this technique for the TURNUS framework is given in [109]. This technique has also been applied on our design case study, which we call the ***TEM*** (short for **T**race **E**xecution buffer size **M**inimization) technique. The results are given in Section 5.3.

Buffer size and throughput exploration

Given the minimum and maximum buffer size configurations x_β^0 and x_β^{max}, the objective is to find other buffer size configurations with total size $B(x_\beta)$ where $B(x_\beta^0) \leq B(x_\beta) \leq B(x_\beta^{max})$. The buffer size configuration with the minimum size can be found using heuristics as presented above; the maximum size is taken as the design with an unbounded or very large fifo sizes that would result in a high throughput design. Correspondingly in terms of critical path,

$$CP(x_\beta^{max}, \lambda) \leq CP(x_\beta, \lambda) \leq CP(x_\beta^0, \lambda) \tag{5.5}$$

where λ is a given scheduler for the buffer size configuration. Since the CP is a weighted execution trace, the weights are platform-specific, where for hardware implementation, it is simply the latency (i.e. number of clock cycles) for an action firing. For a fired action v_i in an execution trace, the weight w_{v_i} corresponds to the latency of the action for a buffer size configuration x_β^{max}. By reducing the relevant buffer sizes (which is shown next) to obtain new buffer size configuration x_β, the weight is updated as follows,

$$w_{v_i}(x_\beta, \lambda) = w_{v_i} + w_{v_i}^{x_\beta} + w_{v_i}^{\lambda} \tag{5.6}$$

Where $w_{v_i}^{\lambda}$ represents the execution time overhead introduced by the scheduler λ and $w_{v_i}^{x_\beta}$ is the time overhead introduced by the finite buffer size configuration x_β. Since there are no scheduling latency in hardware implementation, the term $w_{v_i}^{\lambda}$ will be ignored. The term $w_{v_i}^{x_\beta}$ is in fact the additional latency incurred by a blocked action firing due to insufficient output buffer space. It represents the time elapsed from the moment v_i becomes fire-able until its output buffer has enough space to store the produced tokens. Here we can see that in equation 5.5, the CP is longer for buffer configuration x_β compared to x_β^{max} due to an additional weight term $w_{v_i}^{x_\beta}$.

We will now present how the buffer sizes can be incremented for resource-throughput exploration. For each exploration step, the size of the *critical buffers* C_B is increased by the number of blocked tokens $\hat{\tau}(b_i)$ for a buffer interconnection i, i.e. the next buffer configuration is given by

$$x_\beta^{k+1} = x_\beta^k + \hat{\tau}(b_*), b_* \in C_B \tag{5.7}$$

The set of critical buffers C_B is retrieved from the blocked buffers of the critical actions C_A in the CP. By increasing the buffer size of the interconnections on the C_A, the makespan can be reduced since a higher output rate is obtained due to a larger output buffer space. This process of finding the critical buffers, incrementing, and reevaluate CP can be performed repeatedly until a saturation is reached where further increment of the critical buffers lead to no further reduction in makespan. The implementation of this technique on the TURNUS framework is given in [108]. This technique has also been applied on our design case study, which we call the ***TEO*** (short for **T**race **E**xecution buffer size **O**inimization) technique. The results are given in the next section.

5.3 Experimental results

In this section, the efficacy of the four buffer size minimization and optimization techniques presented in the previous section (HEM, HEO, TEM, and TEO) are evaluated and compared. For case study, two main components of the MPEG-4 AVC/H.264 decoder are utilized (Figure 7.8), *Decoder_Y* and *Decoder_U/V*. They consists of respectively, 188 and 58 buffer interconnections that need to be assigned. The performance is evaluated based on two criteria: total buffer size and throughput. The total buffer size is obtained by summing the sizes of all the FIFO buffers in the network in terms of bits. The throughput is obtained by first simulating the design in hardware to obtain the latency, i.e. number of clock cycles per QCIF video frame. Then, the design is synthesized and implemented on a Xilinx Virtex-5 FPGA to obtain the maximum operating frequency. In this case, $f_{max} = 114 MHz$ is obtained for the *Decoder_Y*, and $f_{max} = 79 MHz$ for *Decoder_U/V*. These frequency values are constant for all buffer size configurations since the design architecture remains the same for all cases. The throughput is calculated based on these latency and frequency values.

Figure 5.8 shows the results of applying the HEM technique on both designs (Section 5.2.1). For *Decoder_Y* and *Decoder_U/V* respectively, we obtain such execution after 122 and 49 iterations with total buffer size of 314,892 and 1,126,691 bits. All other configurations are incomplete and results in a deadlock. The throughput for these deadlock-free designs are shown in Table 5.3, together with results for non-optimized buffer configuration with $b = 8197$ and the HEO technique. The throughput for using the HEM technique is reduced by 48% and 16% respectively for *Decoder_Y* and *Decoder_U/V*. In terms of resource, the HEM technique uses the least amount, as expected, with up to 19x less total buffer size. The HEO technique results in exactly the same throughput as the non-optimized $b = 8197$ buffer size configuration, but uses 2.4x and 4.5x less total buffer size respectively for *Decoder_Y* and *Decoder_U/V*.

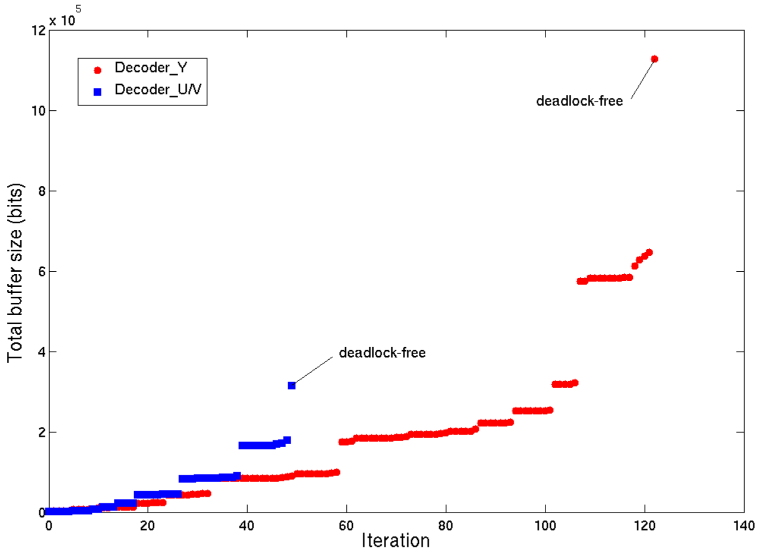


Figure 5.8: Results using the *HEM* technique on the *Decoder_Y* and *Decoder_U/V* of the MPEG-4 AVC/H.264 decoder case studies. The decoders are simulated for several iterations until a complete and deadlock-free execution are obtained for the given buffer size.

The following presents results using the TEM and TEO techniques, shown in Figures 5.9 and 5.10 respectively for *Decoder_Y* and *Decoder_U/V*. For both graphs, we can see that TURNUS provides a quite accurate estimate on the expected throughput, compared to actual hardware simulation for the buffer size configuration. The maximum throughput obtained is 797 and 1092 QCIF fps with total buffer size of 633704 and 194922 bits respectively for *Decoder_Y* and *Decoder_U/V*. After these points, further increment of the buffer size results in no gain

Table 5.3: Comparison of total buffer size and throughput for fixed buffer configuration b=8197, *HEM*, and *HEO* techniques on the *Decoder_Y* and *Decoder_U/V* of the MPEG-4 AVC/H.264 decoder case studies.

Configuration method	**Decoder_Y**		**Decoder_U/V**	
	Buffer size (bits)	**Throughput (QCIF fps)**	**Buffer size (bits)**	**Throughput (QCIF fps)**
b=8197	19595264	916	4898816	1092
HEM	1126691	621	314892	941
HEO	8030232	916	1105094	1092

in throughput due to the local minimum. If the HEO and the TEO (final configuration) techniques are compared, the HEO technique for *Decoder_Y* results in throughput of 916 QCIF fps compared to 797 QCIF fps (15% higher), but also results in higher resource with almost 12x more. For *Decoder_U/V*, the throughput are the same between TEO and HEO at 1092 QCIF fps, but the HEO technique results in around 6x more resource. For minimum buffer size comparison, the technique using TURNUS have shown to be superior in terms of resource, with up to 5x less (*Decoder_U/V*), but also lower throughput with up to 20% less (*Decoder_Y*).

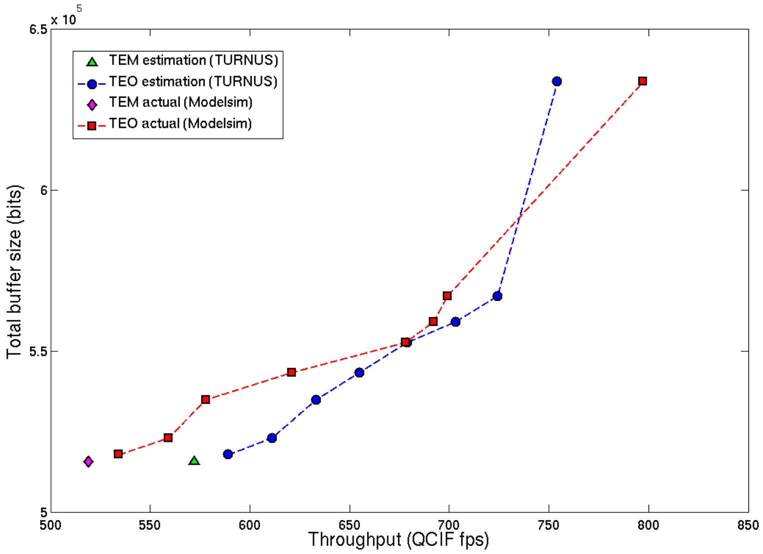


Figure 5.9: Throughput versus buffer size graph for estimated (TURNUS) and actual (Modelsim) results using the *TEM* and *TEO* techniques on the *Decoder_Y* of the MPEG-4 AVC/H.264 decoder case study.

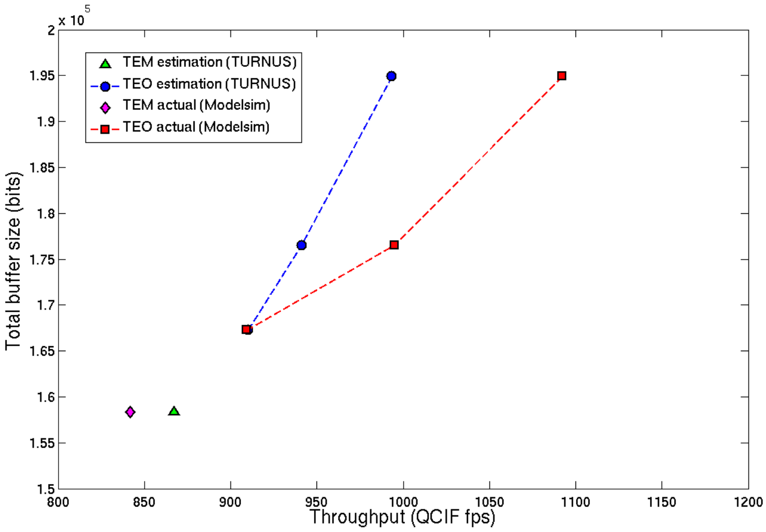


Figure 5.10: Throughput versus buffer size graph for estimated (TURNUS) and actual (Modelsim) results using the *TEM* and *TEO* techniques on the *Decoder_U/V* of the MPEG-4 AVC/H.264 decoder case study.

It is also interesting to compare the complexity of each technique and their convergence time. The un-optimized constant buffer size assignment is the simplest one, where virtually no time is required to obtain the configuration. The next is the HEO technique, where a valid buffer size configuration is obtained after a single simulation run to obtain the capacity of each buffer interconnection (in the order of minutes for the case studies). The HEM technique performs several simulation runs until deadlock is resolved. Therefore, the required time is design dependent, where on average a valid buffer size configuration is obtained after about 2 hours for the design case studies using a general purpose computer. For the TEM technique, the minimum buffer size configuration is obtained on average after 3 hours. The TEO technique to obtain other configurations are relatively fast afterwards, with several seconds for each one.

5.4 Summary

In this chapter, we have presented several buffer size minimization and optimization techniques for CAL programs based on two different approaches: 1) using a hardware program execution and 2) using a dataflow program execution trace. For each approach, two techniques were implemented: first is finding the minimum or close-to-minimum size for a deadlock free execution, and second is to use a larger buffer size for higher throughput. All the four techniques have been applied on the U/V-branch and the Y-branch decoder part of the MPEG-4 AVC/H.264 decoder, and then evaluated and compared for performance. It can be concluded that using approach-1 results in the highest throughput design with large total buffer size, while approach-2 results in the smallest buffer size but with low throughput. The techniques have been proven to be effective in minimizing and optimizing the buffer interconnection sizes, with up to 37x smaller size achieved compared to a direct un-optimized buffer size assignment.

6 Design case studies: MPEG-4 video decoders

This chapter introduces two complex video decoders that are used as design case studies in this work. The video decoders are used to validate the efficacy of the refactoring and optimization techniques presented in the previous three chapters. Furthermore, we also aim to validate and prove the viability of our systems design methodology and the associated tools and techniques using some of the most complex signal processing system available today - which is without doubt the MPEG-4 video decoders.

This chapter is organized as follows. First, the fundamentals of video codecs are presented, i.e. on both sides of compression and decompression, based on the reference [105]. Then, the Reconfigurable Video Coding (RVC) standard is presented, which adopts CAL as its specification language, and proposes a new way of specifying video codecs. The following section presents the two MPEG-4 decoders that are used in this work: the MPEG-4 Part 2 Visual Simple Profile (SP) and the MPEG-4 Advanced Video Coding (AVC)/H.264 Constrained Baseline Profile (CBP) decoders. Both were implemented for the RVC standard, using a subset of the CAL language called RVC-CAL. In these sections, the fundamentals of the standards are first presented, followed by their design and implementation with CAL. Finally in the last part, a concluding remark is provided based on all of these video coding standards in relation to the main work in this thesis.

6.1 Fundamentals of video codecs

Video codec is a tool that enables compression or decompression of digital video. The purpose of compression is to reduce the size (in case of storage) or bandwidth (in case of transmission) of a *raw* video data. The compressed video data then needs to be decompressed for recovering and viewing the video, which is performed by a video *decoder*. Compression is achieved by removing the *redundancy* , i.e. components that are not necessary for the reproduction of data. This typically involves *statistical* redundancy and can be effectively compressed using *lossless* compression. However, lossless compression of image and video information gives only a moderate amount of compression, for example the JPEG-LS with a compression ratio of around 3-4 times. *Lossy* compression is necessary to achieve higher compression (at around 40-80 times depending on the standard), at the expense of a loss in visual quality. In this case, the decompressed data is not identical to the source data, but for natural scenes and images, typically do not affect the viewer's perception of visual quality. All video coding standards today feature lossy compression, where some loss in visual quality is trade-off with the significantly higher compression ratio.

Most video coding methods exploit both temporal and spatial redundancy to achieve compression. In the temporal domain, there is usually a high correlation (similarity) between frames of video that were captured at around the same time. For example, successive frames are often highly correlated, more so if the frame rate is high. In this case, redundancy can be removed by constructing the current frame based on other frames. In the spatial domain, there is usually a high correlation between pixels that are close to each other, i.e. the values of neighboring samples are often very similar. In this case, redundancy can be removed for example by constructing a given block in the current frame by using neighboring blocks.

The study group who develop video coding standards for the International Standards Organization (ISO) is called the Moving Picture Experts Group (MPEG). It has been responsible for a series of important standards, starting with the MPEG-1 (compression of video and audio CD playback), and following on with the very successful MPEG-2 (storage and broadcasting of television-quality video and audio). The MPEG-4 (coding of audio-visual objects) is the next standard to be developed, that deals specifically with audio-visual coding. Another group who develop video coding standards, but for the International Telecommunication Union Telecommunication Standardization Sector (ITU-T) is called the Video Coding Experts Group (VCEG). It has been responsible mainly for a series of standards related to video communication over telecommunication networks and computer networks. The first standard was the H.261 for videoconferencing, followed by the H.262 that was developed jointly with the MPEG-2 standard. The H.263 standard was also developed which provides further enhancement and are more efficient. Another joint video project between MPEG and VCEG is the MPEG-4 Advanced Video Coding (AVC)/H.264. It is the current video coding standard, and possibly the most widely used today. This standard, together with the MPEG-4 Part 2 Visual and the MPEG-2/H.262 standards have had a particularly strong impact and have found their way into a wide variety of products.

All of the video codecs that is compatible with the H.261, MPEG-2/H.262, H.263, MPEG-1, MPEG-2, MPEG-4 Visual and MPEG-4 AVC/H.264 have been based on the same generic design that incorporates a motion estimation and compensation front end (sometimes described as DPCM), a transform stage and an entropy encoder. The model is often described as the hybrid DPCM/DCT codec. The most recent HEVC decoder also follows such model. Figure 6.1 and Figure 6.2 depict a generic DPCM/DCT video encoder and decoder respectively. In the encoder, video frame n ($\mathbf{F}_n$) is processed to produce a coded (compressed) bitstream, and in the decoder, the compressed bitstream is decoded to produce a reconstructed video frame $\mathbf{F}'_n$. The encoding process is as follows:

1. An input video frame $\mathbf{F}_n$ is presented for encoding and is processed in units of a macroblock (e.g. 16x16 luma and 8x8 chroma regions).
2. $\mathbf{F}_n$ is compared with the *reference* frame, e.g. the previous encoded frame ($\mathbf{F}_{n-1}$). A motion estimation function find a 16x16 region in this reference frame (or a sub-sample interpolated version) that matches the current macroblock in $\mathbf{F}_n$ according to some criteria. The offset between the current macroblock position and the chosen reference region is a motion vector MV.
3. Based on the chosen motion vector MV, a motion compensated prediction **P** is generated.
4. **P** is subtracted from the current macroblock to produce a residual or difference macroblock **D**.
5. Each sub-block is quantized (**X**).
6. The DCT coefficients of each sub-block are reordered and run-level coded.
7. Finally, the coefficients, motion vector, and associated header information for each macroblock are entropy encoded to produce the compressed bitstream.

The decoding process is as follows:

1. A compressed bitstream is entropy decoded to extract coefficients, motion vector, and header for each macroblock.
2. Run-level coding and reordering are reversed to produce a quantized, transformed macroblock **X**.
3. **X** is rescaled and inverse transformed to produce a decoded residual $\mathbf{D}'$.
4. The decoded motion vector is used to locate a 16x16 region in the decoder's copy of the reference frame $\mathbf{F}_{n-1}$. This region becomes the motion compensated prediction **P**.

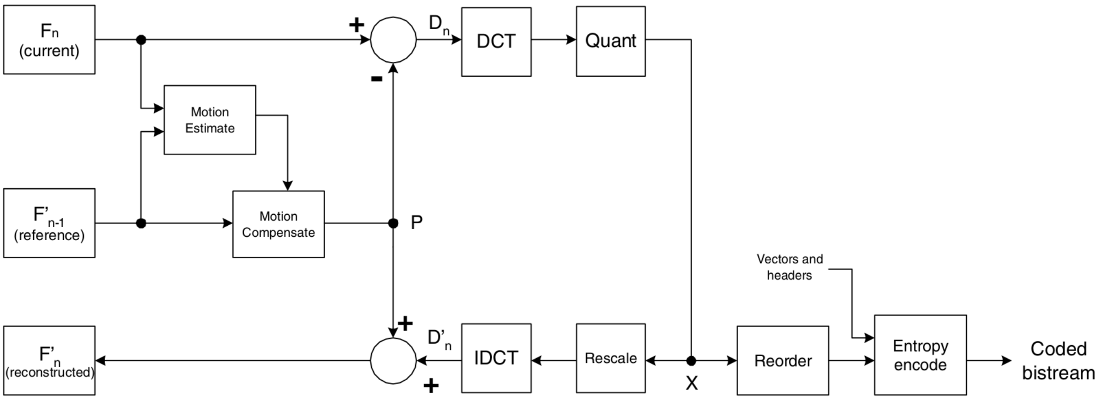


Figure 6.1: Generic DPCM/DCT video encoder used in most video coding standards.

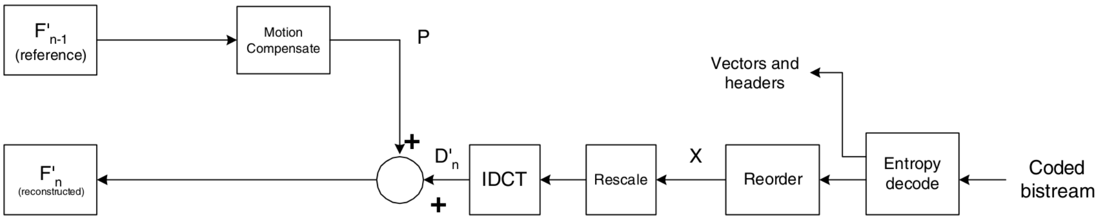


Figure 6.2: Generic DPCM/DCT video decoder used in most video coding standards.

5. **P** is added to $\mathbf{D}'$ to produce a reconstructed macroblock. The reconstructed macroblocks are saved to produce decoded frame $\mathbf{F}_n$.

Note that the encoder also requires a decoding path as shown in Figure 6.1. This is necessary to ensure that the encoder and decoder use identical reference frames ($\mathbf{F}_{n-1}$) for motion compensated prediction. The detailed description of each encoder and decoder components are dependent on the video coding standards, and can be found in [105] for MPEG-4 Part 2 Visual and MPEG-4 AVC/H.264 standards.

6.2 MPEG Reconfigurable Video Coding (RVC) Standard

The reconfigurable video coding (RVC) standard in the ISO/MPEG proposes a new paradigm for specifying and designing complex video codecs [87]. An overview of RVC framework is given in Figure 6.3. Together with the encoded video data, RVC also requires a specification of the decoder that consists of a network description (FNL) and the bit-stream syntax description (BSDL). These descriptions provide details of the video decoder that is required to decode the video data. The video decoder is first assembled from the so-called Video Tool Library

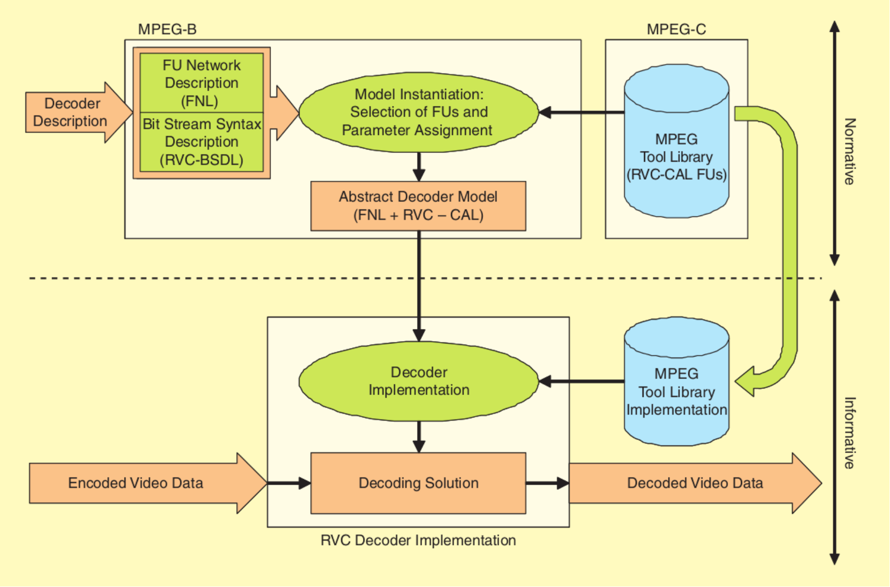


Figure 6.3: The normative and informative components of the RVC framework. The normative components are the standard languages used to specify the abstract decoder model and the standard library of the FU. The informative parts are examples of tools that synthesize a decoder implementation possibly using proprietary implementations of the standard library.

(VTL) that consists of various video decoding components (also called Functional Units (FU)) designed using the RVC-CAL language. The assembled decoder in RVC-CAL can then be synthesized to software and/or hardware implementation languages using the relevant tools. The implemented decoder can finally be used to decode the video data. It is clear that this new methodology of dynamically assembling the video decoder, instead of having a fixed implementation, allows higher degree of flexibility, reusability, and modularity across various platforms and performance requirements. Currently, the RVC standard covers two MPEG-4 codecs, the MPEG-4 Part 2 Visual codec for the Simple Profile (SP) and the MPEG-4 Advanced Video Coding (AVC)/H.264 codec for the constrained baseline (CBP) and the progressive high profiles (PHP). The HEVC codec description for the RVC standard is currently under development.

6.3 MPEG-4 Simple Profile (SP) decoder

MPEG-4 SP decoder is one of the decoder profiles in the MPEG-4 Part 2 Visual standard for decoding digital video. The standard is in fact a successor to the popular MPEG-2 standard, and offers improvements in 1) compression efficiency by making use of more advanced compression algorithms, and 2) flexibility by providing an extensive set of *tools* for coding and manipulating digital media. The grouping of the tools for a particular type of application is called *profiles*. The standard defines 21 different profiles. Some examples include those used for coding arbitrary shaped objects (Core and Main profiles), scalable objects (Core Scalable and Scalable Texture), and high-quality video (Simple Studio and Core Studio). In this thesis, one of the most popular profiles in the standard is used, the Simple profile, primarily for low bit rate and low resolution applications that are mandated by network bandwidth, device size, etc. Example are mobile phones, some low end video conferencing systems, and electronic surveillance systems.

6.3.1 Fundamentals

The MPEG-4 SP decoder is capable of decoding *video objects* using two different tools: I-VOP and P-VOP respectively for intra-coded (spatial encoding/decoding) and inter-coded (temporal encoding/decoding) rectangular Video Object Plane (VOP) in progressive format. The overview of the I-VOP and P-VOP decoding tools respectively are given in Figure 6.4 and 6.5. The details of the stages in I-VOP decoding tool is as follows:

- ***Entropy decoding: VLD and RLD***. VLD, or variable length decoding, is used to extract coefficients, motion vectors, and header for each macroblock. They are based on Huffman codes, and are defined on a pre-calculated coefficient probabilities. Frequently-occuring symbols are represented by short codes, while less common symbols are represented with long ones. The extracted information is then sent to the Run-level decoder (RLD) where a triplet of (last, run, level) is extracted. *last* indicates whether this is the final non-zero coefficient in the 8x8 block, *run* signals the number of preceding zero coefficients and *level* indicates the coefficient sign and magnitude.

- ***Reorder***. The decoded run-level is re-ordered to a macroblock representation from a zig-zag scan representation. The purpose of having a zig-zag scan representation is to efficiently group together zero-valued coefficients (which typically dominates the non-zero-valued coefficients). During this step, the *intra prediction* process may also be performed, where the DC and AC coefficient of an 8x8 block is predicted from the neighboring blocks.

- ***Inverse quantization:*** Q^{-1}. The reordered data is rescaled using the scaling parameter *QP*, which can take values from 1 to 31. Larger values of QP produce a larger quantization step size, and therefore higher compression and distortion. The typical method of performing the inverse quantization is described by the following. The DC coefficient in an intra-coded macroblock is rescaled by

$$DC = DC_Q \times dc\_scaler \tag{6.1}$$

where DC_Q is the quantized coefficient, DC is the rescaled coefficient and dc_scaler (typical value is 8) is a parameter defined in the standard. All other transform coefficients (including AC and inter DC) are rescaled as follows. If QP is odd and $F_Q \neq 0$, then $|F| = QP \times (2 \times |F_Q| + 1)$. If QP is even and $F_Q \neq 0$, then $|F| = QP \times (2 \times |F_Q| + 1) - 1$. If $F_Q = 0$, then $|F = 0|$.

- ***IDCT***. The rescaled data is sent for inverse transform using an 8x8 block to produce the decoded video data in terms of pixels. The action of the IDCT can be described by the following equation:

$$X_{ij} = \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} C_x C_y Y_{xy} \cos\frac{(2j+1)y\pi}{2N} \cos\frac{(2i+1)x\pi}{2N} \tag{6.2}$$

where **X** is the transformed block in time-domain, **Y** is the block in the frequency domain, and N is the size of the transform block. C_x and C_y are defined as follows:

$$C_i = \sqrt{\frac{1}{N}}(i = 0) \tag{6.3}$$

and

$$C_i = \sqrt{\frac{2}{N}}(i > 0) \tag{6.4}$$

Each macroblock in a coded I-VOP contains a header (defining the macroblock type, which blocks in the macroblock contain coded coefficients, signalling changes in quantization parameter, etc) followed by coded coefficients for each 8x8 block. In the decoder, the sequence of the variable-length codes are decoded to extract the quantized transform coefficients which are re-scaled and transformed by an 8x8 IDCT to reconstruct the decoded I-VOP.

The P-VOP coding tools include one additional tool called the ***motion compensation***, in addition to the tools in the I-VOP. During this step, the current frame is predicted and constructed from the previously decoded frame using a motion vector. This implies that a memory element is required to store one previously decoded frame to use for motion compensation. The default block size for motion compensation is 16x16 samples for luma, and 8x8 samples for chroma, resulting in one motion vector per macroblock. The decoded residual data is added to the prediction to reconstruct a decoded macroblock, performed by the Motion-Compensated Reconstruction (MCR). Furthermore, Macroblocks within a P-VOP may be coded in Inter mode or Intra mode. Inter mode typically give the best coding efficiency, but intra mode may be useful in regions where there is not a good match in a previous VOP, such as a newly-uncovered region.

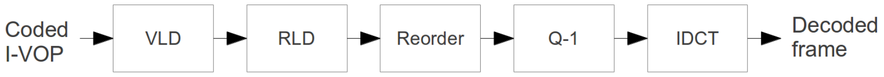


Figure 6.4: I-VOP decoding stages.

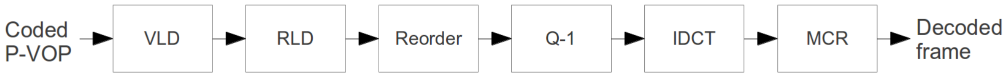


Figure 6.5: P-VOP decoding stages.

6.3.2 CAL design and implementation

The MPEG-4 SP decoder was the first video coding application specified in CAL. It was originally developed by Xilinx using a serial decoding architecture, and then modified in [83] to exploit coarse-grain parallelism for the three color spaces of Y, U, and V. The top-level overview is given in Figure 6.6. The decoder is a network formed by a set of interconnected actors. The Parser (that performs entropy decoding) is a hierarchical networks of actors, while all other blocks are atomic actors specified in CAL. Note that for readability, only one edge is represented when two actors are connected by more than one edge.

Although the decoder is the simplest one among all other profiles in the MPEG-4 Part 2 Visual standard, it is still substantially complex. The document that describes all the tools and profiles [13] in this standard run to over 500 pages, with large parts pertaining to the core encoding and decoding process utilized by all profiles. The overview of the complexity of the design is given in Table 6.1. Overall, this *original* design consists of 44 actor instances, 103 FIFO interconnections, and 349 total actions. In terms of lines of code, CAL implementation consists of 4185 lines (without blanks and comments), while the generated C and VHDL codes respectively are 12,773 and 60,212 lines, roughly 3x and 14x more. Table 6.2 presents the design complexity of the dataflow networks and sub-networks in terms of the number of code lines for the specification in XDF, which is in XML format. The overall number of lines is 579 for a total of 8 dataflow networks.

Table 6.1: Design complexity of the MPEG-4 SP decoder for each actor in terms of the number of instances, number of FIFO interconnections, number of actions, and the number of code lines (without blank and comments) in CAL, generated C, and generated HDL. The total number of FIFOs and actions are normalized to the number of instances.

Sub-network	**Actor**	**# of inst.**	**# of FIFOs**	**# of actions**	**CAL # of lines**	**Gen. C # of lines**	**Gen. HDL # of lines**
Parser	byte2bit	1	1	2	68	116	323
	parseheaders	1	1	70	1269	3841	17632
	splitter_btype	1	1	9	82	637	1384
	block_expand	1	3	5	95	255	733
	mv_sequence	1	1	11	253	527	1929
	mv_recon	1	3	19	349	1182	5185
	splitter_b	1	2	11	125	520	1757
	splitter_mv	1	1	12	50	555	1392
Texture_Y/U/V	dc_split	3	1	2	59	137	296
	dc_addr	3	1	7	231	396	2911
	inv_dc_pred	3	5	10	253	762	3020
	inv_scan	3	2	6	110	359	2066
	inv_ac_pred	3	4	5	158	298	1268
	inv_quant	3	3	3	82	257	665
	idct	3	1	8	285	467	3493
Motion_Y/U/V	motion_addr	3	2	12	380	848	3402
	picture_buffer	3	3	2	27	144	2269
	interpolation	3	2	4	133	307	1390
	add	3	3	8	131	398	1159
	Merge	3	3	3	45	206	614
TOTAL		**44**	**103**	**349**	**4185**	**12773**	**60212**

Table 6.2: Design complexity of the MPEG-4 SP decoder for each network and sub-network in terms of the number of code lines in XDF (XML), number of instances, and the total number of lines normalized to the number of instances.

Network/ Sub-network	**XDF (XML) # of lines**	**# of instances**	**Total # of lines**
Top	94	1	94
Parser	128	1	128
Texture_Y/U/V	64	3	192
Motion_Y/U/V	55	3	165
Total	**341**	**8**	**579**

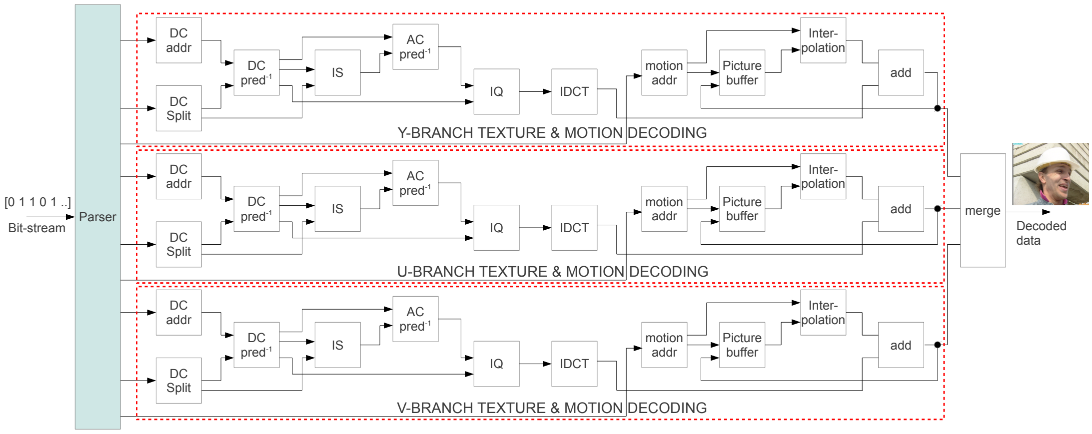


Figure 6.6: Top-level overview of the MPEG-4 SP decoder for the RVC standard. All actors are atomic, except the Parser as hierarchical networks of actors.

6.4 MPEG-4 Advanced Video Coding (AVC)/H.264 decoder

The MPEG-4 Advanced Video Coding (AVC)/H.264 decoder is currently the most commonly used format for decoding high definition video. It has also been the current video decoding standard since its completion in May 2003. The intent was to create a standard capable of providing good quality at substantially lower bit rates (i.e. higher compression ratio) than previous standards, without increasing the complexity of the design too much that it would be impractical or excessively expensive to implement. Indeed, the average compression ratio of around 60:1 is better than the MPEG-4 Part 2 Visual at 40:1, but comes at the cost of nearly 2x higher complexity. In large part, this is due to the adoption of many new technologies such as variable block size motion estimation and compensation, intra-frame prediction, integer transforms, etc. Similar to the MPEG-4 Part 2 Visual, the standard defines 21 set of profiles with different supported features for a given profile. In the present work, the Constrained Baseline Profile (CBP) is used, primarily for low cost applications such as in videoconferencing and mobile applications.

6.4.1 Fundamentals

The overview of the main stages in the MPEG-4 AVC/H.264 decoder is given in Figure 6.7. The decoder first receives a compressed bitstream from the Network Abstraction Layer (NAL) (with additional information for transmission and storage), and then entropy decodes the data elements to produce a set of quantized coefficients $\mathbf{X}$. These are scaled and inverse transformed to give $\mathbf{D}'_n$. Using the header information decoded from the bitstream, the decoder creates a prediction block $\mathbf{P}$. Finally, $\mathbf{P}$ is added to $\mathbf{D}'_n$ to produce $u\mathbf{F}'_n$, which is filtered

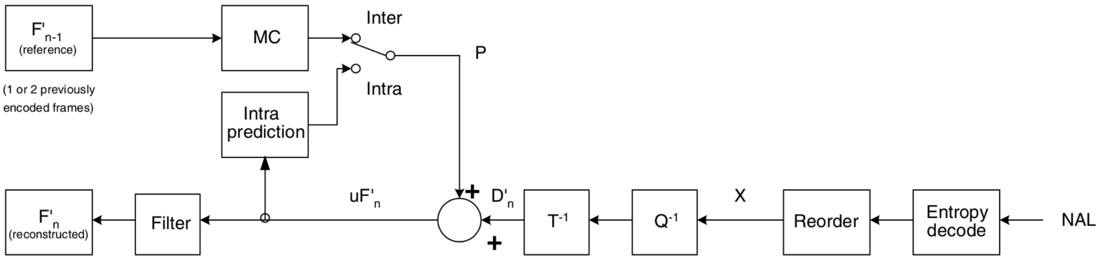


Figure 6.7: Overview of the main stages in the MPEG-4 AVC/H.264 decoder.

to create each decoded block $\mathbf{F}'_n$. Note that the decoding procedure is *almost* similar to the previous standards. In the context of the CBP profile, the difference mainly lies in three aspects: 1) the use of the concept of *slice*, where a slice is a set of macroblocks in raster scan order, 2) the use of a deblocking filter to smooth out the decoded picture, and 3) the details of the functional blocks. The following provides further details on each of these features.

In contrast to the MPEG-4 Part 2 Visual standard that decodes one full video frame either using I-VOP or P-VOP, the MPEG-4 AVC/H.264 standard allows a single video frame to be decoded using both intra and inter predictions. For this, a group of macroblocks in a single video frame is partitioned into slices, that could either be an *I slice* or a *P slice*. An I slice may contain only I macroblock types that are predicted using intra prediction from decoded samples in the current slice. A P slice may contain P and I macroblock types, where a P type is predicted using inter prediction from one or more reference picture(s). Furthermore, an inter-coded macroblock may be divided into macroblock partitions, i.e. blocks of size 16x16, 16x8, 8x16, or 8x8 luma samples (and associated chroma samples). If the 8x8 partition size is chosen, each 8x8 sub-macroblock may be further divided into sub-macroblock partitions of size 8x8, 8x4, 4x8 or 4x4 luma samples (and the associated chroma samples). The different block size is often referred to as the Variable-block size (VBS).

The next main feature of the MPEG-4 AVC/H.264 decoder is the deblocking filter for reducing blocking distortion. It is applied after the inverse transform to smooth block edges and improve the appearance of decoded frames. Filtering is applied to vertical and/or horizontal edges of a 4x4 blocks in a 16x16 macroblock (except for edges on slice boundaries), first in the vertical direction, then the horizontal direction. The important filter parameter is the *boundary strength, bS*. Typical values range from 0 for no filtering, to 4 for strongest filtering. The standard defines a set of rules for choosing the bS parameter (in the encoder), but essentially, filtering is designed to be stronger at places where there is likely to be significant blocking distortion, such as the boundary of an intra coded macroblock or a boundary between blocks that contain coded coefficients.

The following presents details of other functional blocks in Figure 6.7:

- ***Entropy decoding***. The entropy decoder used in the CBP profile is called the Context-Based Adaptive Variable Length Coding (CAVLC). The method is applied on the decoding side to produce a zig-zag ordered 4x4 (and 2x2) blocks of transform coefficients. Similar to previous standards, it uses a run-level coding to represent strings of zeros compactly. The zig-zag scan are often sequences of ±1, and CAVLC signals the number of high-frequency ±1 coefficients ('Trailing Ones') in a compact way. CAVLC is also designed to take advantage of the correlation in the nonzero coefficients in neighboring blocks. For this, the number of coefficients is encoded using a look-up table and the choice of look-up table depends on the number of nonzero coefficients in the neighboring blocks. Besides this, CAVLC also takes advantage of the nonzero coefficients that tends to be larger at the start of the reordered array (near DC coefficient), and smaller towards the higher frequencies. For this, it adapts the choice of VLC look-up table for the level parameter depending on recently coded level magnitudes.

- ***Reordering***. The reordering process in the decoder is similar to the previous standard as explained in Section 6.3.1, i.e. from a zig-zag scan representation to a macroblock representation.

- ***Transform and quantization***. The MPEG-4 AVC/H.264 decoder uses three transforms depending on the type of residual data that is to be coded: a Hadamard transform for the 4x4 array of luma DC coefficients in intra macroblocks predicted in 16x16 mode, a Hadamard transform for the 2x2 array of chroma DC coefficients (in any macroblock), and a DCT-based transform for all other 4x4 blocks in the residual data. The DCT transform is slightly different than the one in the previous standard, where 1) it is an integer transform where all operations can be carried out using integer arithmetic, without loss of decoding accuracy, 2) the core part can be implemented using only additions and shifts, and 3) scaling multiplication is integrated into the quantizer, reducing the total number of multiplications. The IDCT in the MPEG-4 AVC/H.264 standard is defined explicitly using a sequence of arithmetic operations with the following:

$$\mathbf{Y} = \begin{bmatrix} 1 & 1 & 1 & 0.5 \\ 1 & 0.5 & -1 & -1 \\ 1 & -0.5 & -1 & 1 \\ 1 & -1 & 1 & -0.5 \end{bmatrix} \left(\begin{bmatrix} \mathbf{X} \end{bmatrix} \otimes \begin{bmatrix} a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \\ a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \end{bmatrix} \right) \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0.5 & -0.5 & -1 \\ 1 & -1 & -1 & 1 \\ 0.5 & -1 & 1 & -0.5 \end{bmatrix} \tag{6.5}$$

where **Y** is the transformed block in time-domain from a block in frequency-domain, **X**, with $a = 0.5$ and $b = \sqrt{\frac{2}{5}}$. The Hadamard transform at the decoder is defined as follows:

$$\mathbf{W}_{QD} = \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \begin{bmatrix} \mathbf{Z}_D \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \right) \tag{6.6}$$

where $\mathbf{W}_{QD}$ is a transformed block in time-domain block for a 4x4 luma DC coefficient, and $\mathbf{Z}_D$ is a block in the frequency-domain. A similar 2x2 matrix is used for transforming a 2x2 chroma DC coefficient.

The basic inverse quantization operation is that of rescaling the coefficients using a quantization parameter, *QP*. In addition, the MPEG-4 AVC/H.264 standard defines two additional parameters, Q_{step} and *PF*, where $\mathbf{V} = Q_{step} \times PF \times 64$ is defined for $0 \leq QP \leq 5$ for each coefficient position so that scaling becomes:

$$\mathbf{W}'_{ij} = \mathbf{Z}_{ij}\mathbf{V}_{ij} \times 2^{floor(QP/6)} \tag{6.7}$$

- ***Intra prediction***. In intra decoding mode, a prediction block **P** is formed based on previously reconstructed blocks, and is added to the transformed block for filtering. For the luma samples, **P** is formed for each 4x4 block or for a 16x16 macroblock. There are a total of nine optional prediction modes for each 4x4 luma block, four modes for a 16x16 luma block and four modes for the chroma components. During encoding, the encoder selects the prediction mode for each block that minimizes the difference between **P** and the block to be encoded. At the decoding side, the current block is predicted using this prediction mode.

- ***Inter prediction***. The inter prediction process is similar to previous standards (MPEG-4 Part 2 Visual SP), but with two notable differences. The first is the variable-block size from 16x16 down to 4x4, and the second is the use of quarter-pixel sub-sample prediction. In the former, large partition size is appropriate for homogeneous areas of the frame, while small partition size may be beneficial for details areas. In the latter, quarter-pixel interpolation results in higher design complexity, but offers better compression performance. The samples at half sample positions is decoded by applying a 6-tap filter to the nearest integer position in the vertical or horizontal direction, for example,

$$b = Clip((E - 5 \times F + 20 \times G + 20 \times H - 5 \times I + J + 16) >> 5) \tag{6.8}$$

where b is the half pixel sample, and E, F, G, H, I, and J are the integer samples. The *Clip* function clips the value in the range from 0 to 255. The samples at quarter sample positions is decoded by applying a 2-tap filter to the nearest integer position in the vertical or horizontal direction, for example,

$$a = (G + b + 1) >> 1 \tag{6.9}$$

where a is the quarter pixel sample, G is the integer sample, and b is the half pixel sample. Furthermore, the MPEG-4 AVC/H.264 standard also supports encoding motion vector for each partition. This takes the advantage that neighboring partitions are often highly correlated and so each motion vector is predicted from vectors nearby, i.e. previously decoded partitions.

6.4.2 CAL design and implementation

The design of the MPEG-4 AVC/H.264 CBP decoder for the RVC standard was first introduced in [44]. Figure 6.8 presents the top-level overview of the video decoder. Similar to the CAL design of the MPEG-4 SP decoder, the MPEG-4 AVC/H.264 CBP decoder is designed to exploit coarse-grain parallelism for the three color spaces of Y, U, and V. The Y-branch decoder consists of a residual part (transform and quantization), intra prediction part (for block sizes of 16x16 or 4x4), inter prediction with half and quarter pixel interpolation, and deblocking filter. The U/V-branch decoder is similar, except that the intra prediction part supports block size of 8x8, and inter prediction part with bilinear pixel interpolation.

The document that describes all the tools and profiles [12] in the MPEG-4 AVC/H.264 standard runs to almost 300 pages, with large parts pertaining to the core encoding and decoding process utilized by all profiles, including the CBP profile used in this work. The overview of the complexity of the CAL design is given in Table 6.3. Overall, this *original* design consists of 90 actor instances, 268 FIFO interconnections, and 1282 total actions. In terms of lines of code, the CAL implementation consists of 20,562 lines, while the generated C and VHDL codes respectively consist of 64,828 and 353,538 lines, roughly 3x and 17x more. Table 6.4 presents the design complexity of the dataflow networks and sub-networks in terms of the number of code lines for the specification in XDF, which is in XML format. The overall number of lines is 4131 for a total of 27 dataflow networks.

Table 6.3: Design complexity of the MPEG-4 AVC/H.264 decoder for each actor in terms of the number of instances, number of FIFO interconnections, number of actions, and the number of code lines (without blank and comments) in CAL, generated C, and generated HDL. The total number of FIFOs and actions are normalized to the number of instances.

Sub-network	Actor	# of inst.	# of FIFOs	# of actions	CAL # of lines	Gen. C # of lines	Gen. HDL # of lines
Parser	algo_synp	1	1	35	4781	11066	81014
	intra_split	1	4	8	278	542	4304
	block_exp	1	4	23	420	1144	4563
	block_split	1	2	15	211	794	3593
	mmco	1	6	23	479	1295	8347
	reflist	1	10	13	609	1160	11554
	mgnt_interpred	1	5	1	69	328	1052
	mvlx_reconstr	1	7	38	1805	4903	34907
	refidx_frame_num	1	3	5	60	254	782
	mv_reord	1	5	14	259	917	2726
Res_Y	is_zigzag_4x4	2	1	30	211	1052	3466
	hadamard_1ht1d	2	1	8	110	358	1055
	transpose_4x4	4	1	31	220	1079	3586
	hadamard_scale	1	3	20	298	1101	5514
	hadamard_reord	1	1	30	211	1052	3465
	intra_16x16	1	3	10	398	487	1305
	scaler_4x4	1	3	17	297	1149	7939
	it_4x4_1d	2	1	8	110	358	1057
	it_addshift	1	1	1	52	89	157
	mgnt_4_16	1	1	5	100	239	1414
Res_U/V	hadamard_chroma	2	2	2	90	190	1489
	is_zigzag_4x4	2	1	30	211	1052	3471
	iq_chroma	2	2	4	194	229	604
	scaler_4x4	2	3	5	194	5436	5243
	it_4x4_1d	4	1	8	112	358	1057
	it_addshift	2	1	1	51	89	157
	transpose_4x4	4	1	31	220	1079	3586
	mgnt_4_8	2	1	5	86	236	1234
Pred_Y	picture_buffer	1	14	108	1259	4413	18846
	parser_info	1	5	19	292	1096	4793
	neighbour_16x16	1	3	10	232	699	5253
	mgnt_16x16	1	2	16	127	735	2797
	intra_16x16	1	5	17	398	813	5923
	add_16x16	1	2	1	67	126	209
	neighbour_4x4	1	3	12	269	572	4236
	mgnt_4x4	1	4	18	458	1243	9456
	intra_4x4	1	5	22	464	1268	13203

	split_16_to_4	1	1	4	72	200	1002
	add_4x4	1	2	1	67	126	209
	mgnt_4_16	1	1	4	72	200	1001
	hq_interp	1	5	9	370	837	8802
	blocks_reorder	1	3	20	464	986	5471
	add_inter	1	2	1	67	126	209
	select	1	4	9	97	372	911
	deb_filter	1	3	7	168	422	3853
Pred_U/V	picture_buffer	2	12	22	433	4413	10194
	parser_info	2	4	12	185	720	3704
	neighbour_8x8	2	3	9	234	570	4320
	mgnt_8x8	2	2	11	197	602	3071
	intra_8x8	2	5	15	434	866	6357
	add_8x8	2	2	1	67	126	209
	bilinear_interp	2	5	7	186	425	1939
	blocks_reorder	2	3	20	442	986	5384
	add_inter	2	2	1	67	126	209
	select	2	3	6	71	268	674
	deb_filter	2	3	9	201	612	4895
Merger	display_poc	1	4	9	169	538	3411
	render_y	1	4	14	248	671	7841
	render_u	1	4	14	246	664	7888
	render_v	1	4	14	246	664	7888
	merger	1	3	3	57	307	739
TOTAL		**90**	**268**	**1282**	**20562**	**64828**	**353538**

6.5 Conclusion

In this chapter, the fundamentals of video codecs have been presented, followed by an approach on how the codecs can be specified in a new way using the RVC standard. This is followed by the two MPEG-4 decoder case studies that are used in this work: the MPEG-4 Part 2 Visual SP decoder, and the MPEG-4 AVC/H.264 CBP decoder. It is shown that fundamentally, both are very complex systems with extensive processing requirements. This is also apparent in the CAL design and implementation of these decoders, with close to 100 actors and more than 1200 actions in the case of MPEG-4 AVC/H.264 decoder. In the next section, We show how the appropriate combinations of the refactoring and optimization techniques presented in the previous three chapters can be used to rapidly and effectively explore the design and implementation space of these complex decoders.

Table 6.4: Design complexity of the MPEG-4 AVC/H.264 decoder for each network and sub-network in terms of the number of code lines in XDF (XML), number of instances, and the total number of lines normalized to the number of instances.

Network/ Sub-network	**XDF (XML) # of lines**	**# of instances**	**Total # of lines**
Top	280	1	280
Parser	348	1	348
Cavlc_expand	88	1	88
Gen_inter_info	277	1	277
Decoder_Y	292	1	292
Decoder_U/V	264	2	528
Prediction_Y	240	1	240
Prediction_U/V	208	2	416
Residual_Y	125	1	125
Residual_U/V	81	2	162
DCR	86	1	86
IS_IQ_Y	87	1	87
IT4x4	50	3	150
I_IQ_U/V	81	1	81
Intra16x16	120	1	120
Intra4x4	145	1	145
Inter_Y	128	1	128
Intra8x8	118	2	236
Inter_U/V	113	2	226
Disp_renderer	116	1	116
Total	**3247**	**27**	**4131**

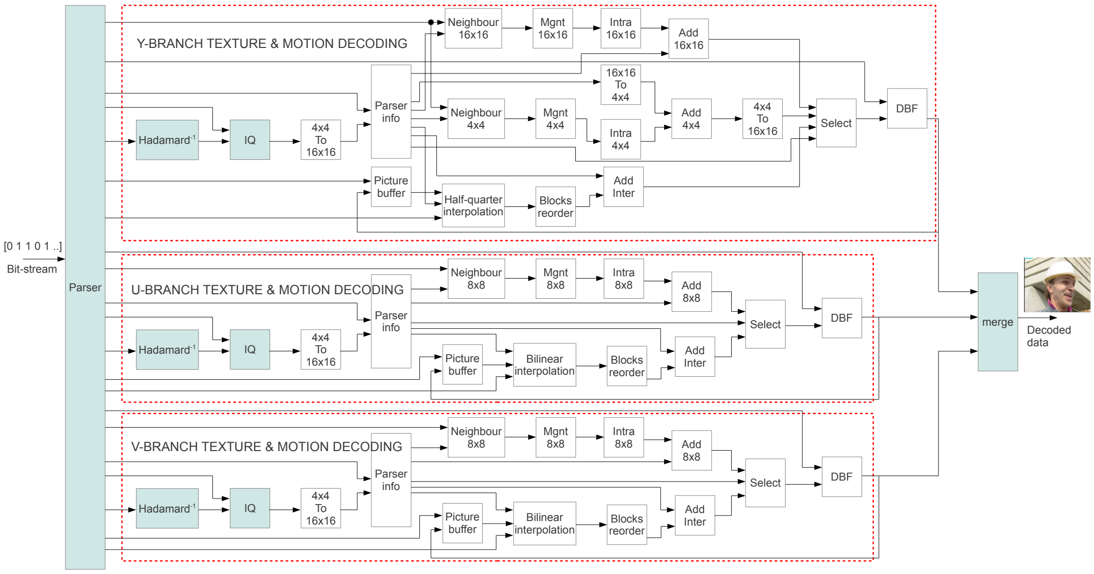


Figure 6.8: Top-level overview of the MPEG-4 AVC/H.264 CBP decoder for the RVC standard. All blocks are atomic actors, except the Parser, inverse Hadamard transform, inverse quantization (IQ), and the merger. These represent a hierarchical networks of actors.

7 Multi-dimensional design space exploration

In this chapter, a methodology to explore and evaluate design points in a multi-dimensional space is presented. The design points, or sometimes referred to as design alternatives, are obtained by appropriate combinations of the refactoring and buffer-size minimization and optimization techniques presented in chapters 3, 4, and 5. These techniques are applied on complex MPEG-4 decoders presented in chapter 6. For complex designs that typically results in many design points with multiple performance criteria, the task of analyzing the points for feasibility is not a straightforward task, especially for a multi-objective requirement. We also provide in this chapter, a systematic methodology for design space exploration based on an automatic analysis of the design points such that various performance criteria can be explored graphically on multi-dimensional plots, and design point(s) can be selected based on multiple user requirements.

This chapter is organized as follows. The first section presents background and related works in design space exploration, both in general terms and then specifically for embedded systems and signal processing. The following section then presents definitions and metrics used in this work for design space exploration and evaluation. This is followed by an overview of the data analysis tool to automatically evaluate the design points in the exploration space. The next two sections present experimental results of applying the refactoring and buffer size minimization techniques, and the corresponding space exploration and evaluation for two design case studies: the MPEG-4 SP decoder, and the MPEG-4 AVC/H.264 decoder. Next, comparison of our results to similar works in literature is provided, and finally, the last section presents summary of the chapter.

7.1 Background and related works

Design space exploration (DSE) refers to the activity of discovering and evaluating design alternatives during system development [68]. The task of DSE is typically divided into two parts. First is to obtain various design alternatives from a reference design, either automatically (for analyzable part of design), and/or manually for evaluating different types of architecture. The design alternatives for exploration typically spans multiple design objectives and criteria, such as latency, frequency, memory requirement, power, and area. The second task involves finding and eliminating inferior designs from various design alternatives, and collecting a set of final candidates that can be further studied. The analysis of the design alternatives can also be used to find design configurations that satisfy a set of constraints. The exploration and evaluation of the design points for multiple objectives is called multi-dimensional DSE.

The work on DSE is not only limited to embedded systems and signal processing, but spans a wide variety of applications in many different discipline from chemical to civil engineering, and economics to mechanical engineering. The commonly used notion in finding an optimal set from various alternatives is called the *Pareto-optimal*, and is in fact derived from the study of economics (defined in the next section). The goal of performing DSE however, is common across all disciplines, that is to find design point(s) in the exploration space that optimizes a given set of objectives.

In the area of embedded systems and signal processing, the method to obtain the design points are mostly performed semi- or fully-automatically so as to obtain as many different (mostly feasible) design points as possible in a short period of time. The work in [74] explores alternative design points based on software/hardware implementation choices which include system partitioning and processor architecture. A design is specified using an SDF network, and the subgraphs are explored for implementation using various number of embedded processors and hardware IP cores. In a different approach, the work in [101] utilizes C and SystemC for exploring different software and hardware partitioning schemes. Several platforms are compared which include DSP and embedded processors, application specific instruction processor, FPGAs (with soft-core control), and ASICs (with ARM control). Another approach in [18] is based on an library-based interactive user interface where designers could adjust the architectural definition using constraints and library components. A system architecture is generated that meets the constraints which include number and type of processors, access time of memories, contribution of co-processors, DMA or CPU controlled transfers, etc. The work that is more closely related to the work in this thesis is given in [56] where various hardware implementation design point is explored. The design points are obtained by exploring different pre-defined architectures with different performance parameter values. The authors presented 16 different FIR filter structures with 8 different operating voltage values for each structure for exploring throughput, area, and power. Since the design is a relatively simple FIR filter, HDL was used as design specification; in contrast to this work, a high-level design methodology is used for the exploration and evaluation of a substantially more complex MPEG-4 decoders.

For CAL programs, the first work that was reported for DSE is given [84]. It describes what CAL designers need to do in order to move in the design space. The implementation target is for software implementation, where each point in the design space represents a triplet of a CAL program (architecture), a schedule, and a partition. The various permutations of the triplet represent design points in the exploration space, where performance criteria such as resource and throughput can be evaluated. This work has been extended in [88] where in addition to the triplets, buffer dimensioning has also been used to explore the design space. The DSE techniques from previous works is software platform dependent, and has to be adapted differently for hardware platform target. In the next section, the properties of a design point is formally defined for the case of hardware implementation.

Once the design points have been obtained, they can be analyzed using some data visualization tools such as *Spotfire*, *VisDB*, and *CViz* [123]. Within each tool, one can arbitrarily choose which dimensions of the data to visualize, and assign them 3D coordinate axes or other methods of indicating value to see relationships and correlation between the criteria. One can also dynamically constrain the data by removing points that do not satisfy some set criteria through a process called *brushing*. Instead of utilizing these tools, we have developed our own tool for this purpose, which is described in Section 7.3.

7.2 Metrics for design space exploration

Before presenting and analyzing the design points in the exploration space for our design case studies, the terms and notations used for our methodology for DSE are defined as follows.

Definition 1. *(Design Point)* A design point is a point in a n-dimensional space $\mathbf{E}^n$, where $D_m = (P_m, C_m), m \in M$ is a single point from $|M|$ number of points. Each point $D_m = (P_m, C_m)$ is associated with parameter $P_m = (A_m, F_m, B_m)$ and criteria $C_m = (LAT_m, FRE_m, THR_m, REG_m, LUT_m, BRM_m, SLI_m, DSP_m)$ that defines the property of the design point m. The range of the dimension of the exploration space n is from 1 to $|C|$, where $|C|$ is the cardinality of design criteria.

Definition 2. *(Design parameter)* A design parameter for a given point D_m, $P_m = (A_m, F_m, B_m)$ consists of an architectural definition A_m, pipelining sequence F_m, and buffer size configuration B_m. The parameter A is obtained from refactoring for latency (chapter 3), parameter F from refactoring for frequency (chapter 4), and parameter B from buffer size minimization and optimization (chapter 5).

Definition 3. *(Design criteria)* A design criteria for a given point D_m, $C_m = (LAT_m, FRE_m, THR_m, REG_m, LUT_m, BRM_m, SLI_m, DSP_m)$ consists of a latency LAT_m, measured by the number of clock cycles per video frame (typically QCIF, unless otherwise stated); maximum operating frequency FRE_m, measured in MHz; maximum throughput $THR_m \propto \frac{FRE_m}{LAT_m}$, measured by the frame rate (typically the number of QCIF frames per second (fps), unless otherwise stated); the number of slice registers REG_m; the number of slice LUT LUT_m, where each slice contains four LUTs and four flip-flops (for Xilinx Virtex-5); the number of 36Kb Block RAM BRM_m; the number of occupied slice, $SLI_m = \frac{max(REG_m, LUT_m)}{4}$; and the number of DSP48E slice DSP_m, where each slice contains a 25x18 multiplier, an adder, and an accumulator.

A design point in a multi-dimensional space consists of $|P|$ parameters (that can be configured/modified by a user) with $|C|$ design performance criteria, i.e. $f : \mathbb{R}^{|P|} \rightarrow \mathbb{R}^{|C|}$. The evaluation of design points is based on the following definitions:

Definition 4. *(Dominance relation)* Let $f, g \in D$ be two design points in the exploration space. Then f is said to dominate g denoted as $f \succ g$, iff

1. $\forall m \in M : f_m \geqslant g_m$
2. $\exists k \in M : f_k > g_k$

Definition 5. *(Pareto set)* Let $F \subseteq D$ be a set of vectors. Then the Pareto set $F^* \subseteq F$ is defined as follows : F^* contains all vectors $g \in F$ which are not dominated by any vector $f \in F$, i.e.

$$F^* = \{g \in F | \nexists f \in F : f \succ g\} \tag{7.1}$$

The Pareto set can then be used to construct the *Pareto Frontier* in the exploration space that represents the boundary of the non-dominated design points, i.e. points that is more preferred in all criteria. Conversely, a dominated point is one for which there exists another point that is equal or better in all performance criteria and strictly better in at least one. The graph in Figure 7.1 shows an example of six design points for the evaluation of resource and throughput criteria. We can see that the points D_1 and D_2 are dominated by the point D_3 since it could achieve the throughput of the dominated points using less resources. Similarly, the points D_5 and D_6 dominate the point D_4. The Pareto set is therefore $\{D_3, D_5, D_6\}$. The two criteria Pareto set in this example can also be extended to multiple criteria by evaluating design points in a multi-dimensional space.

Another important evaluation when analyzing design points are the lower and upper bounds for a given criteria. Indeed, they represent the dynamic range for each criteria. The lower

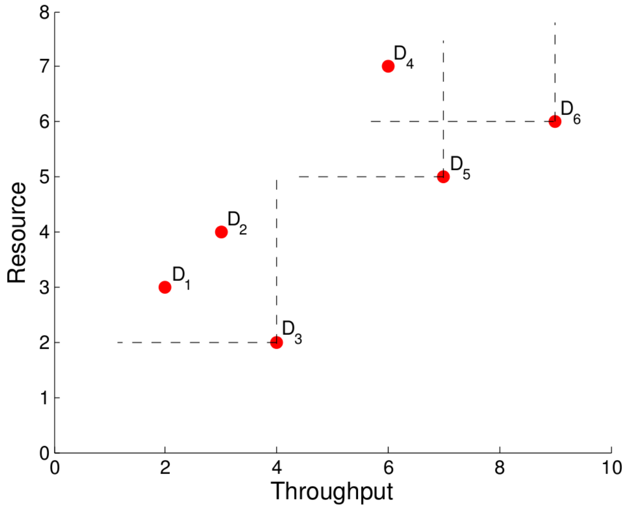


Figure 7.1: Example of analyzing six design points for Pareto set. The non-dominated points are $\{D_3, D_5, D_6\}$.

bound is also known as the *Nadir objective vector* given by:

$$Z_i^{nad} = \max_{x \in D} f_i(x), i \in (1, 2, \ldots, |C|) \tag{7.2}$$

Similarly, the lower bound is known as the *ideal objective vector*, given by:

$$Z_i^{ideal} = \min_{x \in D} f_i(x), i \in (1, 2, \ldots, |C|) \tag{7.3}$$

where $f_i(x)$ is the function that maps a design point x to the value of criteria i. In the example in Figure 7.1 with $i = 0$ for resource and $i = 1$ for throughput, $Z_0^{nad} = 7$, $Z_0^{ideal} = 2$, $Z_1^{nad} = 9$, and $Z_1^{ideal} = 2$. The dashed line represents the Pareto frontier.

7.3 Methodology for automatic data analysis

A tool has been developed to automatically analyze the design points such that exploration and evaluation can be done systematically and efficiently. The tool has been developed using Java, with an overview given in Figure 7.2. It takes in the design points represented by the design parameters and criteria. Users could specify three additional controls, where reports

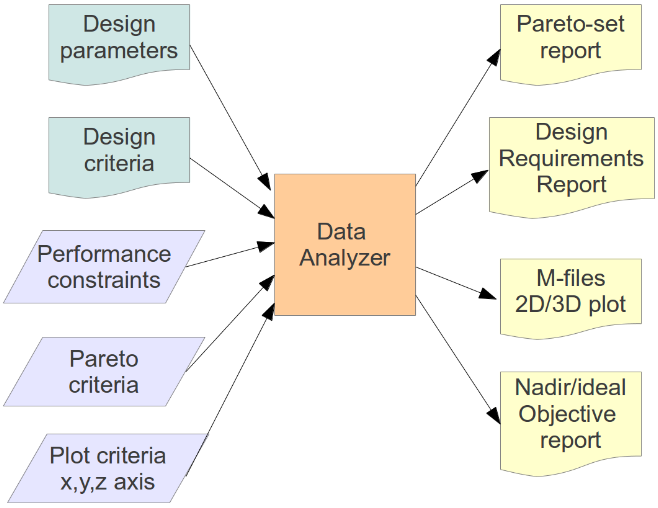


Figure 7.2: Overview of the data analyzer tool to systematically and efficiently evaluate the design points in the exploration space.

are generated based on a set of specification:

1. **Performance constraint**. Users can set the required value(s) for at least one of the optimization criteria. An error would be thrown if the set value is out of the upper and/or lower bounds of the criteria. If this option is activated, a design requirement report is generated with feasible design point(s) for the set of constraint(s).
2. **Pareto criteria**. Users can set the design criteria to find the Pareto set for a single- or multi-objective criteria. If this option is activated, the Pareto-set report is generated.
3. **Plot criteria: x,y,z axis**. Users can set which criteria to assign to which axis for plotting either 1-, 2-, or 3-dimensional scatter plot. For 3-dimensional graph, a stem plot is given to provide a clear visualization of results. If this option is activated, MATLAB m-file(s) are generated for the corresponding n-dimensional plot, where the file(s) can be run directly in MATLAB to generate the graph.

7.4 Case study-1: MPEG-4 SP decoder

The specification of the original design of the MPEG-4 SP decoder in CAL is given in Section 6.3. Starting from this original design, other design points are obtained by dataflow refactoring and buffer size minimization and optimization such that some design criteria can be improved. All the alternative CAL design specifications of the decoder are synthesized to HDL for implementation on Xilinx Virtex-5 FPGA to obtain design performance criteria FRE, REG, LUT, BRM, SLI, and DSP as defined in Section 7.2. The criteria LAT and THR are obtained from a hardware simulation using Modelsim. Simulation is performed only using the *Foreman* QCIF (176x144 pixels) video sequence. The inter-prediction frames are stored using local/on-chip block RAM. For higher resolution designs such as HD720p with 1280x720 pixels per frame where block RAM is not feasible to be used, an external memory implementation is required together with the design.

Table 7.1 presents 38 design points for the decoder case study with the corresponding design parameters and criteria. The strategy employed to obtain the design points are as follows. The original design point D_0 is first re-factored for latency by memory optimizations and task parallelism. This is performed for two different buffer size configurations: *HEM* (for design points D_1 to D_3) and *HEO* (for design points D_4 to D_7) as explained in chapter 5. The design points on the edges (D_0, D_3, D_4, and D_7) are each taken for refactoring for frequency, which corresponds respectively to design points D_8 to D_{14}, D_{15} to D_{21}, D_{22} to D_{28}, and D_{29} to D_{35}. The final two design points, D_{36} and D_{37} are obtained by reducing the operating frequency of the original design point D_0 for lower throughput requirement, which we call the *frequency-reduction* technique. The details of the design points for latency refactoring, frequency refactoring, and buffer size optimization respectively are given in Tables 7.2, 7.3, and 7.4.

Table 7.1: Design points and the corresponding parameters and criteria for the MPEG-4 SP decoder case study. The units for LAT, FRE, and THR respectively are clock cycles per QCIF frame, MHz, and QCIF frames per second.

Design point	**Parameters**			**Criteria**							
	A	F	B	LAT	FRE	THR	LUT	REG	BRM	SLI	DSP
D_0	0	0	0	183123	42	230	18917	22536	92	5634	18
D_1	1	0	0	121025	43	353	19280	23525	92	5881	18
D_2	2	0	0	102635	43	414	19555	24344	93	6086	18
D_3	3	0	0	94341	43	455	20004	25368	98	6342	18
D_4	0	0	1	140787	42	299	20070	24336	99	6084	18
D_5	1	0	1	77783	43	549	21252	27166	102	6792	18
D_6	2	0	1	66573	43	638	22394	29342	105	7336	18
D_7	3	0	1	63248	43	678	23754	31296	109	7824	18
D_8	0	1	0	183123	76	416	18917	22693	92	5673	18
D_9	0	2	0	183123	80	436	19056	23733	92	5933	18
D_{10}	0	3	0	183123	81	440	19525	23981	92	5995	18
D_{11}	0	4	0	183123	90	490	19510	24138	92	6035	18
D_{12}	0	5	0	183123	90	490	20066	25430	92	6358	18
D_{13}	0	6	0	183123	95	518	20218	25560	92	6390	18
D_{14}	0	7	0	183123	96	524	20414	26687	92	6672	18

D_{15}	3	1	0	94341	76	801	20207	25515	98	6379	18
D_{16}	3	2	0	94341	78	825	20676	26646	98	6662	18
D_{17}	3	3	0	94341	83	878	20813	26814	98	6704	18
D_{18}	3	4	0	94341	90	951	20800	26995	98	6749	18
D_{19}	3	5	0	94341	90	952	21356	28250	98	7063	18
D_{20}	3	6	0	94341	95	1008	21510	28347	98	7087	18
D_{21}	3	7	0	94341	96	1019	21723	29336	98	7334	18
D_{22}	0	1	1	140787	76	537	20271	25335	99	6334	18
D_{23}	0	2	1	140787	76	538	20410	26408	99	6602	18
D_{24}	0	3	1	140787	80	569	20879	26614	99	6654	18
D_{25}	0	4	1	140787	90	636	20864	26785	99	6696	18
D_{26}	0	5	1	140787	90	639	21421	28012	99	7003	18
D_{27}	0	6	1	140787	90	639	21573	28102	99	7026	18
D_{28}	0	7	1	140787	96	680	21654	29109	99	7277	18
D_{29}	3	1	1	63248	76	1195	23957	31354	109	7839	18
D_{30}	3	2	1	63248	83	1306	24426	32470	109	8118	18
D_{31}	3	3	1	63248	88	1393	24563	32665	109	8166	18
D_{32}	3	4	1	63248	89	1413	24550	32730	109	8183	18
D_{33}	3	5	1	63248	90	1421	25106	34068	109	8517	18
D_{34}	3	6	1	63248	90	1429	25258	34183	109	8546	18
D_{35}	3	7	1	63248	96	1519	26835	35132	109	8783	18
D_{36}	0	8	0	183123	5	30	18917	22536	92	5634	18
D_{37}	0	9	0	183123	22	120	18917	22536	92	5634	18

Table 7.2: Specific refactoring for latency applied on the MPEG-4 SP decoder case study. The unit for latency is clock cycles per macroblock. The techniques are applied cumulatively from D_0/D_4 to D_3/D_7.

Design points	**Technique**	**Actor**	**Latency (C.C./MB)**
D_0/D_4	*original*	-	1850/1422
D_1/D_5	Data-packing	picture_buffer	1222/786
D_2/D_6	Data-parallelism	inverse_scan	1037/672
D_3/D_7	Data-parallelism	inverse_ac_pred	953/639

Each design criteria can also be evaluated for upper and lower bounds to get an overview of the range that could be set. Table 7.5 presents the range for each criteria for the design case study.

Figure 7.3 shows a 3-dimensional graph for three major performance criteria of occupied slice (SLI), throughput (THR), and frequency (FRE) for the design case study. As shown on the graph, refactoring for latency (green/circle and blue/square points) results in throughput

Table 7.3: Specific refactoring for frequency applied on the MPEG-4 SP decoder case study. Design points D_{36} and D_{37} in Table 7.1 refers to the frequency-reduction technique.

Design points	**Actor(s)**	**Action(s)**	**# of pipeline stages**	$\overline{f_{max}}$**(MHz)**
D_0	*original*	-	-	42
$D_8/D_{15}/D_{22}/D_{29}$	inverse_dc_pred	read_intra	2	76
$D_9/D_{16}/D_{23}/D_{30}$	picture_buffer	read_address	2	79
$D_{10}/D_{17}/D_{24}/D_{31}$	inverse_dc_pred	read_intra	3	85
$D_{11}/D_{18}/D_{25}/D_{32}$	inverse_dc_pred	read_intra	4	90
$D_{12}/D_{19}/D_{26}/D_{33}$	inverse_quant	ac	2	90
$D_{13}/D_{20}/D_{27}/D_{34}$	inverse_dc_pred	getdc_inter	2	93
$D_{14}/D_{21}/D_{28}/D_{35}$	idct	calc_row	2	96

Table 7.4: Specific buffer size optimization technique for each design point of the MPEG-4 SP decoder case study.

Design points	**Technique**
$D_0 - D_3, D_8 - D_{21}, D_{36}, D_{37}$	Hardware Execution Minimization (HEM)
$D_4 - D_7, D_{22} - D_{35}$	Hardware Execution Optimization (HEO)

Table 7.5: Nadir and ideal objective vectors for each design criteria for the MPEG-4 SP decoder case study. The units for LAT, FRE, and THR respectively are clock cycles per QCIF frame, MHz, and QCIF frames per second.

	Design Criteria							
	LAT	FRE	THR	REG	LUT	BRM	SLI	DSP
z^{nad}	63248	5	30	18197	22536	92	5634	18
z^{ideal}	183123	96	1519	26835	35132	109	8783	18

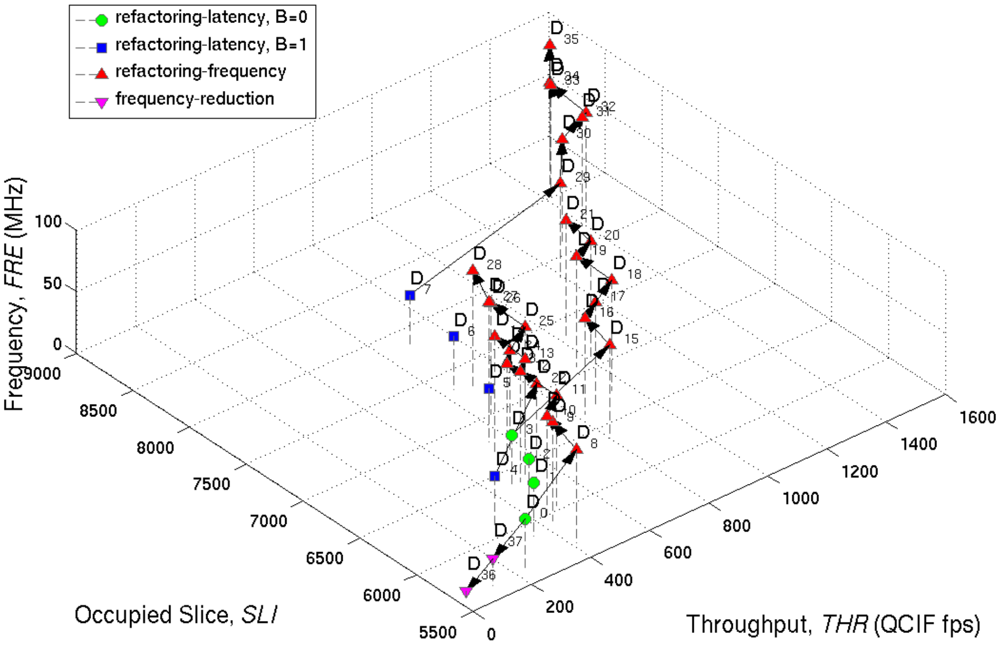


Figure 7.3: 3D plot of frequency, occupied slice, and throughput for MPEG-4 SP decoder case study.

improvement, without an increase in the operating frequency. In contrast, pipelining improves throughput by means of a higher operating frequency (red/upward triangle points). It is also interesting to see that the operating frequency can be reduced significantly from the original design point D_0 in order to obtain real-time 30 fps decoding of QCIF and CIF resolutions (D_{36} and D_{37}), which corresponds respectively to 5MHz and 22MHz. It is also interesting to see that by applying this combination of refactoring and buffer minimization techniques, throughput can be improved by 6.6x compared to the original design, with occupied slice increment of only by 49%. The design point D_{35} shows the highest throughput with 1519 QCIF fps at a maximum frequency of 96 MHz.

The target implementation device for our design case study is on Xilinx Virtex-5 XC5VLX110T that contains 69120 number of slice registers, 69120 number of slice LUTs, and 148 number of 36kb BRAM. Since the maximum number of slice and BRAM required respectively are only 33532 and 109, all design points could easily fit in the FPGA. In terms of throughput, the design spans from 30 QCIF fps to 1519 QCIF fps. In other words, the design could perform well into the 30 HD720p fps, which is equivalent to a throughput of 1092 QCIF fps. In order to find the best point for various throughput requirement, the Pareto analysis is used to find the relevant Pareto sets. Figures 7.4, 7.5, and 7.6 respectively show 2-dimensional graph plots of throughput versus slice register, slice LUT, and block RAM with the corresponding Pareto

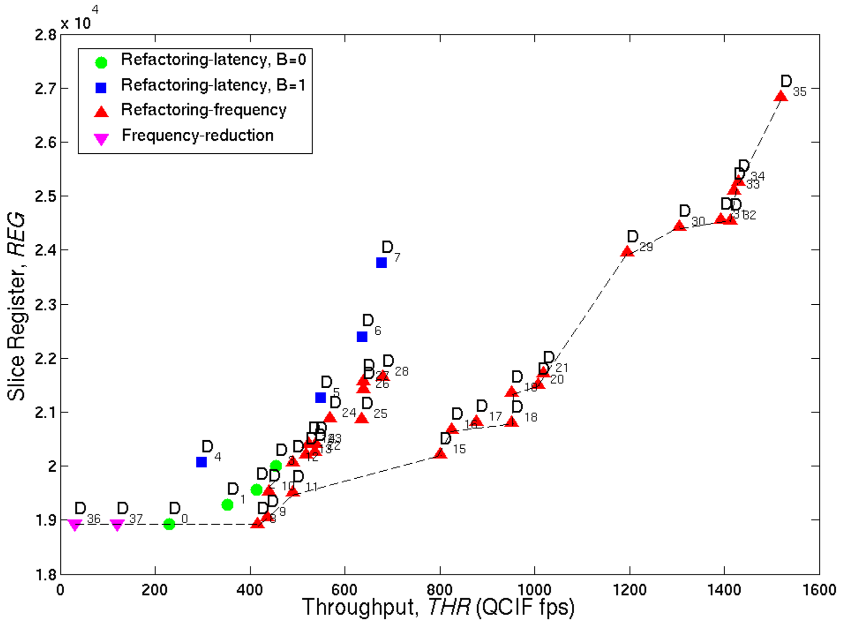


Figure 7.4: 2D plot of throughput versus slice register for MPEG-4 SP decoder case study. Dashed lines represent the Pareto frontier with the set $\{D_0, D_8, D_9, D_{11}, D_{15}, D_{16}, D_{18}, D_{19}, D_{20}, D_{21}, D_{29}, D_{30}, D_{32}, D_{33}, D_{34}, D_{35}, D_{36}, D_{37}\}$.

set and frontier. For all resource criteria, the following design points result in the lowest cost in terms of resource for a given throughput requirement: D_{36} for 30 QCIF fps, D_{37} for 120 QCIF/30 CIF fps, D_{11} for 480 QCIF/30 4CIF fps, and D_{29} for 1092 QCIF/30 HD720p fps.

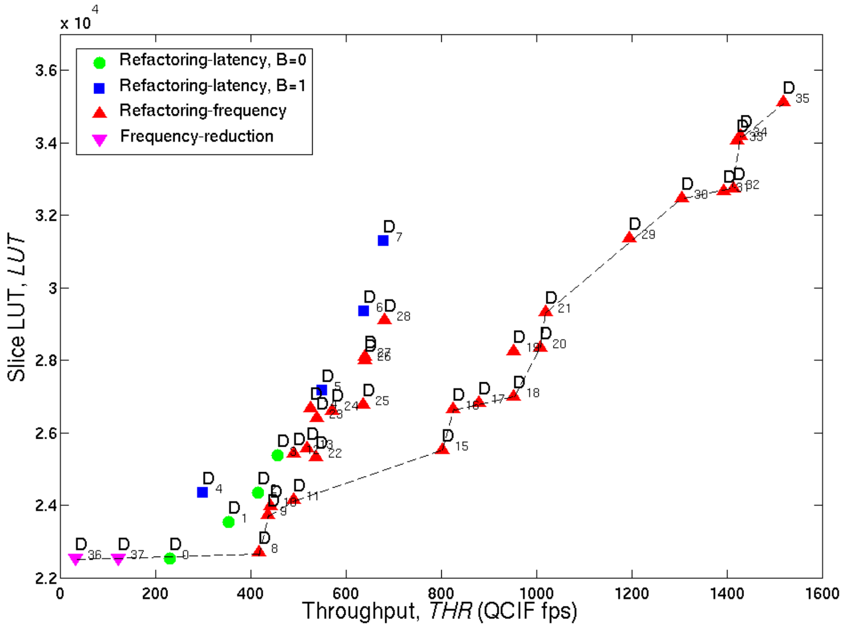


Figure 7.5: 2D plot of throughput versus slice LUT for MPEG-4 SP decoder case study. Dashed lines represent the Pareto frontier with the set $\{D_0, D_8, D_9, D_{10}, D_{11}, D_{15}, D_{16}, D_{17}, D_{18}, D_{19}, D_{20}, D_{21}, D_{22}, D_{23}, D_{29}, D_{30}, D_{31}, D_{32}, D_{33}, D_{34}, D_{35}, D_{36}, D_{37}\}$.

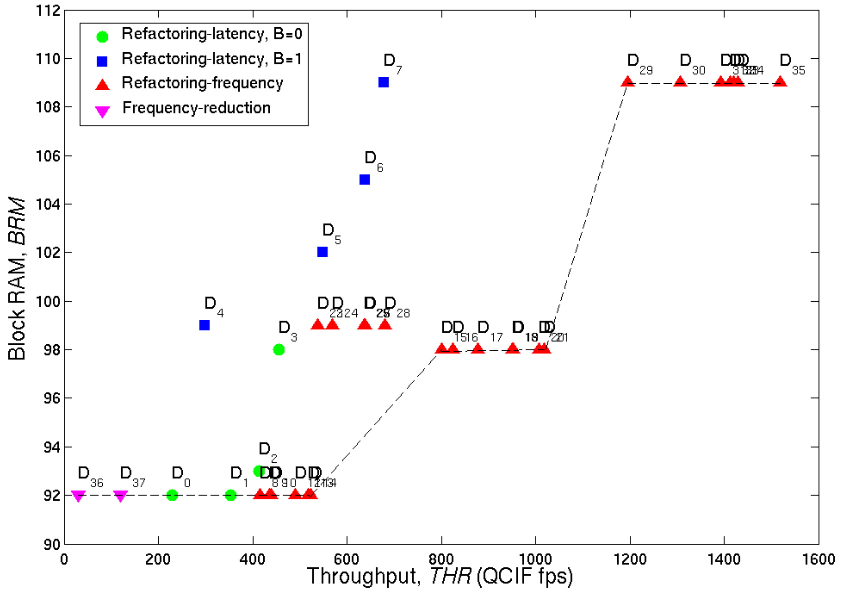


Figure 7.6: 2D plot of throughput versus block RAM for MPEG-4 SP decoder case study. Dashed lines represent the Pareto frontier with the set $\{D_0, D_1, D_8 D_9, D_{10}, D_{11}, D_{12}, D_{13}, D_{14}, D_{15}, D_{16}, D_{17}, D_{18}, D_{19}, D_{20}, D_{21}, D_{29}, D_{30}, D_{31}, D_{32}, D_{33}, D_{34}, D_{35}, D_{36}, D_{37}\}$.

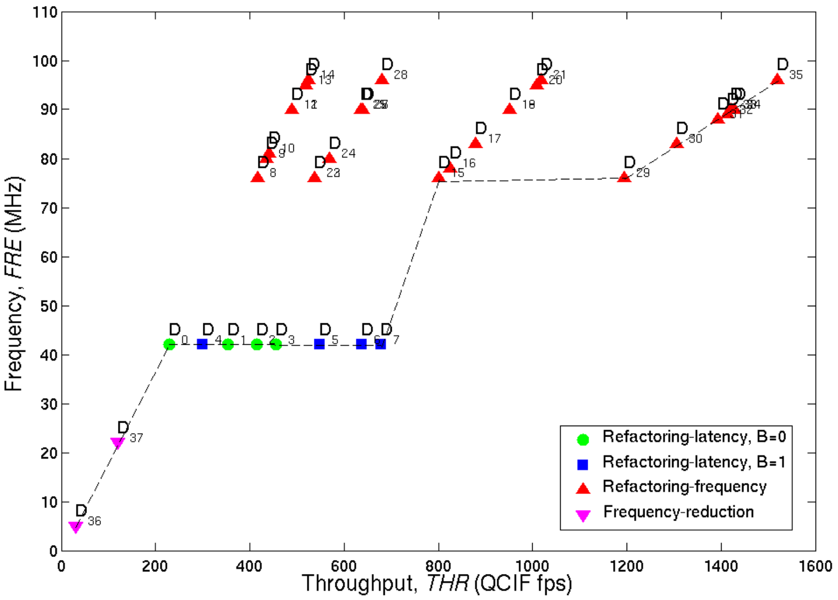


Figure 7.7: 2D plot of throughput versus frequency for MPEG-4 SP decoder case study. Dashed lines represent the Pareto frontier with the set $\{D_0, D_1, D_2 D_3, D_4, D_5, D_6, D_7, D_{15}, D_{29}, D_{30}, D_{31}, D_{32}, D_{33}, D_{34}, D_{35}, D_{36}, D_{37}\}$.

Another important design objective is to reduce power for a given throughput requirement, which can be achieved by selecting design points with the lower operating frequency. The 2D plot of throughput versus frequency and the corresponding Pareto frontier and set is given in Figure 7.7. The best design point for throughput requirement of 480 QCIF/30 4CIF fps is now D_5 with operating frequency of 42 MHz, as opposed to D_{11} with operating frequency of 90 MHz. The occupied slice however, is higher for point D_5 with 6792, compared to only 6060 for point D_{11}.

7.5 Case study-2: MPEG-4 AVC/H.264 decoder

Starting from the original description of the decoder in CAL (Section 6.4), it is synthesized to both C and HDL respectively for implementation on Xilinx Virtex-5 FPGA and general purpose computer with Intel i7 2.3 GHz CPU. The performance is summarized in Table 7.6 for various components of the decoder. For hardware implementation, the full decoder is found not only to be too large to fit in our target device, but also results in only a modest performance in terms of throughput. The required number of slice LUT is around 800k, whereas the largest Virtex-5 family FPGA contains only around 200k slice LUT. In terms of performance, the maximum throughput (THR) achieved is only 43 QCIF fps with a maximum frequency (FRE)

Table 7.6: Performance summary of the original design for the following MPEG-4 AVC/H.264 decoder components: Full decoder, Decoder_Y, Decoder_U/V, and Parser. The design is implemented on a Xilinx Virtex-5 FPGA (XC5VLX110T), and a general purpose computer with Intel i7 2.3GHz CPU. The buffer interconnections are assigned using the HEO technique. The units for LAT, FRE, and THR respectively are clock cycles per QCIF frame, MHz, and QCIF frames per second.

Platform	Component	LAT	FRE	THR	REG	LUT	BRM	SLI
FPGA	Full decoder	660124	29	43	83804	809866	189	202467
	Decoder_Y	650710	56	86	24809	46288	77	11572
	Decoder_U/V	95100	79	831	12389	24471	78	6118
	Parser	134323	29	214	20301	50745	0	12686
CPU	Full decoder	-	2300	59	-	-	-	-
	Parser	-	2300	2327	-	-	-	-

of 29 MHz. This design is not feasible to be implemented on FPGA due to the very high complexity and low performance obtained. Therefore, we propose to implement the decoder on a heterogeneous hardware/software platform, where the parts of the decoder with high potential for parallelism (*Decoder_Y* and *Decoder_U/V*) are implemented on hardware/FPGA. As shown in the Table, the three decoding components (*Decoder_Y*, *Decoder_U* and *Decoder_V*) together results in reasonable amount of resource with cumulative slice reg (REG) and slice LUT (LUT) respectively of 91416 and 44391.

The serial part of the decoder has been identified as the bitstream parser, which is more suited to be implemented on software due to the higher operating frequency offered by general purpose CPUs compared to FPGAs. As shown in Table 7.6, the parser performs very well on a CPU, with throughput of 2327 QCIF fps when measured for the output rate of the y-branch residual output. The full decoder on the other hand, only achieves a maximum throughput of 59 QCIF fps. This is due to the parallel architecture of the main decoding parts (*Decoder_Y* and *Decoder_U/V*) that could not be handled well in the current software architecture. The top-level block diagram of the proposed hardware/software implementation of the decoder is given in Figure 7.8. The parser will be implemented using a general purpose CPU, while the three decoding components and the merger (with function to simply merge the decoded bitstream) will be implemented on FPGA. The following presents results and exploration of the decoding components for both chroma (U and V) and luma (Y) samples.

7.5.1 Decoder_U/V

As shown in Table 7.6, the original design of the *Decoder_U/V* component on FPGA results in a relatively high throughput at 831 QCIF fps. If the target is to achieve real-time 30 fps

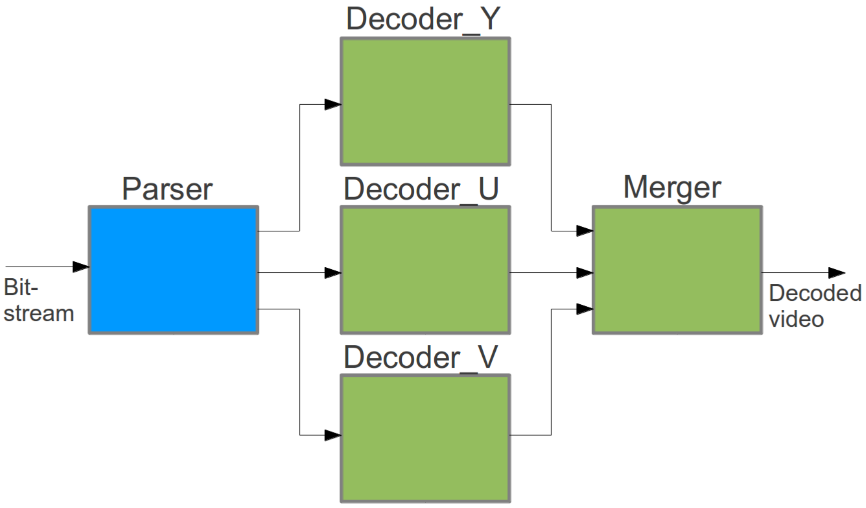


Figure 7.8: Simplified top-level view of the MPEG-4 AVC/H.264 decoder. The Parser is to be implemented on a general purpose CPU, while the main decoding components and the merger on FPGA.

for HD720p resolution (at 1092 QCIF fps), then only around 30% improvement is required. The list of all design points with the corresponding design parameters and criteria is given in Table 7.7. The strategy is to first perform memory refactoring/optimization from the original design point (D_0) until the desired throughput is achieved. The data-packing and redundancy-elimination techniques are applied on the actor `picture_buffer` (parameter $A = 1$) to obtain design point D_1, and then the redundancy-elimination technique is applied on the actor `deblocking_filter` (parameter $A = 2$) to obtain design point D_2. This point is then taken for buffer minimization to reduce the amount of resource. For this, the *TEM* (design point D_3) and *TEO* (design points D_4, D_5, and D_6) techniques are used. Design points with different operating frequency are also explored for lower throughput requirements of real-time 4CIF, CIF, and QCIF resolutions (design points D_7, D_8, and D_9 respectively). This corresponds to design parameters of $F = 1$, $F = 2$, and $F = 3$ respectively. The 3-dimensional graph plot of the design points is given in Figure 7.9.

The lower and upper bounds of each design criteria is given in Table 7.8. With the maximum occupied slice (*SLI*) of 6118, all design points could easily fit the target device Xilinx Virtex-5 device. In terms of throughput (THR), the value spans from 32 to 1092 QCIF fps. Based on the 3-dimensional graph in Figure 7.9, the following design points represent the best point for a given throughput requirement: D_9 for 30 QCIF fps, D_8 for 120 QCIF/30 CIF fps, D_7 for 480 QCIF/30 4CIF fps, and D_6 for 1092 QCIF/30 HD720p fps. These points also represent the best points in terms of frequency, as can be seen from the graph in Figure 7.10.

Table 7.7: Design points and the corresponding parameters and criteria for the *Decoder_U/V* component of the MPEG-4 AVC/H.264 decoder case study. The units for LAT, FRE, and THR respectively are clock cycles per QCIF frame, MHz, and QCIF frames per second.

Design point	**Parameters**			**Criteria**							
	A	F	B	LAT	FRE	THR	REG	LUT	BRM	SLI	DSP
D_0	0	0	1	95100	79	831	12491	24384	79	6096	62
D_1	1	0	1	86814	79	910	12278	24248	79	6062	62
D_2	2	0	1	72365	79	1092	12389	24471	80	6118	62
D_3	2	0	2	93785	79	842	11404	22694	49	5674	62
D_4	2	0	30	86910	79	909	11425	22767	49	5692	62
D_5	2	0	31	79384	79	995	11428	22771	49	5693	62
D_6	2	0	32	72410	79	1091	11431	22776	50	5694	62
D_7	2	1	2	93785	45	480	11404	22694	49	5674	62
D_8	2	2	2	93785	12	128	11404	22694	49	5674	62
D_9	2	3	2	93785	3	32	11404	22694	49	5674	62

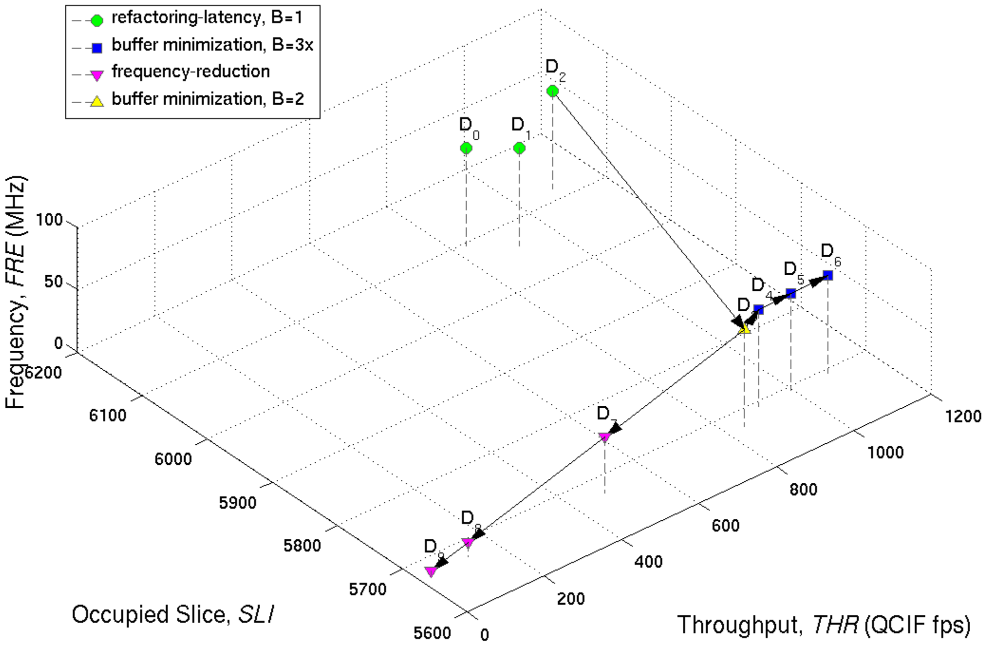


Figure 7.9: 3D plot of frequency, occupied slice, and throughput for the *Decoder_U/V* component of the MPEG-4 AVC/H.264 decoder case study.

Table 7.8: Nadir and ideal objective vectors for each design criteria for the *Decoder_U/V* component of the MPEG-4 AVC/H.264 decoder case study. The units for LAT, FRE, and THR respectively are clock cycles per QCIF frame, MHz, and QCIF frames per second.

	Design Criteria							
	LAT	FRE	THR	REG	LUT	BRM	SLI	DSP
z^{nad}	72365	3	32	11404	22694	49	5674	62
z^{ideal}	95100	79	1092	12491	24471	80	6118	62

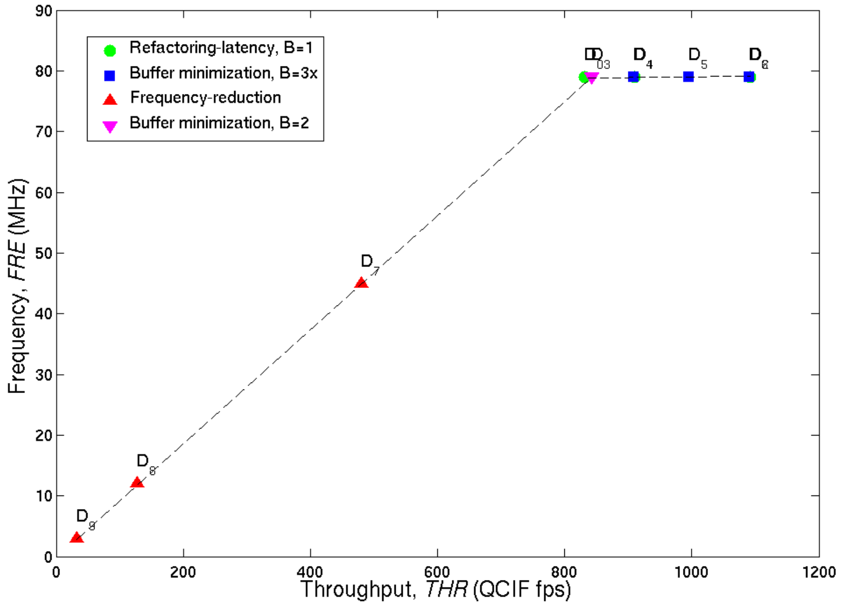


Figure 7.10: 2D plot of throughput versus frequency for the *Decoder_U/V* component of the MPEG-4 AVC/H.264 decoder case study. Dashed lines represent the Pareto frontier with all design points D_0 to D_9 in the set Pareto set.

7.5.2 Decoder_Y

As shown in Table 7.6, the throughput obtained for the *Decoder_Y* component of the MPEG-4 AVC/H.264 decoder is 86 QCIF fps. The objective here is to apply some of the refactoring and buffer minimization and optimization techniques such that throughput can be improved significantly, while at the same time minimizing the resources. The approach taken is as follows. Starting from the original design, refactoring for latency is applied for 11 iterations (design points D_1 to D_{11}). Then, four design points are selected for frequency refactoring, which are D_1, D_5, D_9, and D_{11}. This is performed for 5 iterations each, to obtain respectively the design points D_{12} to D_{16}, D_{17} to D_{21}, and D_{22} to D_{26}. In order to minimize resource, the TEM buffer minimization technique is applied for design points D_{26} and D_{16} respectively to obtain the design points D_{32} and D_{41}. The TEO buffer optimization technique is also applied on the design point D_{32} to obtain design points D_{33} to D_{40}. For real-time CIF and QCIF resolution requirements, the operating frequency is reduced for the design point D_{41} to obtain design points D_{42} and D_{43}. The corresponding parameters and criteria for each design point is given in Table 7.9, with specific actor refactoring for latency, frequency, and buffer size optimization respectively given in Tables 7.10, 7.11, and 7.12. The lower and upper bounds for each criteria is given in Table 7.13.

Table 7.9: Design points and the corresponding parameters and criteria for the *Decoder_Y* component of the MPEG-4 AVC/H.264 decoder case study. The units for LAT, FRE, and THR respectively are clock cycles per QCIF frame, MHz, and QCIF frames per second.

Design	**Parameters**			**Criteria**							
point	A	F	B	LAT	FRE	THR	REG	LUT	BRM	SLI	DSP
D_0	0	0	1	650710	56	86	24809	46288	77	11572	62
D_1	1	0	1	592146	56	95	24512	47356	77	11839	62
D_2	2	0	1	344877	56	163	29141	58924	102	14731	62
D_3	3	0	1	282924	56	198	32126	69594	111	17399	62
D_4	4	0	1	238302	56	235	29874	102103	119	25526	62
D_5	5	0	1	177914	56	315	28451	97357	119	24339	62
D_6	6	0	1	159702	56	351	28852	99065	119	24766	62
D_7	7	0	1	143529	56	391	35143	99601	119	24900	62
D_8	8	0	1	134496	56	417	33065	115814	120	28954	62
D_9	9	0	1	124470	56	451	35348	141665	121	35416	62
D_{10}	10	0	1	117640	56	477	36455	169033	122	42258	62
D_{11}	11	0	1	104396	56	537	36989	220185	124	55046	62
D_{12}	1	1	1	592146	64	108	24725	47750	77	11938	62
D_{13}	1	2	1	592146	80	135	25100	47993	77	11998	62
D_{14}	1	3	1	592146	96	162	25241	48153	77	12038	62
D_{15}	1	4	1	592146	109	184	25331	48630	77	12158	62
D_{16}	1	5	1	592146	114	193	25448	48997	77	12249	62

D_{17}	5	1	1	177914	64	361	28674	97784	119	24446	62
D_{18}	5	2	1	177914	80	449	29987	97960	119	24490	62
D_{19}	5	3	1	177914	96	538	30104	97158	119	24290	62
D_{20}	5	4	1	177914	109	614	30250	97610	119	24403	62
D_{21}	5	5	1	177914	114	641	30487	97986	119	24497	62
D_{22}	9	1	1	124470	64	516	35484	141992	121	35498	62
D_{23}	9	2	1	124470	80	642	35918	142275	121	35569	62
D_{24}	9	3	1	124470	96	769	36085	142441	121	35610	62
D_{25}	9	4	1	124470	109	877	36221	142951	121	35738	62
D_{26}	9	5	1	124470	114	917	36324	143261	121	35815	62
D_{27}	11	1	1	104396	64	615	37150	220512	124	55128	62
D_{28}	11	2	1	104396	80	765	37584	220795	124	55199	62
D_{29}	11	3	1	104396	96	917	37751	220961	124	55240	62
D_{30}	11	4	1	104396	109	1046	37887	221471	124	55368	62
D_{31}	11	5	1	104396	114	1093	37990	221781	124	55445	62
D_{32}	9	5	2	219567	114	520	25984	61839	113	15460	62
D_{33}	9	5	30	213387	114	535	25985	61872	113	15468	62
D_{34}	9	5	31	203859	114	560	25990	61938	113	15485	62
D_{35}	9	5	32	197250	114	578	26998	61989	113	15497	62
D_{36}	9	5	33	183525	114	622	26110	62190	113	15548	62
D_{37}	9	5	34	168228	114	678	26120	62173	113	15543	62
D_{38}	9	5	35	164853	114	692	26128	62220	113	15555	62
D_{39}	9	5	36	163047	114	700	26139	62280	113	15570	62
D_{40}	9	5	37	143037	114	798	26228	62534	119	15634	62
D_{41}	1	5	2	592137	114	193	24427	27604	42	6901	62
D_{42}	1	6	2	592137	71	120	24427	27604	42	6901	62
D_{43}	1	7	2	592137	18	30	24427	27604	42	6901	62

The 3-dimensional plot of frequency (FRE), occupied slice (SLI), and throughput (THR) for the decoder case study is given in Figure 7.11. Overall, the throughput range is from 30 to 1092 QCIF fps (36x difference), occupied slice range from 6901 to 55445 (8x difference), and frequency range from 18 MHz to 114 MHz (6x difference). With refactoring for latency (green/circle points), we can see that throughput is improved independently of operating frequency. This is in contrast to refactoring for frequency (red/downward triangle points) where throughput is improved by applying a higher operating frequency. The Buffer minimization technique is also very effective with up to 2.3x reduction in the occupied slice. The subsequent application of buffer optimization results in a further 53% increase in throughput. It is also interesting to see that for real-time QCIF and CIF resolution requirements, the occupied slice is the minimum at 6901 with operating frequency of 71 MHz and 18 MHz respectively.

The design points are now analyzed for implementation on the target Xilinx Virtex-5 FPGA.

Table 7.10: Specific refactoring for latency applied on the *Decoder_Y* component of the MPEG-4 AVC/H.264 decoder case study. The unit for latency is clock cycles per macroblock. The techniques are applied cumulatively from D_0 to D_{11}.

Design point	**Technique**	**Actor**	**Latency (C.C./MB)**
D_0	*original*	-	6573
D_1	Data-packing Redundancy-elimination	picture_buffer_y	5981
D_2	Data-parallelism(2x)	half_quarter_interpolation	3483
D_3	Data-parallelism(3x)	half_quarter_interpolation	2857
D_4	Data-parallelism(4x)	interp_reorder_y add_pix_sat demux_parser_info_y	2407
D_5	Redundancy-elimination	deblocking_filter_y	1797
D_6	Data-parallelism(4x)	picture_buffer_y deblocking_filter_y	1613
D_7	Data-packing	interp_reorder_y	1449
D_8	Task-parallelism(2x)	half_quarter_interpolation	1358
D_9	Task-parallelism(3x)	half_quarter_interpolation	1257
D_{10}	Task-parallelism(4x)	half_quarter_interpolation	1195
D_{11}	Task-parallelism(5x)	half_quarter_interpolation	1054

Table 7.11: Specific refactoring for frequency applied on the *Decoder_Y* component of the MPEG-4 AVC/H.264 decoder case study. The techniques are applied cumulatively from 1 to 5. $F = 6$ and $F = 7$ in Table 7.9 refers to the frequency-reduction technique

Design points	**Actor(s)**	**Action(s)**	**# of pipeline stages**	$\overline{f_{max}}$ **(MHz)**
$D_1/D_5/D_9, D_{11}$	*original*	-	-	56
$D_{12}/D_{17}/D_{22}/D_{27}$	half_quarter_interpolation	getPixVal_done	2	64
$D_{13}/D_{18}/D_{23}/D_{28}$	half_quarter_interpolation picture_buffer_y idct_scaler	getPixVal_done writeData_done read_coeff	3 3 4	80
$D_{14}/D_{19}/D_{24}/D_{29}$	picture_buffer_y intrapred_luma16x16	writeData_done write_mode	4 2	96
$D_{15}/D_{20}/D_{25}/D_{30}$	idct_scaler intrapred_luma16x16	read_coeff write_mode	5 4	109
$D_{16}/D_{21}/D_{26}/D_{31}$	idct_scaler intrapred_luma16x16	read_coeff write_mode	8 7	114

Table 7.12: Specific buffer size optimization technique for each design point of the *Decoder_Y* component of the MPEG-4 AVC/H.264 decoder case study.

Design points	**Technique**
D_0 - D_{31}	Hardware Execution Optimization (HEO)
D_{32}, D_{41} - D_{43}	Trace Execution Minimization (TEM)
D_{33} - D_{40}	Trace Execution Optimization (TEO)

Table 7.13: Nadir and ideal objective vectors for each design criteria for the *Decoder_Y* component of the MPEG-4 AVC/H.264 decoder case study. The units for LAT, FRE, and THR respectively are clock cycles per QCIF frame, MHz, and QCIF frames per second.

	Design Criteria							
	LAT	FRE	THR	REG	LUT	BRM	SLI	DSP
z^{nad}	104396	18	30	24427	27604	42	6901	62
z^{ideal}	650710	114	1093	37790	221781	124	55445	62

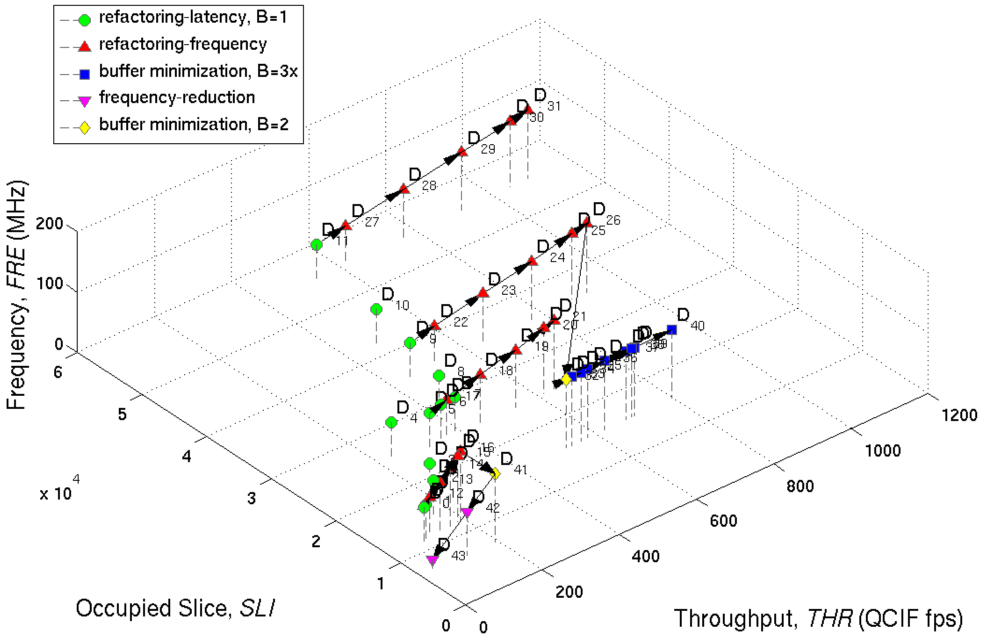


Figure 7.11: 3D plot of frequency, occupied slice, and throughput for the *Decoder_Y* component of the MPEG-4 AVC/H.264 decoder case study.

Table 7.14 presents attributes of four of the largest available devices for this FPGA family. The original target device is on the XCV5VLX110T, but as shown on the maximum number of slice, some design points do not fit this device. Therefore, larger (and more expensive) device is required for these design points. In order to find the design point with the highest throughput for each of the four devices, the Pareto analysis is performed for throughput versus slice LUT as shown in the graph in Figure 7.12. Note that the graphs of throughput versus slice registers and throughput versus block RAM are not analyzed for Pareto set since all design points are found to be bounded by these criteria. For the devices XC5VLX110T, XC5VLX155T, and XC5VLX220T, it is found that the design point D_{40} shows the highest throughput with 798 QCIF fps. Using the largest available device in XC5VLX330T, it is possible to achieve throughput of up to 917 QCIF fps with the design point D_{26}. However, it should be noted that the values for the design criteria (Table 7.9) is obtained with synthesis to the XC5VLX110T device. When using larger devices with higher number of block RAMs and DSP48Es, it is possible that the design point D_{26} with throughput of 917 QCIF fps and D_{31} with throughput of 1092 QCIF fps respectively would fit in the XC5VLX220T and XC5VLX330T devices.

Table 7.14: Attributes of four of the largest devices in the Virtex-5 FPGA family.

Device	**# of slice**	**# of slice Reg**	**# of slice LUT**	**# of 36kb BRAM**	**# of DSP48E**
XC5VLX110T	17280	69120	69120	148	64
XC5VLX155T	24320	97280	97280	212	128
XC5VLX220T	34560	138240	138240	212	128
XC5VLX330T	51840	207360	207360	324	192

We now aim to find a design point with two different objectives: low resource, typically an objective for low cost systems; and low frequency, an objective for low power systems. For low resource requirement, the graph in Figure 7.12 is analyzed, with the following results: design point D_{43} for 30 QCIF fps with $LUT = 27604$, D_{42} for 120 QCIF/30 CIF fps with $LUT = 27604$, D_{32} for 480 QCIF/30 4CIF fps with $LUT = 61839$, and D_{31} for 1092 QCIF/30 HD720p fps with $LUT = 221781$. For low frequency requirement, the throughput versus frequency graph in Figure 7.13 is analyzed. It is found that the same design points are obtained for real-time QCIF, CIF, and HD720p resolution requirements. For real-time 4CIF resolution however, the design point D_{11} results in the lowest frequency of 56 MHz, compared to the design point D_{32} with 114 MHz. The resource for the design point D_{11} is much higher at $LUT = 220185$.

7.5.3 Combining Decoder_Y and Decoder_U/V

Since the three main decoding components are implemented in an FPGA, the resource can be estimated for a given design point in the combined *Decoder_Y* and *Decoder_U/V* exploration

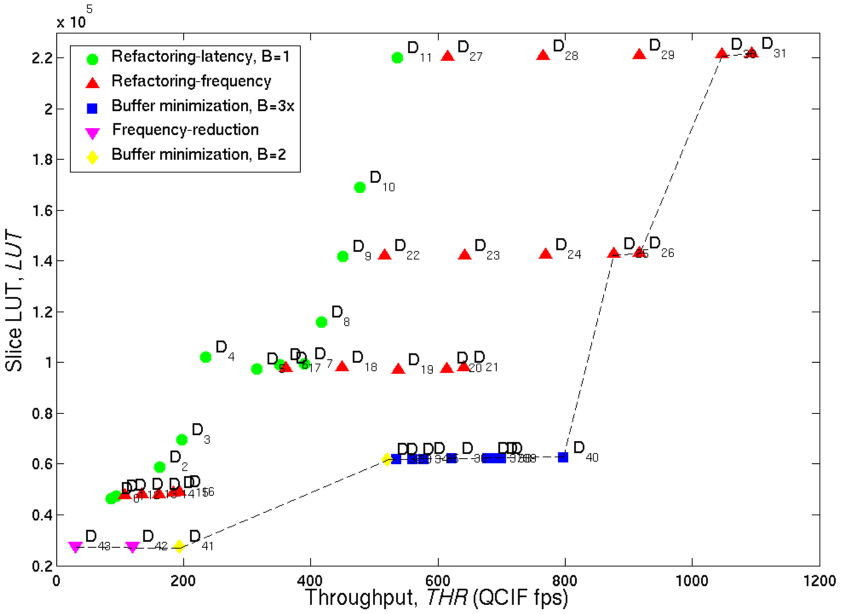


Figure 7.12: 2D plot of throughput versus slice LUT for the *Decoder_Y* component of the MPEG-4 AVC/H.264 decoder case study. Dashed lines represent the Pareto frontier with the set $\{D_{25}, D_{26}, D_{30} D_{31}, D_{32}, D_{33}, D_{34}, D_{35}, D_{37}, D_{38}, D_{39}, D_{40}, D_{41}, D_{42}, D_{43}\}$.

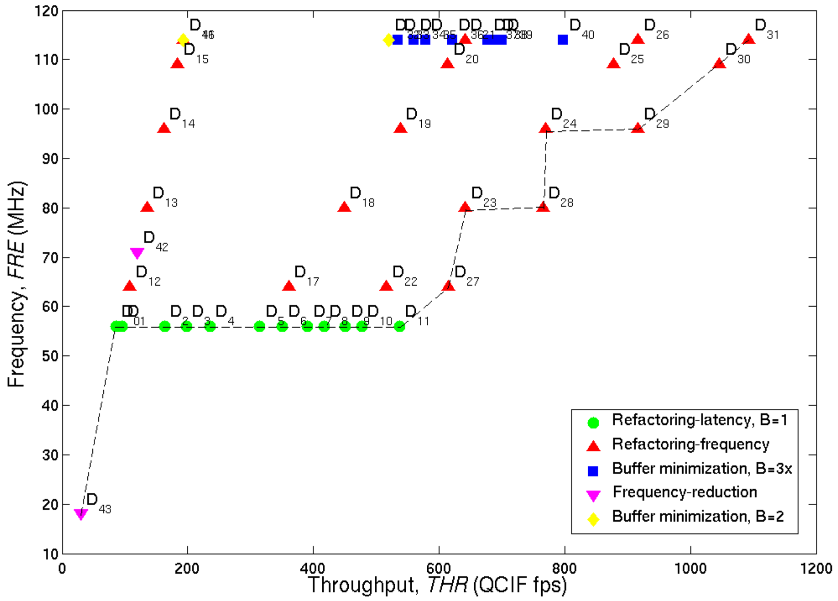


Figure 7.13: 2D plot of throughput versus frequency for the *Decoder_Y* component of the MPEG-4 AVC/H.264 decoder case study. Dashed lines represent the Pareto frontier with the set $\{D_0, D_1, D_2, D_3, D_4 D_5, D_6, D_7, D_8, D_9, D_{10}, D_{11}, D_{23}, D_{24}, D_{27}, D_{28}, D_{29}, D_{30}, D_{31}, D_{43}\}$.

space by adding together the relevant criteria values from the *Decoder_Y* and *Decoder_U/V* exploration spaces. For example, the required slice for all three of the decoding components for a given design point p in the combined exploration space is given by

$$SLI_p^{yuv} = 2 \times SLI_q^{uv} + SLI_p^{y} \tag{7.4}$$

since there are two chroma and one luma components. Each design point p in the *Decoder_Y* space can be mapped to a similar point p in the combined space. The point q in the *Decoder_U/V* space for a given point p is selected from a Pareto set for the *Decoder_U/V* space at the given criteria value of the point p.

In terms of frequency, $FRE_p^{yuv} = max(FRE_q^{uv}, FRE_p^{y})$, i.e. the worst case frequency requirement. In terms of latency, $LAT_p^{yuv} = LAT_p^{y}$ since the critical path (i.e. longest path) is on the y-branch of the decoder. Table 7.15 shows the values of several design criteria for four different throughput requirement in the combined exploration space.

Table 7.15: Values of several design criteria for various throughput requirements in the combined *Decoder_Y* and *Decoder_U/V* exploration space.

Throughput (QCIF fps)	**Design Criteria**		
	SLI	LAT	FRE
30	18249	592137	18
120	18249	592137	71
480	66394[a]/26808[b]	104396[a]/219567[b]	56[a]/114[b]
1092	66833	104396	114

[a] low frequency requirement
[b] low resource requirement

7.6 Comparison with related works

Table 7.16 presents comparison of the MPEG-4 SP design in terms of implementation platform and specification language(s), and criteria for frequency, throughput, and resource. Overall, our design outperforms all other designs in all criteria. For example, compared to the IP core from Xilinx [7], a throughput of around 1500 QCIF fps at 100 MHz is obtained in our case, while the IP core was designed for only up to 480 QCIF fps at around the same frequency. In terms of resource, our design also utilizes roughly 10% less slice. At low throughput requirement (15 QCIF fps) as compared to [32] where they have used a software controller and a hardware core, we are able to execute the decoder at 6x less clock speed at 2 MHz compared to their implementation using classical methodology at 12 MHz. In terms of resource, our design

Table 7.16: Comparison of the present work with similar works in literature for MPEG-4 SP decoder implementation. The present work is shown for two design points in the case of minimum and maximum throughput design.

	Platform	**Specification language**	**Frequency (MHz)**	**Throughput (QCIF fps)**	**Resource (slice/gates)**
This work (THR_{min})	Virtex-5	CAL	5	30	5634
This work (THR_{max})	Virtex-5	CAL	96	1519	8383
[7]	Virtex-4	RTL	100	480	6230
[59]	ASIC	RTL	85	480	155000
[32]	ARC/Virtex-1k	C/RTL	12	15	8074
[39]	ARM+FPGA	C/systemC	-	15	-

consumes around 43% less slice.

For the MPEG-4 AVC/H.264 decoder case study, the bit-stream parser is implemented on a general purpose CPU (Intel i7 2.3GHz), while the main decoding components (*Decoder_U/V* and *Decoder_Y*) and the merger on FPGA. Table 7.17 presents the comparison. As compared to the IP core from *Fastvdo* [57], this work results in similar throughput at similar frequency at 1092 QCIF fps and around 115 MHz. In terms of resource however, our design utilizes 50% more slice, although the bit-stream parser is not included in our resource utilization. Another IP core is provided by *coreEL* [58] where their design have shown to perform up to 2045 QCIF fps on an ASIC platform (advertised at Full HD1080p). For low throughput requirement, our design is found to be superior to the SystemC implementation given in [115] in terms of frequency, where an operating frequency of only 18MHz is required compared to 110MHz (factor of 6) at the same 30 QCIF fps requirement. Their design however, utilizes 27% less slice. Finally, when compared to a software/hardware implementation for low throughput requirement [120], our implementation results in a lower operating frequency at roughly 5 MHz compared to 10 MHz for their design (factor of 2) at the same 7.4 QCIF fps. The resource utilization however, could not be compared directly since their implementation is on ASIC with only the inverse quantization, inverse transform, and the motion compensation units on hardware.

The final important point to note in relation to similar works in literature is design productivity. Since CAL programs are specified at high-level, any optimizations on the program are also being applied at this architectural level, which generally results in simpler and more effective optimizations (and sometimes higher resource utilization). Moreover, alternative design points could also be found relatively quickly. In contrast to this work, all the similar works given in Table 7.16 and Table 7.17 using classical and other design methodologies present only

Table 7.17: Comparison of the present work with similar works in literature for MPEG-4 AVC/H.264 decoder implementation. The present work is shown for two design points in the case of maximum and minimum throughput design.

	Platform	**Specification language**	**Frequency (MHz)**	**Throughput (QCIF fps)**	**Resource (slice/gates)**
This work (THR_{min})	i7+Virtex-5	CAL	2300/18	30	18249
This work (THR_{max})	i7+Virtex-5	CAL	2300/114	1092	66833
[57]	Virtex-4	RTL	115	1092	44544
[58]	ASIC/FPGA	RTL	-	2045	-
[115]	Virtex-4	SystemC	110	30	23257
[120]	ARM+ASIC	C/RTL	140/10	7.4	28826

a single, or at most, two design points, and do not provide any effective space exploration.

7.7 Summary

In this chapter, we have presented a systematic methodology to explore and evaluate various design alternatives that are obtained using the techniques given in previous chapters. First, background and related works on design space exploration were provided in the general sense and in the specific domain of embedded systems and signal processing. The design points were first formally defined in the exploration space, together with the metrics that is used for evaluation. These definitions and metrics became the basis to construct an automatic tool to analyze the design points for various objectives and requirements. Following this, the design space exploration of two MPEG-4 decoders were presented: the MPEG-4 SP and the MPEG-4 AVC/H.264. The MPEG-4 SP decoder was implemented fully on hardware, while the MPEG-4 AVC/H.264 decoder on a heterogeneous software/hardware platforms. For each of these design case studies, design space exploration and evaluation have been performed. The results obtained were compared to similar works in literature, in terms of implementation platform and languages, operating frequency, performance, and resource. The results obtained are very promising, where it was shown that some design alternatives are comparable or superior to other similar works in several design criteria.

8 Conclusion

In this final concluding chapter, an overall summary of what has been contributed is first presented, followed by the impact of this work on the wider research community. In the last section, some possible future work and directions are given based on the outcome of this thesis.

8.1 Summary

This thesis presents an original research work that aims to optimize and explore complex dataflow programs for implementation on hardware and heterogeneous platforms. To this end, we first described the classical design methodology and its limitations for implementing complex DSP applications. Several high-level languages and models were also introduced as alternatives to the classical model, including C and SystemC models, synchronous languages, pre-configured blocks and templates, and dataflow programming models. The latter have been shown to be efficient and provides a natural abstraction for data-intensive DSP applications, but the currently available tools and models are found to be lacking and insufficient for a complete design process from specification to implementation. In this thesis, the following contributions are made:

- **Validation of a new systems design methodology with CAL**. The complete SW/HW co-design flow from specification to implementation were validated in this thesis for very complex applications. However, what is lacking from the flow is the optimization tools and techniques that can be applied on the dataflow program, especially for hardware implementation target. This is a very important aspect in the design process, where a design most likely has to be iterated several times in order to reach the desired performance. Three new strategies have been identified to achieve this objective, as described next.

- **A refactoring technique to achieve significant reduction in system latency**. This includes data and task parallelism, and memory optimizations by packing data tokens and eliminating redundant access. The techniques were first presented in the general case for the DPN MoC in dataflow programs, and then applied on the critical actors of the MPEG-4 AVC/H.264 decoder case study. The performance gain shown in the experimental results (by up to 5x latency reduction) have shown to agree with the analytical latency reduction estimation. Due to the use of the DPN MoC, the techniques were only applied semi-automatically, but the approach is generic and can be applied on any CAL designs that require a reduction in system latency.

- **A refactoring technique to achieve a significant increase in maximum operating frequency**. The technique was designed to find a pipeline schedule for a given action, such that the total pipeline register width is minimized, for a given throughput constraint. The technique is a fully automated approach where a single actor implementation is transformed into a multi-actor implementation. One of the limitations of the automated program was that it only accepts an acyclic single-action actor. For this, a methodology has been presented to reduce a complex action in any arbitrary actor into a representation that can be used by the automated tool. Experimental results have proven the efficacy of the automatic synthesis and optimization tool with almost 2x less total register width between the best and the worst case pipeline schedule. The efficacy of the technique has also been proven when applied on an at-large MPEG-4 decoder case study, where only a minor additional resource was required for several iterations of pipelining.

- **Buffer interconnection size minimization and optimization**. The techniques were designed for a generic DPN MoC using two different approaches: on the hardware execution level, and on the dataflow program level. For each approach, two techniques were implemented, one for finding the minimum or close to minimum buffer size, and the other for using larger buffer size for higher throughput requirements. All the four techniques were compared and contrasted. For the MPEG-4 AVC/H.264 *decoder-Y* component case study, it is found that using the dataflow program level approach results in the smallest total buffer size for deadlock-free execution (at low throughput), while using the hardware execution level approach results in the highest throughput (using larger total buffer size). Both approaches show that the total buffer size can be reduced by multi-fold compared to a direct constant buffer size assignment.

- **Methodology for design space exploration and evaluation**. The final contribution is a methodology to systematically and efficiently explore and evaluate design alternatives in the exploration space. The design points were obtained by appropriate combinations of the refactoring and optimization techniques, and then evaluated using multidimensional plot and Pareto analysis for various performance criteria. For MPEG-4 SP decoder hardware implementation case study, throughput has been shown to increase by about 6.6x compared to the original design, while consuming only 49% more resource. When this is compared to similar works in literature, the design have been shown to outperform all other designs in all criteria of throughput, frequency, and resource, both for low and high throughput requirement designs. For the software/hardware heterogeneous implementation of the MPEG-4 AVC/H.264 decoder case study, the `Parser` was partitioned to software, and the rest of the components to hardware. The hardware components have been optimized separately for the Y-branch and the U/V-branch decoders. Overall for the hardware components, throughput has been improved by about 12x with an additional occupied slice of 5x. Compared to similar works in literature, the performance is comparable to one commercial IP core, but lags behind to another with an ASIC implementation. For low throughput requirement comparison with SystemC, the design utilized around 30% more resource, but with an operating frequency of about 5x less.

8.2 Research impact

What do the results mean in the context of hardware and heterogeneous design and implementation of DSP systems? It has been proven before in numerous work that dataflow programming could significantly enhance designers productivity. This work takes a further step in proving that performance can also be improved seamlessly, with results shown that in some cases, overall performance could even rival those using other mainstream design methodologies. Furthermore, this work also proves that design alternatives can be found quickly and effectively by appropriate program optimizations at high-level, which is not always the case using classical low-level design methodology. **The techniques that have been presented here are generic and can be implemented on any DSP systems, and certainly would help in quickly designing and implementing high performance future video codecs**.

8.3 Future work and direction

Optimizations of large and complex systems are possibly a never-ending task. This is due to the NP nature of such problems where a global or perfect solution can never be found. So, despite some very promising results presented in this thesis, there are in fact many other directions that can be taken to improve the current state of research. The following are several optimization techniques that are not considered in the present work, but are interesting applications to the design case studies. The first is arithmetic operator resource sharing within an action for reducing the total resource; the second is the so-called cross-actor optimizations for resource and/or throughput improvements, and the third is the multi-clock domain optimization for reducing the average operating frequency. All of these techniques have been described in Section 2.5.2.

As for actual physical implementation, it should also be noted that the hardware implementation of the MPEG-4 SP decoder has been proven to work properly on a Xilinx Virtex-5 evaluation board. The heterogeneous implementation of the MPEG-4 AVC/H.264 decoder on the other hand, has not been verified on a physical platform, but merely proven by simulation. The challenge here is to provide the necessary interface for our chosen partitioning scheme with the bit-stream parser on software and the rest on hardware. The output of the parser contains more than 15 channels that need to be routed to the hardware platform. Since software implementation contains a single output channel, and hardware with multiple input channels, the signals have to first be *serialized* on the software side, and then *de-serialized* on the hardware side. One of the proposed serializer-deserializer architectures is given in [116]. Work is currently on-going to implement this interface architecture for heterogeneous implementation of MPEG-4 decoders, where the result would certainly be interesting for demo purposes.

Another issue with the current CAL specification of the MPEG-4 decoders is the use of dynamic actors. The specifications in fact utilize several dynamic actors, which, as explained in chapter 5, cannot be scheduled at compile time and the required buffer size for deadlock-free execution cannot be determined for all execution order. This creates the problem that correctness and deadlock-free execution cannot be guaranteed for all possible video input stream. However, by carefully analyzing the design of the video decoders, the dynamic actors can in fact be converted to a static one since the various possible input patterns are finite and can actually be controlled in the action, instead of checking the input patterns dynamically. It is certainly an interesting direction to re-design and re-implement the video decoders using only static actor types so that boundedness is guaranteed for all possible input data. It is also interesting to see how various performance criteria is compared when trading-off the flexibility of dynamic actors to the analyzability of static ones.

Bibliography

[1] Esterel Technologies. http://www.esterel-technologies.com/.

[2] Open RVC-CAL Compiler (Orcc). http://orcc.sourceforge.net/.

[3] *SPW User's Manual.* Cadence Design Systems, Foster City, California, USA.

[4] *COSSAP stream driven simulator user guide.* Synopsys, Mountain View, California, USA, 6.7 edition, 1994.

[5] *Catapult C Synthesis.* Mentor Graphics, Wilsonville, Oregon, USA, 2005.

[6] *AccelDSP Synthesis Tool User Guide Release 10.1.* Altera, San Jose, California, USA, 10.1 edition, March 2008.

[7] *MPEG-4 Simple Profile Decoder v1.3.* Xilinx, April 2008.

[8] *DSP Builder User Guide Software Version 9.1.* Altera, San Jose, California, USA, 9.1 edition, November 2009.

[9] Nios II C2H Compiler User Guide, November 2009.

[10] *Simulink 7 User Guide.* Mathworks, Natick, Massachusetts, USA, 7 edition, September 2010.

[11] Vivado design suite user guide: Programming and debugging, 2012.

[12] ISO/IEC 14496-10. *Information Technology – Coding of Audio-Visual Objects – Part 10: Advanced Video Coding.* International Standard, 2004.

[13] ISO/IEC 14496-2. *Information Technology – Coding of Audio-Visual Objects – Part 2: Visual.* International Standard, 2001.

[14] Yongjin Ahn, Keesung Han, Ganghee Lee, Hyunjik Song, Junhee Yoo, Kiyoung Choi, and Xingguang Feng. Socdal: System-on-chip design accelerator. *ACM Trans. Design Autom. Electr. Syst.*, 13(1), 2008.

[15] A. Aiken and A. Nicolau. Optimal loop parallelization. In *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1988.

[16] Cedell Alexander, Donna Reese, and James Harden. Near-critical path analysis of program activity graphs. In *Proceedings of the IEEE International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 308–317, 1994.

[17] H. Aman-Allah, K. Maarouf, E. Hanna, I. Amer, and M. Mattavelli. CAL dataflow components for an MPEG RVC AVC baseline encoder. *Journal of Signal Processing Systems*, 63(2):227–239, 2011.

[18] M. Auguin, L. Capella, F. Cuesta, and E. Gresset. CODEF: a system level design space exploration tool. In *Acoustics, Speech, and Signal Processing, 2001. Proceedings. (ICASSP '01). 2001 IEEE International Conference on*, volume 2, pages 1145–1148 vol.2, 2001.

[19] Denis Aulagnier, Ali Koudri, Stéphane Lecomte, Philippe Soulard, Joël Champeau, Jorgiano Vidal, Gilles Perrouin, and Pierre Leray. SoC/SoPC development using MDD and MARTE profile. In *Model Driven Engineering for Distributed Real-time Embedded Systems*. ISTE, 2009.

[20] L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, and M. Poncino. SystemC cosimulation and emulation of multiprocessor SOC designs. *Computer*, 36:53–59, 2003.

[21] A. Benveniste and P. Le Guernic. Hybrid dynamical systems theory and the Signal language. *IEEE Transactions on Automatic Control*, 35(5):535 –546, May 1990.

[22] E. Bezati, S. Casale-Brunet, M. Mattavelli, and J.W. Janneck. Synthesis and optimization of high-level stream programs. In *The 2013 Electronic System Level Synthesis Conference*, pages 1–6, 2013.

[23] E. Bezati, R. Thavot, G. Roquier, and M. Mattavelli. High-level dataflow design of signal processing systems for reconfigurable and multicore heterogeneous platforms. *Journal of Real-Time Image Processing*, pages 1–12, 2013.

[24] E. Bezati, H. Yviquel, M. Raulet, and M. Mattavelli. A unified hardware/software co-synthesis solution for signal processing systems. In *Design and Architectures for Signal and Image Processing (DASIP), 2011 Conference on*, pages 1–6, 2011.

[25] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 21(2):151–166, 1999.

[26] G. Bilsen, M. Engels, R. Lauwereins, and J.A. Peperstraete. Cyclo-static data flow. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 5, pages 3255–3258 vol.5, 1995.

[27] S. Casale-Brunet, E. Bezati, C. Alberti, M. Mattavelli, E. Amaldi, and J. W. Janneck. Multi-clock domain optimization for reconfigurable architectures in high-level dataflow applications. In *2013 Asilomar Conference on Signals, Systems and Computers*, pages 1870 –1874, November 2013.

[28] Rong Chen, Marco Sgroi, Grant Martin, Luciano Lavagno, Alberto S. Vincentelli, and Jan Rabaey. Embedded System Design Using UML and Platforms. In *Proceedings of Forum on Specification and Design Languages 2002 (FDL'02)*, September 2002.

[29] E. Cheong. *Actor-Oriented Programming for Wireless Sensor Networks*. PhD Thesis-University of California, Berkeley, September 2007.

[30] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer Berlin Heidelberg, 2002.

[31] J. Dennis. First version of a data-flow procedure language. *Proceedings of the colloque sure la programmation*, (19):362–376, April 1974.

[32] J. Dunlop, A. Simpson, S. Masud, M. Wylie, J. Cochrane, and R. Kinkead. Semiconductor IP core for ultra low power MPEG-4 video decode in system-on-silicon. In *Acoustics, Speech, and Signal Processing, 2003. Proceedings. (ICASSP '03). 2003 IEEE International Conference on*, volume 2, pages II–681–4 vol.2, 2003.

[33] J. Eker and J. Janneck. *CAL Language Report: Specification of the CAL Actor Language*. University of California-Berkeley, December 2003.

[34] Johan Eker, Jorn Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Sonia Sachs, Yuhong Xiong, and Stephen Neuendorffer. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[35] W. Elhamzi, R. Thavot, J. Dubois, J. Gorin, M. Atri, J. Miteran, and R. Tourki. An efficient hardware implementation of diamond search motion estimation using CAL dataflow language. In *Proceedings of the International Conference on Microelectronics, ICM*, 2011.

[36] Rolf Ernst, Jörg Henkel, and Thomas Benner. Hardware-software cosynthesis for micro-controllers. *IEEE Design & Test of Computers*, 10(4):64–75, 1993.

[37] J. Ersfolk, G. Roquier, J. Lilius, and M. Mattavelli. Scheduling of dynamic dataflow programs based on state space analysis. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, pages 1661–1664, 2012.

[38] D. B. MacQueen G. Kahn. Coroutines and networks of parallel processes. In *Information Processing*, pages 993–998, 1977.

[39] L. Garcia, G.M. Callico, D. Barreto, V. Reyes, T. Bautista, and A. Nunez. Towards a configurable SoC MPEG-4 advanced simple profile decoder. *Computers Digital Techniques, IET*, 1(5):451–460, 2007.

[40] M. Geilen, T. Basten, and S. Stuijk. Minimising buffer requirements of synchronous dataflow graphs with model checking. In *Proceedings - Design Automation Conference*, pages 819–824, 2005.

[41] Frank Ghenassia. *Transaction-Level Modeling with SystemC: Tlm Concepts and Applications for Embedded Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[42] E. M. Girczyc. Loop winding-a data flow approach to functional pipelining. In *Proceedings of the IEEE ISCAS*, pages 382–385, May 1987.

[43] G. Goossens, J. Rabaey, J. Vandewalle, and H. De Man. An efficient micro-code compiler for applications specific DSP processors. *IEEE Trans. Computer-Aided Design*, 9:925–937, June 1990.

[44] J. Gorin, M. Raulet, Y-L Cheng, H. Y Lin, N. Siret, K. Sugimoto, and G.G. Lee. An RVC dataflow description of the AVC Constrained Baseline Profile decoder. In *Image Processing (ICIP), 2009 16th IEEE International Conference on*, pages 753–756, 2009.

[45] Ruirui Gu, Jörn W. Janneck, Mickaël Raulet, and Shuvra S. Bhattacharyya. Exploiting statically schedulable regions in dataflow programs. *J. Signal Process. Syst.*, 63(1):129–142, April 2011.

[46] Ruirui Gu, J.W. Janneck, S.S. Bhattacharyya, M. Raulet, M. Wipliez, and W. Plishker. Exploring the Concurrency of an MPEG RVC Decoder Based on Dataflow Program Analysis. *Circuits and Systems for Video Technology, IEEE Transactions on*, 19(11):1646–1657, 2009.

[47] Z. Gu, M. Yuan, N. Guan, M. Lv, X. He, Q. Deng, and G. Yu. Static scheduling and software synthesis for dataflow graphs with symbolic model-checking. In *Proceedings - Real-Time Systems Symposium*, pages 353–364, 2007.

[48] Rajesh K. Gupta and Giovanni De Micheli. Hardware-software cosynthesis for digital systems. *IEEE Des. Test*, 10(3):29–41, July 1993.

[49] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. Spark: a high-level synthesis framework for applying parallelizing compiler transformations. In *International Conference on VLSI Design*, pages 461–466, 2003.

[50] Soonhoi Ha, Choonseung Lee, Youngmin Yi, Seongnam Kwon, and Young-Pyo Joo. Hardware-software codesign of multimedia embedded systems: the peace. In *Embedded and Real-Time Computing Systems and Applications, 2006. Proceedings. 12th IEEE International Conference on*, pages 207–214, 2006.

[51] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305 –1320, September 1991.

[52] B. S. Haroun and M. I. Elmasry. Architectural synthesis for DSP silicon compiler. *IEEE Trans. Computer-Aided Design*, 8:431–447, April 1989.

[53] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–363, June 1977.

[54] J.K. Hollingsworth and B.P. Miller. Parallel program performance metrics: a comparison and validation. In *Supercomputing '92., Proceedings*, pages 4–13, 1992.

[55] G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison Wesley, 2004.

[56] Ramsey Hourani, Ravi Jenkal, W.Rhett Davis, and Winser Alexander. Automated Design Space Exploration for DSP Applications. *Journal of Signal Processing Systems*, 56(2-3):199–216, 2009.

[57] FastVdo http://fastvdo.com/FV264. FV264-H.264/AVC ASIC IP CORE.

[58] CoreEL Technologies http://www.coreel.com/pages/productsDigitalVideoH264CBPDecode.aspx. H.264 CBP Decoder.

[59] Faraday Technology Corporation http://www.faraday tech.com. MPEG-4 ASP ASIC IP Core.

[60] http://www.tik.ee.ethz.ch/ moses/. Portable interpreter infrastructure for simulating a hierarchical networks of actors.

[61] Cheng-Tsung Hwang, Yu-Chin Hsu, and Youn-Long Lin. Pls: A scheduler for pipeline synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12:1279–1286, September 1993.

[62] Ki Soo Hwang, A. E. Casavant, Ching-Tand Chang, and M. A. d'Abreu. Scheduling and hardware sharing in pipelined data paths. In *Proc. ICCAD-89*, pages 24–27, November 1989.

[63] J. Janneck, I.D. Miller, and D.B. Parlour. Profiling dataflow programs. In *Proceedings of the IEEE International Conference on Multimedia and Expo*, pages 1065–1068, 2008.

[64] J. W. Janneck, I. D. Miller, D. B. Parlour, G. Roquier, M. Wipliez, and M. Raulet. Synthesizing hardware from dataflow program: an MPEG-4 simple profile decoder case study. In *Proceeding of the 2008 IEEE Workshop on Signal Processing Systems (SiPS), October 2008*, 2008.

[65] Young-Pyo Joo, Sungchan Kim, and Soonhoi Ha. Efficient hierarchical bus-matrix architecture exploration of processor pool-based MPSoC. *Journal of Design Automation for Embedded Systems*, 2013, May.

[66] Hong-Shin Jun and Sun-Young Hwang. Design of a pipelined datapath synthesis system for digital signal processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(3):292–303, September 1994.

[67] G. Kahn. The semantics of a simple language for parallel programming. *Information Processing*, pages 471–475, 1974.

[68] Eunsuk Kang, Ethan Jackson, and Wolfram Schulte. An approach for effective design space exploration. In Radu Calinescu and Ethan Jackson, editors, *Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems*, volume 6662 of *Lecture Notes in Computer Science*, pages 33–54. Springer Berlin Heidelberg, 2011.

[69] Joachim Keinert, Martin Streubühr, Thomas Schlichter, Joachim Falk, Jens Gladigau, Christian Haubelt, Jürgen Teich, and Michael Meredith. SystemCoDesigner: an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications. *ACM Trans. Des. Autom. Electron. Syst.*, 14(1):1:1–1:23, January 2009.

[70] P. Kidwell. The universal turing machine: a half-century survey. *Annals of the History of Computing, IEEE*, 18(4):73–, 1996.

[71] B. Kienhuis, E. Rijpkema, and E. Deprettere. Compaan: deriving process networks from matlab for embedded signal processing architectures. In *Hardware/Software Codesign, 2000. CODES 2000. Proceedings of the Eighth International Workshop on*, pages 13–17, 2000.

[72] M. Kthiri, P. Kadionik, H. Levi, H. Loukil, A. Ben Atitallah, and N. Masmoudi. A parallel hardware architecture of deblocking filter in h264/avc. In *Electronics and Telecommunications (ISETC), 2010 9th International Symposium on*, pages 341–344, 2010.

[73] K. Lahiri, A. Raghunathan, and S. Dey. System-level performance analysis for designing on-chip communication ar-chitectures. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20:768–783, 2001.

[74] Choonseung Lee, Sungchan Kim, and Soonhoi Ha. A Systematic Design Space Exploration of MPSoC Based on Synchronous Data Flow Specification. *Journal of Signal Processing Systems*, 58(2):193–213, February 2010.

[75] E. A. Lee and S. Ha. Scheduling strategies for multiprocessor real-time DSP. In *GLOBECOM '89: IEEE Global Telecommunications Conference and Exhibition. Communications Technology for the 1990s and Beyond*, volume 2, pages 1279–1283, Los Alamitos, CA, USA, November 1989.

[76] E.A. Lee and D.G. Messerschmitt. Synchronous Data flow. *Proceedings of the IEEE*, 75(9):1235 – 1245, 1987.

[77] E.A. Lee and T.M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773 –801, May 1995.

[78] C. E. Leiserson and J. B. Saxe. Optimizing synchronous systems. *Journal of VLSI Computer Systems*, 1(1):41–67, 1983.

[79] S. Li, X. Wei, T. Ikenaga, and S. Goto. A VLSI architecture design of an edge based fast intra prediction mode decision algorithm for H.264/AVC. In *Proceedings of the ACM Great Lakes Symposium on VLSI, GLSVLSI*, pages 20–24, 2007.

[80] Y. Li and Y. He. Bandwidth optimized and high performance interpolation architecture in motion compensation for H.264/AVC HDTV decoder. *Journal of Signal Processing Systems*, 52(2):111–126, 2008. Cited By (since 1996):4.

[81] Yu Li, Yanmei Qu, and Yun He. Memory Cache Based Motion Compensation Architecture for HDTV H.264/AVC Decoder. In *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, pages 2906–2909, 2007.

[82] Weichen Liu, Zonghua Gu, Jiang Xu, Yu Wang, and Mingxuan Yuan. An efficient technique for analysis of minimal buffer requirements of synchronous dataflow graphs with model checking. In *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 61–70, New York, NY, USA, 2009.

[83] C. Lucarz. *Dataflow programming for systems design space exploration for multicore platforms*. PhD Thesis-EPFL, June 2011.

[84] C. Lucarz, M. Mattavelli, M. Wipliez, G. Roquier, M. Raulet, J. Janneck, I. Miller, and D. Parlour. Dataflow/actor-oriented language for the design of complex signal processing systems. In *Proceedings of the 2008 Conference on Design and Architectures for Signal and Image processing (DASIP)*, November 2008.

[85] Sharad Malik, Kanwar Jit Singh, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Performance optimization of pipelined logic circuits using peripheral retiming and resynthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(5):568–578, May 1993.

[86] E. Martin, O. Sentieys, H. Dubois, and J. L. Philippe. Gaut: An architectural synthesis tool for dedicated signal processors. In *European Design Automation Conference - Proceedings*, pages 14–19, 1993.

[87] M. Mattavelli, I. Amer, and M. Raulet. The reconfigurable video coding standard. *IEEE Signal Processing Magazine*, 27(3):159–164+167, 2010.

[88] M. Mattavelli, S. Casale-Brunet, A. Elguindy, E. Bezati, R. Thavot, G. Roquier, and J. Janneck. Methods to explore design space for MPEG RVC codec specifications. *Signal processing Image Communication, Elsevier*, 2013.

[89] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, New Jersey, USA, 3rd edition, 1994.

[90] G. De Micheli. Hardware synthesis from c/c++ models. In *Design, Automation and Test in Europe Conference and Exhibition 1999*, pages 382–383, 1999.

[91] G. De Micheli and R. K. Gupta. Hardware/software co-design. *IEEE MICRO*, 85:349–365, 1997.

[92] M. Nadeem, S. Wong, G. Kuzmanov, and A. Shabbir. A high-throughput, area-efficient hardware accelerator for adaptive deblocking filter in H.264/AVC. In *Embedded Systems for Real-Time Multimedia, 2009. ESTIMedia 2009. IEEE/ACM/IFIP 7th Workshop on*, pages 18–27, 2009.

[93] Jens-Rainer Ohm, Gary J. Sullivan, Heiko Schwarz, Thiow Keng Tan, and Thomas Wiegand.

[94] P.R. Panda. Systemc - a modeling platform supporting multiple design abstractions. In *System Synthesis, 2001. Proceedings. The 14th International Symposium on*, pages 75–80, 2001.

[95] N. Park and A. C. Parker. Sehwa: A software package for synthesis of pipelines from behavioral specifications. *IEEE Trans. Computer-Aided Design*, 7:358–370, March 1988.

[96] T. M. Parks. *Bounded Scheduling of Process Networks.* PhD Thesis-University of California-Berkeley, December 1995.

[97] D. Parlour. *CAL Coding Practices Guide: Hardware programming in the CAL Actor language.* Xilinx Inc, jun 2003.

[98] P. G. Paulin and J. P. Knight. Force-directed scheduling for the behavioral synthesis of asic's. *IEEE Trans. Computer-Aided Design*, 8:661–679, June 1989.

[99] K. Pingali and A. Arvind. Efficient demand-driven evaluation. part 1. *ACM Transactions in Programming Language Systems*, 7(2):311–333, 1985.

[100] K. Pingali and A. Arvind. Efficient demand-driven evaluation. part 2. *ACM Transactions in Programming Language Systems*, 8(1):109–139, 1986.

[101] Antoni Portero, G. Talavera, Marc Moreno, J. Carrabina, and F. Catthoor. Methodology for energy-flexibility space exploration and mapping of multimedia applications to single-processor platform styles. *Circuits and Systems for Video Technology, IEEE Transactions on*, 21(8):1027–1039, 2011.

[102] R. Potasman, J. Lis, A. Aiken, and A. Nicolau. Loop winding-a data flow approach to functional pipelining. In *Proceedings of the 27th Design Automation Conference*, pages 444–449, 1990.

[103] H. Prabhu, S. Thomas, J. Rodrigues, T. Olsson, and A. Carlsson. A GALS ASIC implementation from a CAL dataflow description. In *NORCHIP, 2011*, pages 1–4, 2011.

[104] E. Ashcroft R. Jagannathan. Eazyflow: A hybrid for parallel processing. In *International Conference in Parallel Processing*, pages 161–165, 1984.

[105] I.E. Richardson. *H.264 and MPEG-4 Video Compression: Video Coding for Next-generation Multimedia.* Wiley, 2003.

[106] G. Roquier, M. Wipliez, M. Raulet, J.W. Janneck, I.D. Miller, and D.B. Parlour. Automatic software synthesis of dataflow program: An MPEG-4 simple profile decoder case study. In *Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on*, pages 281–286, 2008.

[107] Ghislain Roquier, Endri Bezati, and Marco Mattavelli. Hardware and software synthesis of heterogeneous systems from dataflow programs. *J. Electrical and Computer Engineering*, 2012.

[108] J.W. Janneck S.C. Brunet, M. Mattaveli. Buffer optimization based on critical path analysis of a dataflow program design. In *IEEE International Symposium on Circuits and Systems 2013 (ISCAS 2013)*, pages 1–4, 2013.

[109] M. Mattaveli J.W. Janneck S.C. Brunet, C. Alberti. Design space exploration of high level stream programs on parallel architectures: a focus on the buffer size minimization and optimization problem. In *8th International Symposium on Image and Signal Processing and Analysis (ISPA 2013)*, pages 1–4, 2013.

[110] M. Sen and S.S. Bhattacharyya. Systematic exploitation of data parallelism in hardware synthesis of dsp applications. In *Acoustics, Speech, and Signal Processing, 2004. Proceedings. (ICASSP '04). IEEE International Conference on*, volume 5, pages V–229–32 vol.5, 2004.

[111] M. Shafique, L. Bauer, and J. Henkel. A parallel approach for high performance hardware design of intra prediction in h.264/avc video codec. In *Proceedings -Design, Automation and Test in Europe, DATE*, pages 1434–1439, 2009.

[112] T. Sihvo and J. Niittylahti. H.264/AVC interpolation optimization. In *Signal Processing Systems Design and Implementation, 2005. IEEE Workshop on*, pages 307–312, 2005.

[113] Robert Soulé, Michael I. Gordon, Saman Amarasinghe, Robert Grimm, and Martin Hirzel. Dynamic expressivity with static optimization for streaming languages. In *Proceedings of the 7th ACM international conference on Distributed event-based systems*, DEBS '13, pages 159–170. ACM, 2013.

[114] Gary J. Sullivan, Jens-Rainer Ohm, Woojin Han, and Thomas Wiegand. Overview of the High Efficiency Video Coding (HEVC) Standard. *IEEE Trans. Circuits Syst. Video Techn.*, 22(12):1649–1668, 2012.

[115] M. Thadani, P. P. Carballo, P. Hernández, G. Marrero, and A. Núñez. ESL flow for a hardware H.264/AVC decoder using TLM-2.0 and high level synthesis: a quantitative study. In *Proceedings of SPIE - The International Society for Optical Engineering*, volume 7363, 2009.

[116] R. Thavot, A.A.-H. Ab Rahman, R. Mosqueron, and M. Mattavelli. Automatic mutli-connectivity interface generation for system designs based on a dataflow description. In *Proceedings of the 6th Conference on Ph.D. Research in Microelectronics & Electronics*, 2010.

[117] L. Thiele, S. Chakraborty, M. Gries, and S. Kunzli. A framework for evaluating design tradeoffs in packet processing architectures. *ACM*, pages 880–885, 2002.

[118] William Thies, Michal Karczmarek, Michael I. Gordon, David Z. Maze, Jeremy Wong, Henry Hoffman, Matthew Brown, and Saman Amarasinghe. Streamit: A compiler for streaming applications. Technical Report MIT/LCS Technical Memo LCS-TM-622, Massachusetts Institute of Technology, Cambridge, MA, Dec 2001.

[119] Carl Von Platen. D2C: CAL ARM Compiler. *ACTORS Project (http://www.actors-project.eu)*, 2008-2011.

[120] S. Wang, W. Peng, Y. He, G. Lin, C. Lin, S. Chang, C. Wang, and T. Chiang. A software-hardware co-implementation of MPEG-4 Advanced Video Coding (AVC) decoder with block level pipelining. *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 41(1):93–110, 2005.

[121] J.R. Woodward. Computable and incomputable functions and search algorithms. In *Intelligent Computing and Intelligent Systems, 2009. ICIS 2009. IEEE International Conference on*, volume 1, pages 871–875, 2009.

[122] Cui-Qing Yang and Barton P. Miller. Critical path analysis for the execution of parallel and distributed programs. In *Proceedings - International Conference on Distributed Computing Systems*, volume 8, pages 366–373, 1988.

[123] M. A. Yukish. *Algorithms to identify pareto points in multi-dimensional data sets.* PhD Thesis, Pennsylvania State University, August 2004.

[124] H. Yviquel, E. Casseau, M. Wipliez, and M. Raulet. Efficient multicore scheduling of dataflow process networks. In *Signal Processing Systems (SiPS), 2011 IEEE Workshop on*, pages 198–203, 2011.

[125] B. Zatt, L. M. de L. Silva, A. Azevedo, L. Agostini, A. Susin, and S. Bampi. A reduced memory bandwidth and high throughput HDTV motion compensation decoder for H.264/AVC High 4:2:2 profile. *Journal of Real-Time Image Processing*, 8(1):127–140, 2013.

[126] Claudiu Zissulescu, Todor Stefanov, Bart Kienhuis, and Ed Deprettere. Laura: Leiden architecture research and exploration tool. In Peter Cheung and GeorgeA. Constantinides, editors, *Field Programmable Logic and Application*, volume 2778 of *Lecture Notes in Computer Science*, pages 911–920. Springer Berlin Heidelberg, 2003.

Related Personal Publications

[1] A.A.-H. Ab Rahman, H. Amer, A. Prihozhy, C. Lucarz, and M. Mattavelli. Optimization methodologies for complex FPGA-based signal processing systems with CAL. In *Design and Architectures for Signal and Image Processing (DASIP), 2011 Conference on*, pages 1–8, nov. 2011.

[2] A.A.-H. Ab Rahman, S. Casale-Brunet, C. Alberti, and M. Mattavelli. Design Space Exploration and Refactoring Techniques for Dataflow Programs: MPEG-4 AVC/H.264 Decoder Implementation Case Study. In *Design and Architectures for Signal and Image Processing (DASIP), 2013 Conference on*, pages 1–8, oct. 2013.

[3] A.A.-H. Ab Rahman, S. Casale-Brunet, A. Prihozhy, M. Moghadas, and M. Mattavelli. Optimizing Dataflow Specifications of Signal Processing Systems for Hardware and Heterogeneous Implementations. *IEEE Transactions on Circuits and Systems for Video Technology*, 0(0):1–2, 2013. (In progress).

[4] A.A.-H Ab Rahman, A. Prihozhy, and M. Mattavelli. Pipeline synthesis and optimization of FPGA-based video processing applications with CAL. *EURASIP Journal on Image and Video Processing*, 2011:1–28, 2011.

[5] A.A.-H. Ab Rahman, Casale-Brunet S., and M. Mattavelli. Minimization and Optimization of Buffer Interconnection Sizes for Dynamic Dataflow Programs. In *Acoustics, Speech, and Signal Processing (ICASSP), 39th International Conference on*, page 0, May 2013. (In Progress).

[6] A.A.-H. Ab Rahman, R. Thavot, Casale-Brunet S., E. Bezati, and M. Mattavelli. Design Space Exploration Strategies for FPGA Implementation of Signal Processing Systems using CAL Dataflow Program. In *Design and Architectures for Signal and Image Processing (DASIP), 2012 Conference on*, pages 1 –8, oct. 2012.

[7] A.A.H Ab-Rahman, R. Thavot, M. Mattavelli, and P. Faure. Hardware and software synthesis of image filters from CAL dataflow specification. In *2010 Conference on Ph.D. Research in Microelectronics and Electronics (PRIME)*, pages 1–4, 2010.

[8] H. Amer, A.A.-H. Ab Rahman, I. Amer, C. Lucarz, and M. Mattavelli. Methodology and technique to improve throughput of FPGA-based CAL dataflow programs: Case study of the RVC MPEG-4 SP Intra decoder. In *Signal Processing Systems (SiPS), 2011 IEEE Workshop on*, pages 186–191, oct. 2011.

[9] S. Casale-Brunet, E. Bezati, A.A.H. Ab Rahman, J. Janneck, and M. Mattavelli. Analysis and Optimization of Streaming Application: The TURNUS Approach. *ACM Transactions on Embedded Computing Systems: Special Issue on Application of Concurrency to System Design*, 0(0):1–2, 2013. (In progress).

[10] A. Prihozhy, E. Bezati, A.A.H Ab Rahman, and M. Mattavelli. Synthesis and Optimization of Pipelines for Hardware Implementation of Dataflow Programs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 0(0):1–2, 2013. (In progress).

[11] R. Thavot, A.A.-H. Ab Rahman, R. Mosqueron, and M. Mattavelli. Automatic multi-connectivity interface generation for system designs based on a dataflow description. In *Proceedings of the 6th Conference on Ph.D. Research in Microelectronics & Electronics*, 2010.

Ab Al-Hadi Bin Ab Rahman

Curriculum Vitae

PERSONAL DETAILS

Birthdate	October 15, 1981
Birthplace	Selangor, Malaysia
Address	Route Cantonale 39, 1025 St Sulpice VD, CH
Phone	(078) 9422163
Email	`alhadi.abrahman@epfl.ch/alhadi.abrahman@gmail.com`
Nationality	Malaysian
Languages	Malay (mother tongue), English (fluent), French (intermediate)

EDUCATION

Ph.D. Signal Processing 2009-2013

École Polytechnique Fédérale de Lausanne, Switzerland

Thesis: Optimizing Dataflow Programs for Hardware Synthesis.

M.Eng. Electronics and Telecommunications Engineering 2004-2006

Universiti Teknologi Malaysia

Thesis: VLSI Design and Implementation of Adaptive Equalizers.

B.S. Computer Engineering 2000-2004

University of Wisconsin-Madison, USA

WORK EXPERIENCE

Doctoral Assistant 2009-present

École Polytechnique Fédérale de Lausanne, Switzerland - SCI-STI-MM

Responsible for study and research on all aspects of hardware design and implementation using dataflow programming. Also contributed to the study of heterogeneous system implementation with CAL.

Research Assistant 2007-2009

Universiti Teknologi Malaysia - Image processing/VLSI research lab

Responsible for research in image processing algorithms for stereoscopic vision, and research in full-custom and semi-custom digital IC design. Also taught courses and labs in digital electronics.

AWARDS

- UTM/MOHE scholarship for Ph.D. studies at a top foreign university (2009).
- UTM/MOHE scholarship for Master studies in Malaysia (2004).
- MARA scholarship for undergraduate studies at an American top university (2000).

PROFESSIONAL MEMBERSHIP

1. *Graduate engineer* - Board of engineering Malaysia (2006-present)
2. *Student member* - Institute of Electrical and Electronic Engineers, IEEE (2002-present)
3. *Member of signal processing society* - IEEE (2009-present)
4. *Member of circuits and systems society* - IEEE (2009-present)

INTEREST/HOBBY

Travelling, cooking, sports.