EFFICIENT C-BASED SOC ARCHITECTURES AND DESIGN METHODOLOGIES

YIDI LIU

M.Phil

The Hong Kong Polytechnic University

2016

The Hong Kong Polytechnic University

Department of Electronic and Information Engineering

Efficient C-based SoC Architectures and Design Methodologies

Yidi LIU

A thesis
submitted in partial fulfilment of the requirements
for the degree of

Master of Philosophy

August 2015

CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

Yidi LIU (Name of student)

Abstract

ITRS [1] suggest that by 2020 a 10x productivity increase for designing complex SoCs is needed. Two main factors are predicted to help achieving this goal. The first is the re-use of components. ITRS estimates that around 90% of the SoCs will be composed of re-used components. Secondly, the use of new design methodologies to raise the level of abstraction, i.e. High-Level Synthesis (HLS).

Companies have started to rely on High-Level Synthesis (HLS) to increase their design productivity and making use of third party behavioral IPs (3PBIPs) to meet their tight schedules.

C-Based design has many advantages compared to traditional RTL design. The most salient ones include, the increase in design productivity, which allows design teams to meet the increasingly stringent time-to-market requirements, the ability to create smaller designs compared to hand-coded RTL due to its ability to maximize resource sharing and the possibility of generating a set of different micro-architectures with different area vs. performance trade-offs without having to modify the original behavioural description, also called Design Space Exploration (DSE).

In traditional RTL design it is virtually impossible to do DSE as it would involve having to re-write the RTL description completely in order to create the new micro-architecture. Moreover it is common practice not to modify any hardware (HW) block that has been fully verified, even if a more efficient architecture could be achieved in subsequent designs, due to the cost of having to re-verify the new implementation.

HLS is a single process synthesis method, which takes individual behavioral descriptions as inputs and performs resource allocation, scheduling and binding on each of them to obtain an RTL description, which can efficiently execute it.

As mentioned previously, HLS allows designers to generate micro-architectures with different area vs. performance trade-offs. Typically, high-performance designs will consume more HW resources as loops are fully unrolled and functions inlined, while low performance designs tend to be much smaller as resources can be shared, functions do not need to be inlined and loops executed sequentially.

This thesis investigates effective design methods to automatically generate SoC architectures with unique area vs. performance trade-offs when these are fully described at the behavioral level. State-of-the-art HLS tools now include system level design capabilities which allow this. Two main cases are studied: Static schedule and dynamic tasks schedule architectures.

Scheduling and mapping approaches can be classified as online and off-line algorithms. Off-line algorithms have shown to be able to obtain superior results by exploring a larger portion of the design space. These static/off-line methods schedule multiple applications on a system and can be used in order to reduce the complexity of the HW and hence reduce its area and power overheads. On the other side, online methods are much more flexible. It is therefore important to study both approaches.

Publications

1. Y. Liu, B. Carrion Schafer, "Optimization of Behavioral IPs in Multi-Processor System-on-Chips", ASP-DAC, 2016.

2. Y. Liu, B. Carrion Schafer, "Adaptive Combined Macro and Micro-Exploration of Concurrent Applications mapped on shared Bus Reconfigurable SoC", *Electronic System Level Synthesis Conference (ESLsyn)*, San Francisco, 2015.

3. Y. Liu, B. Carrion Schafer, "HW Acceleration of Multiple Applications on a Single FPGA", *International Conference on Field-Programmable Technology (FPT)*, pp. 284-285,2014.

Acknowledgements

First of all, I would like to give my sincere appreciation and thanks to my supervisor Dr. Benjamin Carrion Schafer for his guidance and encouragement in my MPhil period. It is his illuminable instruction and persistent patience that inspires me to work on my research. I benefit a lot from his fantastic and experienced biography as well as the expert advice and intelligent thoughts. It is my pleasure and honour to work under my supervisor and I believe this two years research study is my invaluable experiment.

Next I want to thank my co-supervisor Prof. Francis C.M. Lau. His serious working attitude, strict concept insistence and insightful comment have deeply impressed me and impel me to regard research work in a serious manner. It is interesting to talk with Prof. Lau about the life and career in Hong Kong.

Furthermore, I am grateful to the Hong Kong Polytechnic University and the Electronic and Information Department for their generous and consistent support during my master study. My thanks also go to the clerical staffs including Cora, Janice, Shirley and the other members in the General Office of my department, whose help on the administrative field are much appreciated.

I also want to thank my colleges Anushree Mahapatra, Nandeesh Veeranna, Shuangnan Liu, Dong Liu and the others for their help in my research work. My thanks also go to my friends Junjie Huang, Diyang Xue, Kuo Wang, Xiaojun Huang, Xiaotong Li, Huiling Zhou, Peiya Li, Zhenhui Situ, Linchuang Xu, Qing Liang and the others for my interesting and exciting university life in Hong Kong.

Finally, I would like to give my special thanks to my parents and my girlfriend for their unselfish love, concern and support during my MPhil study period and throughput my life.

Contents

List of Figures

List of Tables

Glossary of Abbreviations

AHB	Advanced High-performance Bus
ALAP	As Late As Possible
ALU	Arithmetic Logical Unit
AMBA	Advanced Micro-controller Bus Architecture
API	Application Program Interface
ASAP	As Soon As Possible
ASIC	Application Specific Integrated Circuit
CPU	Central Processing Unit
CWB	CyberWorkBench
DFG	Data Flow Graph
DMA	Direct Memory Access
DSE	Design Space Exploration
DSP	Digital Signal Processing
EDA	Electronic Design Automation
ESL	Electronic System Level
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FU	Functional Unit
GPU	Graphics Processing Unit
HW	Hardware
HWAcc	Hardware Accelerator
HDL	Hardware Description Language
HLS	High Level Synthesis
HPC	High Performance Computing
IC	Integrated Circuit
IP	Intellectual Property
MPSoC	Multiple Processor System-on-Chip
NoC	Network on Chip
QoR	Quality of Results
RAM	Random Access Memory

ROM	Read-Only Memory
RTL	Register Transfer Level
SoC	System-on-Chip
SW	Software
VHDL	Very High Speed IC Hardware Description Language
VLSI	Very Large Scale Integration

1 Introduction

C-based design is finally being deployed extensively in industry for commercial designs. The increase in productivity combined with the improvement in the quality of the results of commercial HLS tools has convinced many design teams to make the transition. This transition is nevertheless gradual and currently most of the applications being targeted are Digital Signal Processing (DSP) related as HLS has shown in the past that it can rival hand-coded RTL designs for these type of applications.

One advantage of HLS vs. traditional RT-level design methods is that HLS allows the generation of different micro-architectures with unique area vs. performance trade-offs without having to modify the original behavioral description. This is typically done by setting different synthesis options to e.g. synthesize arrays as memories or registers, unroll loops or not and inline functions or not. At the same time state-of-the-art HLS tools now support the complete generation of SoCs at the behavioral level by including bus generator tools, interface generators, hierarchical design methods and RTL to C conversion tools. RTL to C tools allow to convert back legacy RTL code into behavioral code, which can in turn be integrated into the same SoC with newly created C code.

This work makes use of these unique capabilities of C-based VLSI design and HLS tools to generate complete SoCs in C with different area vs. performance trade-offs. This is often called Design Space Exploration (DSE). The aim of DSE is to find Pareto optimal configurations as fast as possible. Because the design space is often extremely large, heuristics have been developed to find these optimal configurations and compared to exhaustive search methods which are able to find the optimal results. Because often exhaustive search methods do not lead to any results at a reasonable time, the results obtained are often referred to *dominating* configurations, as their optimaliity cannot be guaranteed.

1.1 Contribution of this Thesis

This thesis investigates design methods for finding dominating configurations for SoCs with static and dynamic schedules completely described in C. The major contributions made by this work are as follows:

1. Develop a complete automated flow for creating complete behavioral-level SoCs, including the HW/SW partitioning in order to achieve higher performance and more energy efficient systems.

2. Study of the HW/SW partitioning on computationally intensive applications or tasks given in a high-level description by mapping the most computationally intensive kernel onto an FPGA for PC's with re-configurable computing co-processor boards or onto re-configurable fabric within SoCs.

3. Investigate efficient design space exploration methods for static scheduled SoCs with shared buses.

4. Develop exploration methods for finding dominating SoC configurations for dynamic scheduled SoC architectures with shared bus using HLS tools' bus generators, in particular for AMBA AHB bus and optimize the accelerators micro-architecture.

1.2 Thesis Structure

The thesis is divided into five chapters. The first chapter review previous related work in the areas touched in this work. In particular previous work about automatic partitioning, scheduling and system exploration. Chapter 3 introduces how HLS works and describes in detail its three main steps : (1) allocation, (2) scheduling and (3) binding. Automatic HW acceleration by partitioning behavioral descriptions automatically and synthesizing these with HLS is discussed in Chapter 4, mainly for micro-processors enhanced with FPGA-based accelerator co-processor cards. This chapter also studies the efficiency of these accelerators when multiple tasks are mapped concurrently onto a single FPGA. A complete system exploration flow combining micro-exploration (HLS DSE) and macro-exploration for static scheduled, shared bus

systems is presented in Chapter 5. Chapter 6discusses dynamic scheduling SoC systems explorations based on bus arbiters using AMB AHB buses and for heterogeneous MPSoC architecture. The last two chapters, Chapter 7 and Chapter 8 discuss the results obtained by our proposed methods, draw conclusions as well as show future work, respectively.

2 Literature Review

This work touches multiple topics, which have been well studied in the past. In particular, HW/SW co-design, System-Level Design Space Exploration (DSE), High-Level Synthesis DSE and static bus scheduling. Therefore only a representation of the most relevant previous work is given in this section.

HW/SW partitioning has been studied extensively in the past. A good overview of HW/SW work done so far can be found at [2]. The main approach so far has been to convert a given set of tasks into a directed acyclic graph (DAG) and using heuristics [3] or mixed integer linear programming (MILP) [4] methods to iteratively map different operations in the DAG to either SW or HW. The cost function is typically the execution time of the tasks, while the target architecture is also given. Early HW/SW partitioning was done by [5] and [6].

Hendry et al. [7] developed a HW/SW partitioning tool for embedded systems with multiple hardware processes called COSYN by partitioning the design into blocks and selecting the ones with highest speedup potential iteratively. In [8], two heuristic methods, based on simulated annealing and tabu search, were presented to minimize communication cost and improve overall parallelism. In [9] a genetic algorithm approach was taken in order to make a partitioning decision for re-configurable system. A static method [10] based on Bayesian belief networks (BBN) handles the interactions between each unit and makes a classification decision by propagating evidence and casual messages along nodes. Additionally, partitioning algorithm was studied on re-configurable designs. Noguera et al [11] presented a partitioning algorithm for dynamically re-configurable architecture to minimizing reconfiguration latency. In [12] a complete synthesis and partitioning method for adaptive re-configurable computing system (SPARCS) was presented targeting multiple-FPGA systems and performing HLS on their tasks graphs. Closely integrating of partitioning and synthesis was used to predict each partitioning size and performance.

There are also some companies dedicated to HW acceleration using FPGAs. In particular, HW/SW partitioning are used in financial application acceler-

ation[1] because banks can afford the expensive equipment and it is easy to quantify the cost savings of having a faster system (e.g. customized low latency trading HW system or Value at Risk computation). These companies manually analyze the algorithms to be accelerated and then manually create a suitable architecture for this particular application. In some cases they have developed semi-automatic flows to speed the development of the system up [13].

A very important aspect of any HW/SW partitioning method is the simulation of these systems. The authors in [14] and [15] introduced some first approaches to be able to simulate complete systems composed of processors and HWAccs.

Scheduling is another important process affecting the performance of HW/SW co-design. It has been shown to be an NP-complete problem [16], thus many heuristics have been proposed in the past. Usually it can be classified into static and dynamic scheduling. A comparison of different heuristics can be found at [17], where static scheduling has shown to create superior systems compared to dynamic scheduling systems.

Static scheduling is easily implemented with fixed or pre-decided priorities providing a relatively simple complexity. It usually relies on the compiler to make decision on optimization or improvements and built in device as a priority logic. E.A. Lee and D.G. Messerschmitt [18] used static scheduling in synchronous data flow (SDF) to reduce or eliminate runtime overhead. Integer linear programming (ILP) formulation [4] and ant colony optimization [19] were applied to solve scheduling problems based on DAG, considering heterogeneous architectures.

With regards to dynamical scheduling, the task schedule is decided at runtime according to practical situation such as dependence, performance and deadline constraint. Earliest Time First (ETF) algorithm [20] and the Dynamic Level Scheduling (DLS) algorithm [21] are two common used dynamic algorithms. Bauer et al. [22] considered two metrics to decide tasks' priorities based on performance in local scheduling and dependencies in global scheduling.

Static scheduling has the advantage over dynamic scheduling that it is easier to guarantee a reproducible result with a simpler architecture. In contrast

[1] Celoxica, www.celoxica.com and Maxeler, www.maxeler.com

dynamic scheduling systems are more flexible as they can handle different types of systems [23]. Furthermore, a very interesting scheduling technique based on quasi-static techniques was developed to combine advantages of both static and dynamic scheduling in [24].

With regards to system level design space exploration (DSE), the main purpose is to generate different configurations focusing on specific targeted objective(s) to be optimized. The authors in [25] presented a DSE methodology for on-chip communication architecture optimization. Givargis et al. [26] presented a technique to explore the design space of a parametrized SoC architecture on power/performance trade-off by performing aggressive pruning using the different parameter dependencies. In [27], six ESL exploration approaches were described to make decision in behavior, architecture, structure or performance.

Regrading MPSoC DSE, meta-heuristics [28] or pruning techniques [26] have been mostly used. They are also based on the type of simulation abstraction used to model the MPSoC ranging from sequential simulators (e.g. QEMU [29]) and SimpleScalar [30]), transaction level models (TLM) (e.g. OVP [31] to cycle-accurate modelling (e.g. HORNET [32]). However, most of them perform the exploration at higher abstraction levels in order to speed the process up, with its consequent estimation errors. In [33], HLS is used for the accelerators in MPSoCs so that a HLS DSE method is developed to obtain best-performance MPSoCs, but this work uses analytic models to estimate the performance of each newly generated system and hence cannot consider the effect of e.g. bus congestion and but arbiter policy into account.

In [34] it has been shown that workloads might change in modern MPSoC and hence need to be taken into account when performing MPSoCs DSE. Quan et al. [35] extended this work by introducing a hybrid task mapping method that combined static mapping exploration and a dynamic mapping optimizer.

In summary, this work touches areas of HW/SW partitioning, system-level design space exploration, dynamic and static tasks scheduling and system-level accurate simulations. Although these topics have been studied in the past, the novelty of our work is in the complete description of the complex SoC at the behavioral level and the creation of an exploration system on top of commercial HLS tools which allows our work to obtain realistic and accurate results.

The following chapters introduce how HLS is applied in HW/SW design on heterogeneous system.

3 High Level Synthesis

High-Level Synthesis (HLS) is a process that transforms un-timed behavioral descriptions into RT-level descriptions which can efficiently execute these. In contrast to traditional RT-level design process, which makes use of low level Hardware Descriptions Languages (HDLs), HLS typically accepts as inputs high-level programming languages e.g. ANSI-C or C++, allowing designers to focus on the functional behavior and not on the implementation details, which are time consuming and error prone. This leads to an increase in design productivity allowing design teams to meet their tight schedules.

Another advantage of raising the level of abstraction is that less number of lines of code are required compared to RTL descriptions. It has been shown that HLS can reduce the number of lines of code by an average of 10 times [36], which not only leads to short design cycle, but also less bugs and makes it easier to verify and maintain the source code.

3.1 Design Flow

Fig.3.1 [37] shows an overview of the typical complete HLS process. HLS takes as inputs the behavioral description to be synthesized, a set of design constraints and technology libraries of the target ASIC or FPGA. The first step involves parsing the description and creating a formal model. Allocation, scheduling and binding are the main three steps of behind HLS and all of them work interdependently.

Allocation defines the type and the number of HW resources given in the technology library required by the input descriptions. Scheduling times the behavioral description in time step following the operations' dependencies without violated any area and/or latency constraints. Finally binding maps the scheduled operations to individual HW resources. The last step involves writing out the synthesizable RTL code, which can in turn be passed to the logic synthesizer. Thus the final RTL architecture depends on input specification, the HW resource library and the synthesis constraints (e.g. target clock frequency).

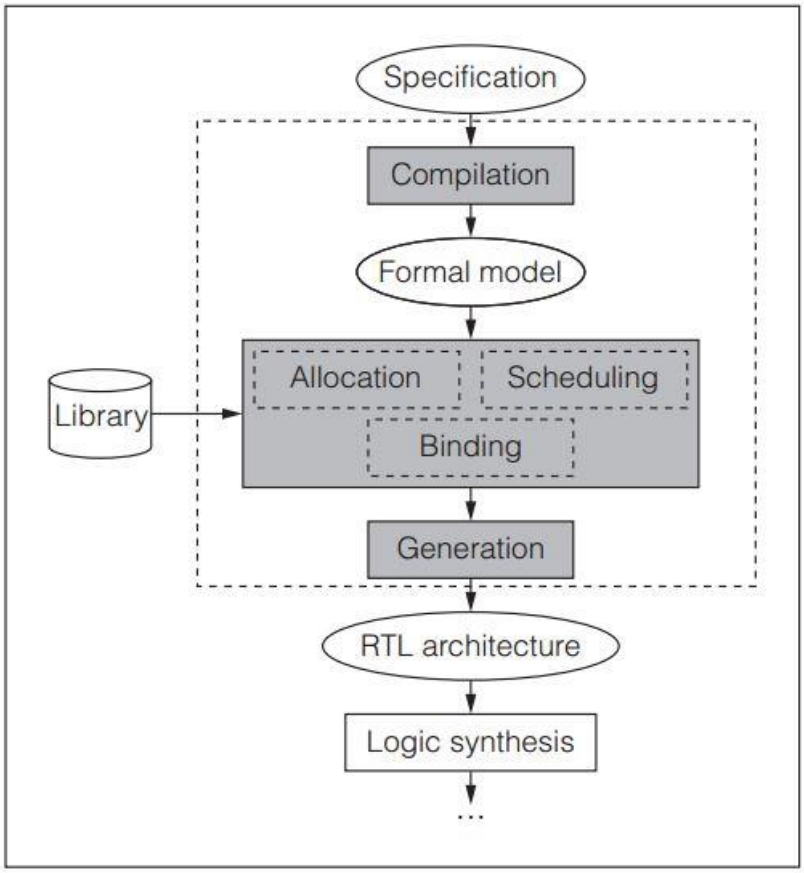


Fig. 3.1: High-level synthesis design flow

3.1.1 Compilation/Parsing

One of the advantages of HLS is that like with any high-level SW programming language, the input description can be compiled, simulated and debugged using standard SW environments (e.g. gcc or g++, gdb and gprof). These characteristics allow designers to test and debug the applications much easier and faster compared to HDLs and provide useful feedback to the designer earlier on so that changes can be easier performed than at the RT-level.

However, not all SW constructs are supported in HLS. Non-synthesizable constructs e.g. dynamic memory allocation and recursion need to be considered when using HLS. Some HLS tools ignore some of these constructs automatically, while others create an error, asking the designers to correct them.

3.1.2 Allocation

The very first step once the behavioral description has been parsed is the allocation of HW components, mainly Functional Units (FUs). The behavioral description is parsed and a FU constraint file (FCNT) is generated. This file contains the number and type of FUs required. By default the HLS tool will try to maximize the parallelism as much as possible a hence allocate as many FUs as possible. The user can at any time overwrite this constraint file manually, setting the maximum number of FUs that the synthesizer can instantiate.

```
// FIR filter
1.  int data[9];          // Input
2.  int result;           // Output
3.  int coeff[9];         // Global Coefficient
4.  main() {              // Process starts
5.      int i, sum;
6.      sum = 0;
7.      LOOP : for (i=0; i<9; i++)
8.          sum += data[i] * coeff[i];
9.      result = sum;
10. }
```

(a)

Adder	1
Multiplier	1
Comparator	1
Accumulator	1

(b)

Adder	9
Multiplier	9

(c)

Fig. 3.2: FU allocation example for FIR filter given in ANSI-C: (a) ANSI-C codes of 9-tap FIR filter; (b) Allocation results required for NOT-unroll "LOOP" requirement; (c) Allocation results required for ALL-unroll "LOOP" requirement

Fig.3.2(a) shows a simple example of a 9-tap FIR filter, which basically computes the sum-of-products (SOP) for the data and coefficients. Line 1 and line 2 declare the inputs and output of the filter, respectively, while lines 7-8 compute the SOPs. As mentioned previously one of the advantages of HLS is that different micro-architectures with unique area vs. performance can

be obtained without having to modify the behavioral descriptions, and only some synthesis options. In this case, the SOP loop could be fully unrolled or not unrolled. If the loop is not unrolled, the process only requires 1 32-bit singed adder, 1 32-bit signed multiplier, 1 comparator and 1 accumulator as Fig.3.2(b) shows, while if the loop is fully unrolled, 9 adders and 9 multipliers are needed as as Fig.3.2(c) shows. Loop unrolling clearly leads to circuits of larger area, as more FUs are required, but also brings the benefit of increasing the performance. In this case reducing the latency from 9 clock cycles to 1 clock cycle, assuming clock period is long enough. Theoretically, in this case, unrolling leads to an approximate $9\times$ larger in resources and $9\times$ faster in performance design. It should be noted that unrolling does not always guarantee a better performance design due to constraints such as clock period and due to the costs of the multiplexers required to share the FUs, especially for FPGAs.

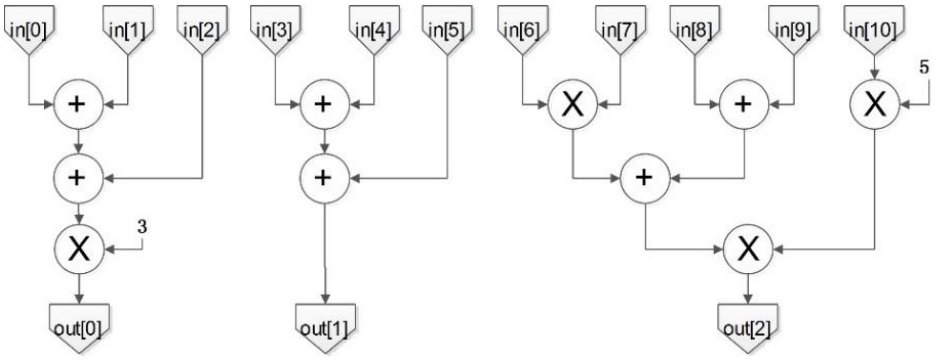
3.1.3 Scheduling

The next step after resource allocation is the scheduling. Scheduling determines in which clock step each operation should be executed at, so that no precedence constraint and data dependence is violated. Scheduling process basically follows the DFG restricted by resource and timing constraint. In order to make full use of the resources and reduce the latency, independent operations can be scheduled at the same clock step and executed in parallel. Operation chaining is available by directly connecting operations' outputs to inputs of the next operations in the DFG, while additional registers are needed if the connection crosses multiple clocks. Also multi-cycle operation are typically allowed if the clock period is too small and/or the FU delay is too large.

Many scheduling algorithms have been proposed in HLS [38]–[40]. The two most basic scheduling algorithms, As soon as possible (ASAP) and As late as possible (ALAP), are presented to demonstrate how HLS scheduling works here. ASAP maps operations to their earliest possible start time while ALAP maps operations to the latest possible start time, without any precedence violation. Fig.3.3(a) shows a behavioral description to be scheduled and Fig.3.3(b) its DFG with 6 additions (subtraction can be considered as special addition) and 4 multiplications. Assuming as many FUs as possible, but that the clock period only allows one addition/multiplication in each control step,

```
1.  int in[11];          // Input
2.  int out[3];          // Output
3.  main() {             // Process starts
4.      int a, b, c;
4.      a = in[0] + in[1];
6.      b = in[6] * in[7];
7.      c = b +in[8] + in[9];
8.      out[0] = (a-in[2]) * 3;
9.      out[1] = in[3] + in[4] + in[5];
10.     out[2] = in[10] * 5 * c;
11. }
```

(a)

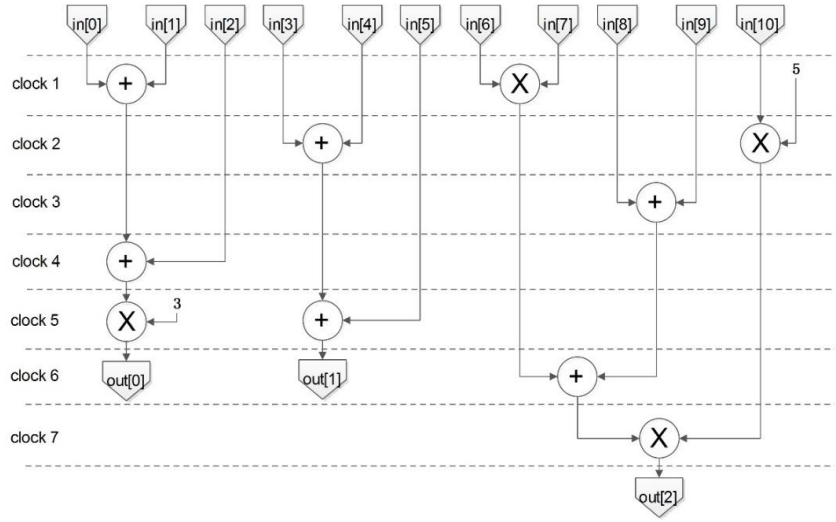


(b)

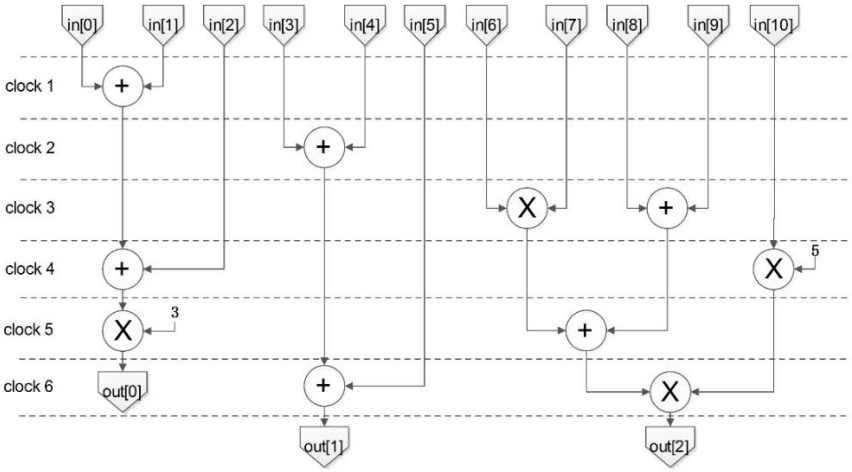
Fig. 3.3: Scheduling example: (a) Example codes in ANSI-C; (b) DFG of source code given in (a)

then the fastest circuit would have a latency of 3 clock cycles and would require 3 adders and 2 multipliers.

When a constraint of one adder and one multiplier is given, only one adder and one multiplier can be executed in each clock step. In this case, two different scheduling results are obtained based on if the scheduler uses an ASAP or ALAP algorithm as shown in Fig.3.4(a) and (b), respectively. It can be seen that the results are different: ASAP returns a 7-clock cycles result while ALAP returns a 6-clock which is also the minimum clock required in this case. However, both algorithms cannot guarantee an optimal solution. The scheduling algorithm as seen has a great impact on the synthesis result.



(a)



(b)

Fig. 3.4: Scheduling results of Fig.3.3 example with one adder and one multiplier constraint: (a) ASAP scheduling; (b) ALAP scheduling

3.1.4 Binding

The last step in HLS is the binding stage. Binding maps each operation to a FU and each variable to a register. Every operation is assigned to a specific FU given in the FCNT file which can execute that operation. Since it is possible that one FU is shared by more than one operation at different clock step, which is also called resource sharing, multiplexers are needed to assign the correct data to the input at the right time and deliver the output to the correct register. A Finite State Machine (FSM) is typically also generated to generate the control signal for these multiplexers. The binding can affect the routability

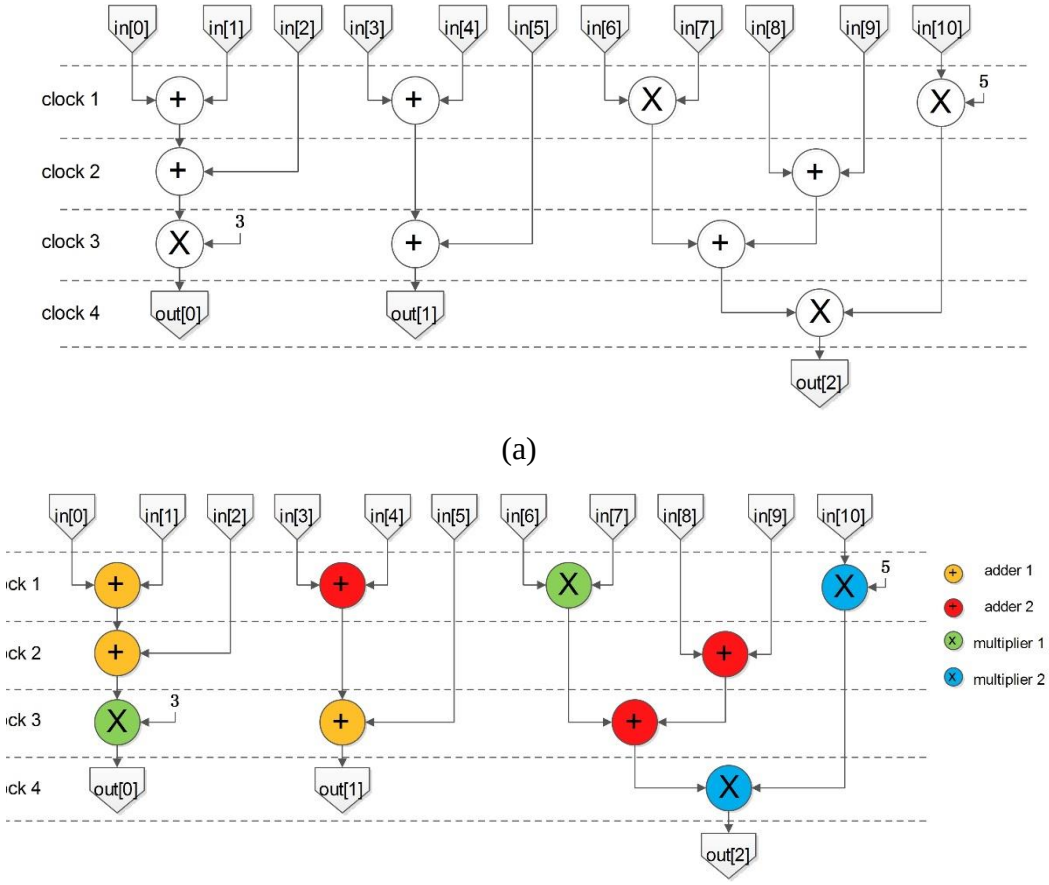


Fig. 3.5: Binding example for Fig.3.3 example with constraint TWO adder and TWO multiplier: (a) ASAP scheduling result; (b) Binding result with weighted bipartite matching algorithm

of the final circuit and hence its wire-length and critical path. It is therefore a very important stage.

Many different binding algorithm have been developed in the past. The main goal is to optimize some objectives, e.g. area, power, wire-length and/or routability. The authors in [41] presented a weighted bipartite matching binding to balance the utilization of each FU. A simple demonstration of binding is given in Fig.3.5 with the weighted bipartite matching binding algorithm. The example code used is the same as that in Fig.3.3(a) with a constraint of two adders, yellow and red, and two multipliers, green and blue, using the scheduling result of an ASAP algorithm shown in Fig.3.5(a) requiring totally 4 clocks cycles. Fig.3.5(b) shows the binding result when every operation is bound to specific operators and each operator appears once

at each clock step. In this case, every adder performs 3 additions and every multipliers performs 2 multiplications. The FU resources are shared using multiplexers at the FUs inputs and outputs.

3.1.5 RTL Generation

HLS synthesizes high-level behavioral descriptions into RTL codes following the three steps described in the previous subsections. The last step of HLS is the RTL generation based on the results obtained from these steps. Fig.3.6 displays the typical RTL architecture generated by HLS, containing a controller (e.g. FSM) and a data path (e.g. dataflow engine). The controller generates the control signals for the datapath, in order to guarantee the correct execution of the complete circuit. It manages the inputs passing them into the corresponding FUs at the correct state and generates the control signals for the multiplexers. It also stores the data to specific storage elements, e.g. registers or RAM. Once the RTL architecture is generated and described in any HDL (e.g. Verilog or VHDL), the next step would involve passing it to a logic synthesizer.

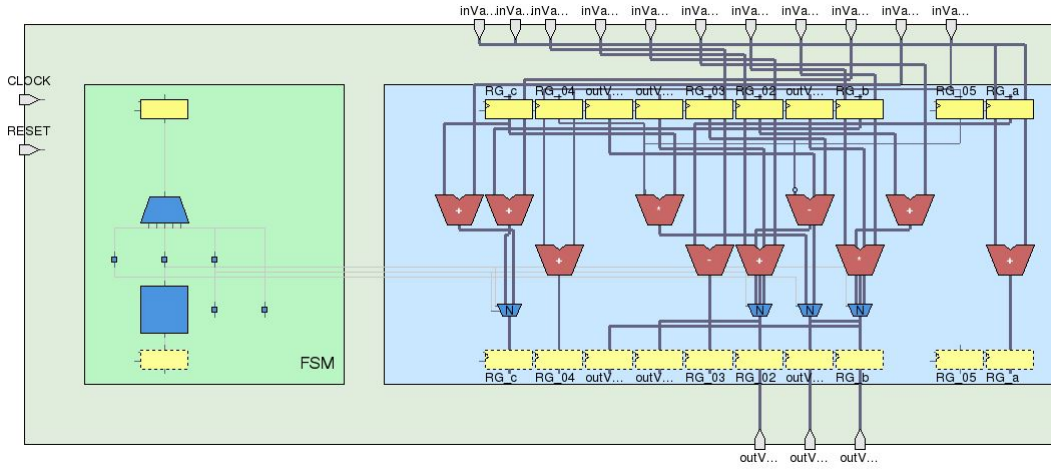


Fig. 3.6: Typical RTL architecture

3.2 Commercial HLS Tools

The need to raise the level of abstraction to become more productive has also been reflective in the number of commercial HLS tools available. Table 3.1 highlights the main vendors, their tool name and the input language

supported. It can be seen that SystemC is the common language supported by all of the HLS tools. Although not a language by itself, it is a C++ class for HW modelling, it has been widely adopted because it has been standardized by the IEEE and allows the modelling of HW related constructs and allows natively the modelling of concurrency.

Table 3.1: Examples for HLS tool

Vendor	Tool Name	Supported Languages
Cadence (Forte)	Cynthesizer	SystemC
Cadence	C-to-Silicon	C, C++, SystemC
Calypto	CatapultC	C++, SystemC
NEC	CyberWorkBench	C, SystemC
Xilinx	Vivado HLS	C, C++, SystemC

3.3 Summary

In summary, HLS synthesizes behavioral descriptions to HDL descriptions by performing allocation, scheduling and binding. The main advantage of HLS is that by taking behavioral descriptions as input it dramatically reduces the gap between SW and HW design and increases the design productivity. Also, HLS DSE allows designers to explore the micro-architectures. There are plenty of HLS tools available in the market as well as open source or open binary ones. In this thesis, CyberWorkBench will be used.

4 Hardware Acceleration

With the introduction of larger FPGA, it is possible to map and concurrently execute multiple applications onto a single FPGA. This allows building heterogeneous accelerator systems with multiple applications running concurrently by mapping the computationally intensive kernels onto the FPGA and the controlling portion to the CPU(s), interfacing both parts via traditional buses e.g. PCIe and USB.

This chapter explores the acceleration of multiple computationally intensive applications given in ANSI-C using HLS mapped onto a single FPGA given the area and communication bandwidth constraint. Coarse-grained partitioning is used as commercial HLS tool is used and the optimal mapping decision can be found by dynamic programming.

4.1 Motivations

Much research has been done in the acceleration of computational intensive applications using FPGAs. Most of them nevertheless are focused on the acceleration of the most computationally intensive kernel [7], [8], [12], [13], which is often the inner-most loop. In all cases they only accelerate a single kernel mapped onto the re-configurable fabric or map a larger application onto multiple FPGA, and hence only deal with a single application to be accelerated at a time. However, it is possible to accelerate multiple application concurrently on the same FPGA as the size and complexity of FPGAs increases.

Fig.4.1 shows a block diagram of the proposed system. It includes multiple applications running on either multiple processors, multiple processor cores or multi-threaded single core system being executed concurrently and a single FPGA is used to accelerate to most computationally intensive kernels of the entire system. This could mean that a single application, the one which benefits most from HW acceleration is fully mapped onto the FPGA or in case that all of the applications benefit from HW acceleration that some parts of all of the applications are mapped onto the FPGA. The main constraints are the size

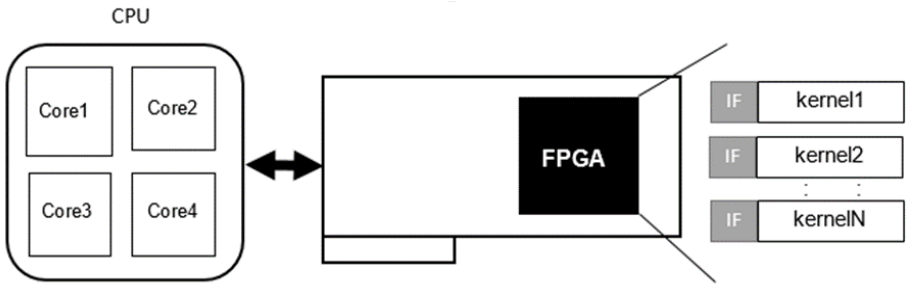


Fig. 4.1: Proposed HW acceleration system overview

of the FPGA and the bandwidth between the host processor and the FPGA board.

Mapping multiple kernels onto the same FPGA poses unique challenges that need to be addressed. Fig.4.2 shows system speed-up with the increase of kernels mapped onto the FPGA. Two cases are represented. The first one does not include bandwidth constraints. It can be seen that the speed-up grows with the number of kernels mapped onto the FPGA. The second one shows the speed-up when a maximum communication bandwidth between the FPGA and the microprocessor is set as a constraint. In this case the speed-up that can be achieved is obviously smaller than that in the un-restricted case. It also grows until a maximum speed-up point and then decreases. This inflection point appears when the bandwidth saturates. Mapping more kernels on the FPGA degrades the speed-up from this point on and at one point, the performance starts getting even worse than running all the applications purely on SW.

Knowing these speed-up limitations it is important to develop a method that takes the communication bandwidth and area overhead of mapping a kernel onto the FPGA into consideration in order to only map those kernels that will maximum the acceleration of the entire system.

4.2 Proposed Flow

Fig.4.3 shows an overview of our proposed flow. The input to our method is a set of computationally intensive applications (APi) which want to be accelerated given in ANSI-C. Our method then automatically partitions these applications into different kernels (Ki) and continues by mapping only those

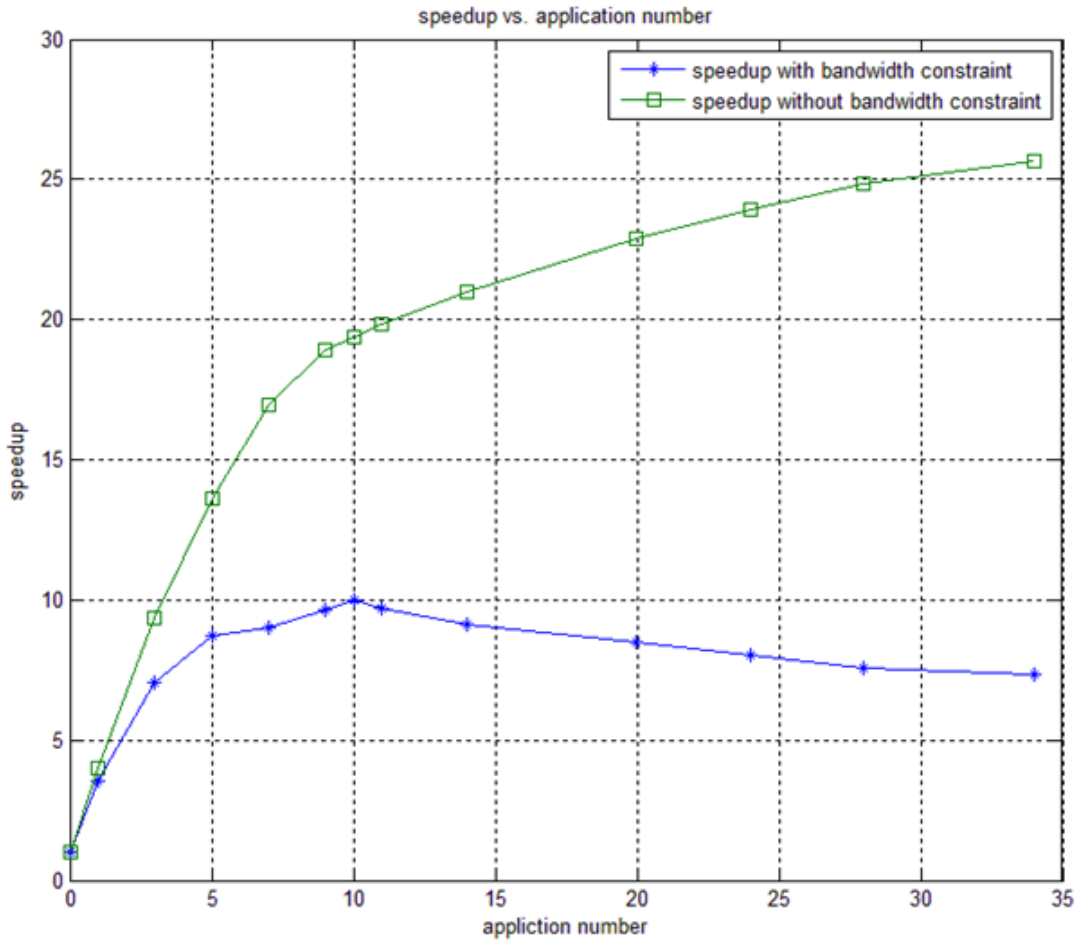


Fig. 4.2: System speed-up vs. number of kernels mapped to HW device with bandwidth and no bandwidth constraint

kernels which maximize the total system acceleration ($SAcc$) onto the FPGA under a given area (MA) and bandwidth (BW) constraint.

In the first step, our method partitions original SW program into separate kernels. The granularity of each kernel is a function,thus decided by the programming style in each application (also called coarse-grained partitioning). Other approaches [28] use task graphs as inputs and can thus create more optimal (finer) partitions (also called fine-grained partitioning). Because a commercial HLS tool is used to synthesize each partitioning kernel, such fine-grained partitioning cannot be done and the internal controllability is lacked. These are the drawbacks caused by endorsing HLS tool in partitioning stage.

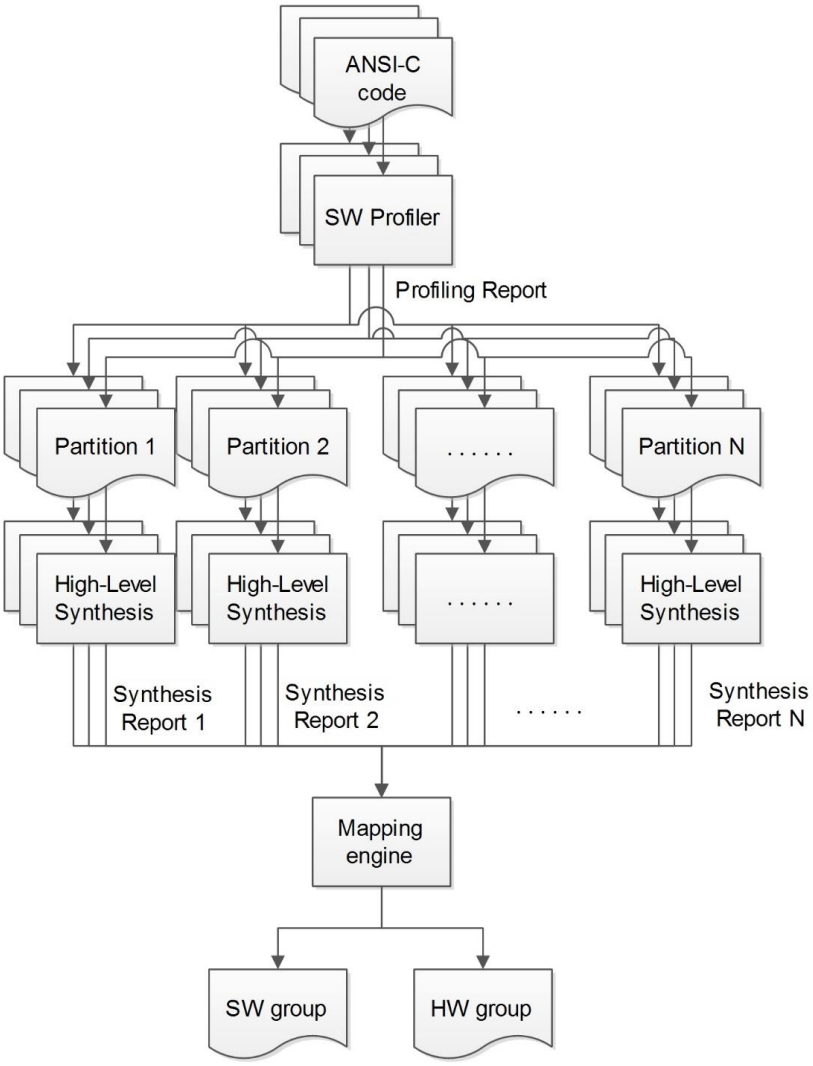


Fig. 4.3: Flow chart overview of proposed flow

However, HLS allows proposed method to get more accurate area and timing results comparing to normal estimation.

The proposed method then continues by performing HLS on each of these kernels and by extracting their design parameters, in particular area (A) the performance measured as latency (L). Based on these parameters bandwidth required (B) is computed for each kernel, which allows method to fully characterize each kernel of each application. Once the partitioning and synthesis stage have been completed, the method continues by allocating the individual kernels of the different application onto the FPGA and figures out the optimal mapping decision with best performance.

The optimal kernel mapping problem concerns how to optimally map computationally intensive kernels $K_{list} = \{K_1, ..., K_N\}$ merged from each application APP_i onto the FPGA with given area MA and a maximum bandwidth BW between the microprocessor and the FPGA so that the acceleration $SAcc$ of the entire system is maximized. This optimization problem is normally decomposed in two main steps[42]: (1) Partitioning of the applications into N disjoint sets of kernels K_n. (2) For all the kernels K_n devise an optimal mapping so that the cost function is maximized (in this case acceleration) under a set of given constraints (here are FPGA area MA and communication bandwidth BW). This problem can be reduced to a 0-1 knapsack problem which can be efficiently solved using dynamic programming [43] shown in Algorithm 1.

Step 1: Reduce Invalid Partition: As the kernels are function-based, each function has different properties, e.g. some have larger number of arguments requiring a large bus bandwidth, while some are too simple to be accelerated and others do not meet the constraints. Thus before making any mapping decision, such kernel are removed from the candidate list in order to reduce the complexity system.

Step 2: Dynamic Programming: This is the main step of the proposed method which is displayed in Algorithm 1. It takes as inputs N valid kernels K_{list} which can be accelerated on the FPGA, the total area constraint MA and the bandwidth constraint BW of the system.

New systems can be created by adding kernels one by one recursively. The system performance can be calculated by

$$SP(n, A) = max(SP(n-1, A), SP(n-1, A - A_n) + P_n) \tag{4.1}$$

where $SP(n, A)$ is the system performance contain 1 to n kernels with area limit of A and, A_n and P_n are the individual area and performance of n^{th} kernel. Each time a new kernel is considered, the resultant system is checked through area limit range from 0 to $(MA - kArea)$ and candidate list is updated if result is better. Because area is not the only considered constraint resultant system is remained if If it is dominated result. In order to avoid the repetition of the area and performance estimation each time a new system is generated, this data is stored in a memory $partition_{mem}$ for fast retrieval, which is shown from line 9 to 14 in Algorithm 1.

Step 3: Report Solution: Once systems considering all kernels have been generated, the optimal (i.e. highest acceleration) solution is selected from the $partition_{list}$ and reported.

4.3 Experimental Results

The experiment are run on an Intel dual 2.40GHz Xeon processor with 16 GBytes of RAM running Linux Fedora release 19. The HLS tool used is CyberWorkBench (CWB) of NEC with version of 5.6 and application designs are taken from the open source Synthesizable SystemC Benchmark suite (S2CBench) [44], re-written in ANSI-C codes. Table 4.1 describes the system benchmarks consisting of application benchmarks used in experiment targeting to architecture with FPGA of Xilinx's Virtex 4 XC4V35SX-10 and PCIe 3.0 of bandwidth 8GT/s.

Table 4.1 lists the system benchmarks to be used in later experiment. The first column is the short name of task benchmarks: Snow3G is a stream cipher producing a key stream; Kasumi is a block cipher used in mobile communication systems; AES is advanced encryption standard cipher encryption algorithm; IDCT is an inverse discrete cosine transform application; Disp is a estimator of disparity; and the last 3, Syn1, Syn2 and Syn3, are synthetic benchmarks composing of Adpcm (Adaptive differential pulse-code modulation), FIR (9-tap FIR filter), Sobel (edge-detection algorithm) and an application for average of 8 values. The second and third columns represent the number of synthesized look-up table and the size of IO ports needed for the top module, respectively. Columns S1-S8 indicate the number of task instantiations involved in system benchmark. The last row displays the total number tasks involved in system.

The constraints used in this experiment are that $MA = 15000(LUT)$ and $BW = 8GT/s$ with FPGA operation frequency of $f = 100MHz$. Table 4.2 shows the found solution by brute-force (exhausted) method and dynamic programming method, where *Funcs* is the number of valid partition, *LUTs* is the logic resources of final solution, *B* is the bus utilization percentage and *Run* is the execution time in seconds. BF indicates brute-force method and DP indicates dynamic programming method. It is easily found that dynamic programming can find out the optimal solution as brute-force does additionally it is much efficient in execution time.

4.4 Summary

This chapter presents a method for mapping kernels from different applications onto a single FPGA to maximize overall system acceleration. It starts by automatically partitioning behavioral descriptions using a SW profiler continuing by synthesizing each partitioning using a commercial HLS tool to extract the area and performance of each new kernel. Here is where HLS also plays an important role, as this information is important because it allows to estimate the speed-up of the entire system composed of these HW accelerated kernels. A dynamic programming is finally presented which can efficiently find the best system under given area and bandwidth constraints.

Algorithm 1: Dynamic Programming For HW Acceleration Partition

Input: $K_{list} = \{K_1, ..., K_n, ..., K_N\}$, N, MA, BW
K_{list}: Kernels list
N: Size of K_{list}
MA: Maximum allowed area
BW: Bandwidth of bus interface

Output: $MD_{opt} = \{MK_{list}, Acc_{max}\}$
MD_{opt}: The optimal mapping decision
MK_{opt}: The mapped kernels list
Acc_{max}: Acceleration of decided mapping system

$VK_{list} \leftarrow delete\_invalid\_kernel(K_{list})$;
$partition_{list} \leftarrow None$
for $(K \in VK_{list})$ **do**
  $kArea \leftarrow get\_kernel\_area(K)$;
  $partition_{mem} \leftarrow None$;
  for $(area \in [0, MA - kArea])$ **do**
    for $(partition \in partition_{list}[area])$ **do**
      $partition \leftarrow partition + K$;
      if $(partition \in partition_{mem})$ **then**
        $(sArea, sBW, sAcc) \leftarrow partition_{mem}[partition]$;
      else
        $(sArea, sBW, sAcc) \leftarrow system_{est}(partition)$;
        $partition_{mem} \leftarrow partition$;
      end
      if $(isValid\_partition(sArea, sBW, sAcc)$ **then**
        $partition_{list}[sArea] \leftarrow partition$;
        $delete\_nonDominate\_partition(partition_{list}[sArea])$
      end
    end
  end
end
$MD_{opt} \leftarrow partition_{list}[MA]$;
return MD_{opt};

Table 4.1: Complex System Benchmarks

Bench	LUT	IO	S1	S2	S4	S5	S3	S6	S7	S8
Snow3G	1979	608			1	1	1	1	1	1
Kasumi	890	66	1	1			1	1		1
AES	2905	258		1	1	1	1		1	1
IDCT	7155	514	1	1	1		1		1	1
Disp	5036	442	1	1			1	1		1
Syn1	3649	218			1	1		1	1	1
Syn2	6456	282				1		1	1	1
Syn3	13704	333						1	1	1
Tasks			3	4	4	4	5	6	6	8

Table 4.2: Experimental Results

					BF	DP
Bench	Funcs	LUTS	B[%]	Sp-Up	Run[s]	Run[s]
S1	4	13081	81.1	7.74	<1	<1
S2	11	8831	60.8	22.32	<1	<1
S3	23	13569	93.5	43.61	39	<1
S4	26	14498	95.1	37.94	363	<1
S5	18	10810	70.9	38.01	1	<1
S6	31	14748	97.1	42.50	13448	<1
S7	33	13744	99.8	30.22	60132	<1
S8	37	13593	97.1	23.27	NA	<1

5 Static Schedule SoC Design Space Exploration

As the demand for more computing power increases, heterogeneous systems are being proposed as new computing paradigm. These systems comprise a processors, on where the SW is executed, and HW accelerators (HWAccs) as well as memory. All these components are connected through a bus, bus hierarchy or even NoC. Bused with arbiters provide these systems with more flexibility, while also causing larger area and performance overheads compared to fixed schedule systems. Also, since HLS is single process method, it is hard to explore the design space of an entire integrated system.

In this chapter, a fixed schedule SoC architecture is considered as underlying architecture and methods to reduced bus conflict, reduce area and performance overheads are developed. A combined macro-(system) and micro-exploration (HLS DES) for these architectures is presented. The result of the system exploration is a trade-off curve of unique configurations of different area vs. throughput.

5.1 Motivations

Many data-intensive applications manifest predictive and repetitive data patterns. This is mainly because data-intensive applications usually have no or less control structures, while the control-intensive section which mainly depends on runtime situation is mapped onto the processor(s). Hence, the performance of these system can be statically estimated at design time and off-line algorithm for scheduling and mapping used as it has been shown that these off-line methods can lead to superior solutions by exploring a larger design space [19].

5.2 Design Exploration Flow

Fig.5.1 shows an overview of our proposed flow. Our method can be decomposed into three phases. The first two are pre-characterization phases, which identify the most computationally intensive kernels from each of the applications and perform a HLS DSE on each of the kernels to be mapped onto the re-configurable fabric. The first phase performs automatic HW/SW partitioning and passes each of the kernels to the HLS explorer in order to get the smallest designs for each latency within a given latency range (explained in detail in the next section). Lastly, the third phase performs the scheduling and exploration of the different area vs. performance system configurations. We define the following terms:

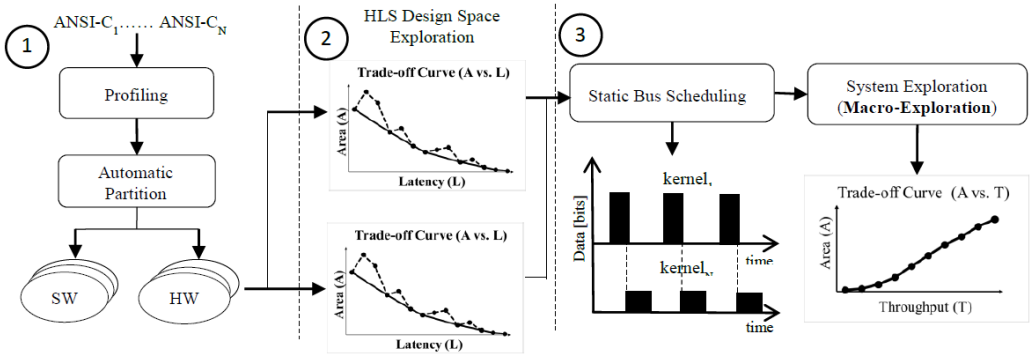


Fig. 5.1: Complete flow overview composed of three phases: 1. automatic HW/SW partitioning; 2. individual processes HLS DES; 3. bus scheduling and system exploration

Definition:System S is comprised of a set of concurrent and independent tasks T, which have to be mapped onto the heterogeneous system. Thus $S = \{T_1, T_2, ..., T_N\}$

Task T is an application given as a behavioral descriptions, which is partitioned into a HW and SW part in the partitioning stage. The HW partition is further explored with the HLS design space explorer, hence

$$T_i = SW \cup HW$$

$$HW = \{D_1 = \{A_1, L_{li}\}, ..., D_N = \{A_N, L_{ri}\}\}$$

where D is a design on the trade-off curve with area A and unique latency L. The objective is to find valid schedules leading to the Pareto-optimal

configurations of the system when area vs. throughput are used as exploration criteria, where the System Throughput ST of a system of N tasks is computed as:

$$ST = \sum_{i=1}^{N} \frac{IO_i}{L_i} f_s \tag{5.1}$$

where IO_i and L_i are the bitwidth and latency of task T_i, receptively, and f_s is the overall system frequency. Here, it is assumed all tasks are in the same operation frequency as a static scheduling method is used.

The input to our method is a set of independent computationally intensive applications, given in ANSI-C, which will be executed concurrently on the heterogeneous system. Our method profiles these applications and identifies the most computationally intensive kernels. The granularity of each kernel is a function and hence is decided by the programming style in each application (also called coarse-grained partitioning), which has been described in Chapter 4.

Once the HW/SW partition is done and the bus interfaces inserted, our method continues by performing a HLS DSE for each of the HW kernels. Starting the VLSI design process from behavioral descriptions has the additional benefit of allowing the generation of different micro-architectures with unique area vs. performance trade-offs, by setting different synthesis directives. Details are given in the next subsection.

The result of a traditional DSE is a trade-off curve containing only dominating designs. These designs are also called Pareto-optimal, but because the design space is too large, Pareto-optimality cannot be guaranteed and hence, these designs are referred to as dominating designs. The obtained trade-off curve is a typically strictly decreasing monotonous trade-off curve (if area vs. latency is considered as exploration target). In this work, not only the dominating designs are recorded, but also the smallest designs for each latency from the fastest to the slowest design. The fastest design $D_{fastest} = \{A_{max}, L_{min}\}$ corresponds to the design with the smallest latency (L_{min}), but largest area (A_{max}) , while the smallest design $D_{smallest} = \{A_{min}, L_{max}\}$, corresponds to the smallest area (A_{min}), but largest latency (L_{max}) one. The trade-off curves in Fig.5.1 (phase 2: High-Level Synthesis Design Space Exploration) shows the result of the exploration. This is very important in our method as invalid bus schedules might result in an increase in the latency for some of the kernels and

hence an increase or decrease in area, because a design with larger latency can now be used.

Once the trade-off curve for each application is created, our method continues by scheduling all the designs found during the DSE of all the different applications mapped onto the system. The number of combinations is therefore extremely large and our efficient scheduling method is able to find valid schedules quickly. Finally the trade-off curve with all dominating valid schedules is created and reported. In this case throughput is used as a performance metric as each design mapped onto the system will have different latencies. The resultant trade-off curve is therefore strictly increasing monotonous (see Fig.5.1 phase 3). The next subsections explain in detail each of the main steps of our proposed method.

5.2.1 HW/SW Partitioning

The pre-characterization stage takes all the applications that need to be considered for acceleration in ANSI-C as inputs. It then continues by profiling each of the application using a standard SW profiler (i.e. gprof). The profiler outputs the number of times each function (kernel) was called and the total execution time spent on each of the functions. This information is used to determine which kernel to map onto the HW accelerator in the mapping stage. Our method then continuous by partitioning each application into disjoint parts. For each partition a new synthesizable ANSI-C description containing only the kernel is generated and modified to make it synthesizable for the commercial HLS tool used in this work. This involves specifying the inputs and outputs of this process as required by the ANSI-C subset of the HLS vendor used in this work [45]. Once the partitions have been done, our method proceeds by performing a HLS DSE on each of the extracted kernels.

5.2.2 HLS Design Space Exploration

One of the advantages of C-based VLSI design over traditional RT-level is that it allows the generation of micro-architectures with different area vs. performance trade-offs without the need of modifying the original behavioral description. This is typically done by setting different global synthesis options which apply to the entire behavioral description, limiting the number of

functional units (FUs) to control the amount of resource sharing and/or by specifying synthesis directives in the form of pragmas at the behavioral description. This allows to e.g. control if an array should be synthesized as RAM, registers or expanded or if a loop should be unrolled, not unrolled or pipelined.

The explorer developed in this work targets only synthesis directives as this is the most important exploration knob as it fixes the underlying micro-architecture. The explorer performs two passes. In the first pass, different micro-architectures are generated by setting unique combinations of pragmas, while the second pass generates designs for those latencies for which the previous pass could not find any designs. In detail:

Step 1: Pragma Explorer The explorable synthesis directives SD are the genes to compose of a chromosome CR used in algorithm. Usually, array (or memory) type, loop synthesis and function operation performs most impacts on design micro-architecture and performance. Then,

$$OP = \{array, loop, func\}$$

where, for example, $array = \{register, expand, logic, ram, rom\}$, $loop = \{unroll = \{no, partial, all\}, fold\}$ and $func = \{goto, fu, inline\}$.

An initial population of N random chromosomes is generated at the beginning of exploration. The main steps to produce new generation are listed as following:

1. **Coupling**: To product next generation, each member in the population randomly couples with another member and produce offspring with a coupling possibility p_c.

2. **Crossover**: Once two members are coupled, a crossover operation is done on these two chromosome by randomly selecting a cut-point and combining cut chromosome with another and two chromosomes of off-spring are generated.

3. **Mutation**: At a mutation possibility p_m, new produced chromosomes suffer mutation by randomly changing a random selected gene value.

4. **Evolution**: After coupling, crossover and mutation, new chromosomes of offspring are generated and they are synthesized by calling HLS to produce new generation of offspring with properties (i.g. area and latency here). The newly produced offspring replace their parents in population once either condition is met: (i) parent is dominated by the offspring (i.e. all objectives are better than parent's); (ii) offspring improve on one or one more best-so-far objective than parents; (iii) offspring makes up a candidate within latency range.

Once new generation is produced, processing will continues by producing next generation, or stops once (i) algorithm has iterates through a certain number of generation G; (ii) the population is converges to a certain status. Additionally, a chromosome library stores all "dead" and "alive" chromosome in order to avoid repetitive production and improve exploration efficiency. This exploration record not only the dominating designs but also the smallest one found for each unique latency.

Step 2: Latency Explorer The explored result from last step Pragma Explorer determines the latency range as $LR = [L_{min}, L_{max}]$, where, L_{min} is the dominating design found with smallest latency and L_{max} is the design with largest one. As design for each unique latency could not be guarantee, thus new design with larger latency and area penalty by inserting additional register for output results to wait for one more clocks. For dominating design $D_n = \{A_n, L_n\}$, the penalty design is $D_{n+1} = \{A_n + \Delta_{reg}, L_n + 1\}$, where Δ_{reg} is proportional to the size of task output.

Go through whole latency range and keep the smallest design at each unique latency either explored one or penalty one. The final exploration result does not guarantee all designs are dominating ones but the smallest and the curve may look like that of phase 2 in Fig.5.1.

5.2.3 Bus Scheduling and System Exploration

Once partition has been established and each HW partitioned task is characterized with the HLS explorer, a system with tasks configuration is scheduled on the shared bus to estimate system performance. The Patero-optimal solutions could be found with an exhaustive search by scheduling tasks for all possible system configurations. However, the problem grows

exponentially with the number of tasks and number of designs obtained by the explorer. It is therefore impractical for larger systems to search exhaustively. One fast and efficient way for exploration is based on dynamic programming laid out as a tree structure with pruning. The scheduling structure is shown in Fig.5.2. At each level one task is added in scheduling tree and the design candidates are grown at each node of last level. Thus, each leaf of scheduling tree (or each node in level N) is one scheduling decision with selected design candidates of all tasks. There are four main steps needed to perform system exploration.

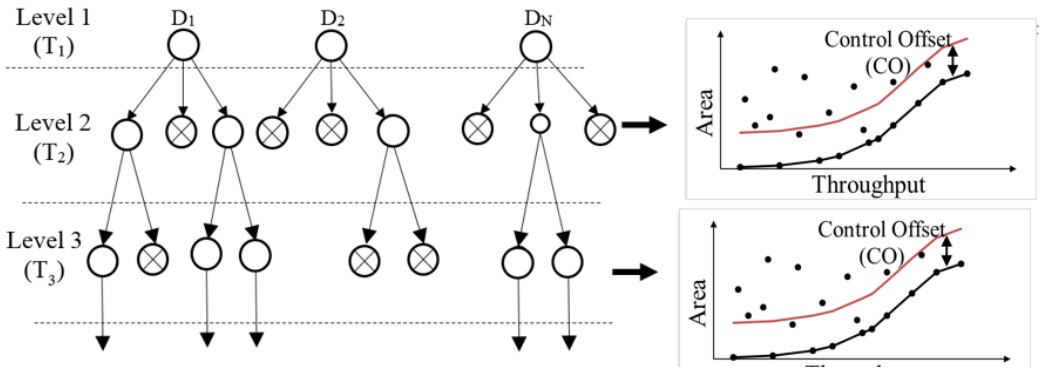


Fig. 5.2: Adaptive schedule method with exploration

Step 1: Task Sorting: The first step is to sort the tasks by the number of designs found during the HLS design space exploration in descending order (from tasks with less designs to the most). $S = \{T_1 < T_2 < \ldots < T_N\}$. This reduces the number of combinations required to schedule as the subsequent pruning stages remove larger number of combinations.

Step 2: Design Scheduling: This step is the main step in the scheduling phase. Fig.5.2 shows the structure used to accelerated the process. The sorted tasks are laid out spatially in a tree structure where each node is a design from each task's trade-off curve. Fig.5.2 shows an overview of this step, where each node at the same level corresponds to a design of the same Task and each edge connects to other designs forming a unique configuration. E.g. Level 1 in Fig. 5.2 corresponds to the designs found during the HLS DSE of T_1. It can be observed that the explorer returned 3 designs, $T_1 = \{D_1 = (A_{max}, L_{min}), D_2 = (A_n, L_{min+1}), D_3 = (A_{min}, L_{max})\}$. All the designs have consecutive latencies ranging in this case from L_{min} until L_{max}. This guarantees that the search space is fully explored. All the designs

of the next task T_2 are scheduled for each of the design of T_1 and a trade-off curve of area vs. throughput is obtained for all the feasible schedules.

The scheduling step proceeds accepting only valid configuration for the scheduled solutions which are feasible and those either on the trade-off curve or within a given Control Offset CO. This control offset can be specified manually as an input to our method. This parameter is extremely important because it controls the amount of pruning in our method. CO=0 leads to a very aggressive pruning technique as only those schedules which are on the trade-off curve in each step are carried forward, while a very lager $CO = \infty$, converts our method into a pure exhaustive search method. This is one of the key contributions of our method. A single parameter can control the quality of results (QoR) vs. the running time of the method.

One key issue when scheduling multiple tasks with different latencies and bandwidth requirements is to quickly detect unfeasible schedules. The HLS explorer returns a list of designs with consecutive latencies. Therefore schedules which require designs to wait in order to avoid bus contingency problems are marked as un-schedulable as another configuration with larger latency will capture a valid schedule. In order to schedule a task, the actual time to transfer data during an access is calculated. E.g. if the bus is 16 bits and the task needs to send 32 bits and the transfer time requires 4 cycles, the total number of communication cycles required are 32/16x 4=8 cycles. This communication is performed as a DMA allowing the communication to overlap with the computation. In order to fully observe if the system S composed of N tasks $S = \{T_1, T_2, \ldots, T_N\}$ is schedulable, our method starts by adjusting the latencies of all the designs $T_1 = \{D_i = (L, A)\}$, making them of equal length. This is done by computing the least common multiplicand (lcm) of the latencies of each designs being scheduled and duplicating each design latency in order to make all designs of the same latency. The lcm will dictate the number of times that each design needs to be duplicated (dup) in each task so that the latency of all the processes are identical dup = lcm/latency. This forces each process to effectively have the same length and hence when scheduling, the effect can be fully observed (required because this work assumes that the tasks are periodically repeating themselves). The scheduler then verifies if the system can have a valid schedule.

Based on these conditions. Out method first verifies if the given system is schedulable or not. If it is not, it discards this system and continues with the

next configuration, speeding therefore the execution of this step considerably up. If the system can be scheduled, the proposed method tries to find a feasible schedule by trying all possible scheduling combinations using an exhaustive search method until a valid scheduled is found, which usually executes quickly.

Step 3: Trade-off curve extraction: Once all the valid schedules for all the tasks within the given tree level are computed, our method continues by deleting all non-optimal configurations and keeping only the dominating ones or the ones within the CO. In this case, throughput is used to measure the performance of the entire system and not latency as in the individual task design space exploration, because each task has a different latency. Our method continues until the very last task is added to the system and returns the final trade-off curve.

5.3 Experimental Results

The experiment are run on an Intel dual 2.40GHz Xeon processor with 16 GBytes of RAM running Linux Fedora release 19. The HLS tool tool used is CyberWorkBench (CWB) of NEC with version of 5.6 and application designs are taken from the open source Synthesizable SystemC Benchmark suite (S2CBench) [44], re-written in ANSI-C codes. Table 5.1 describes the system benchmarks consisting of application benchmarks used in experiment.

Table 5.1: Complex System Benchmarks

Bench	DSE	TC	S1	S2	S3	S4	S5	S6	S7	S8
Snow3G	20	8						1		1
MD5C	77	2	1	1	1	1	1	1	1	1
Adpcm	35	1	1	1	1	1	1	1	1	
Gfilter	21	4						1	1	1
FIR	25	3			1	1	1	1	1	1
Decim	24	1				1	1	1	1	1
Interp	27	2		1	1	1	1	1	1	
IDCT	93	2	1	1	1	1	1	1	1	1
Sobel	20	1					1		1	1
Disp	548	4	1	1	1	1	1	1	1	1
Tasks			4	5	6	7	8	8	9	10

The first column (Bench) shows the names of tasks: Snow3G is a stream cipher producing a key stream; MD5C is a message digest algorithm; Adpcm

is an adaptive differential pulse-code modulation; Gfilter is a graphical filter; FIR is a 9-tap FIR filter; Decim is a 5-stage decimation filter; Interp is 4-stage interpolation filter; IDCT is an inverse discrete cosine transform application; Sobel is a edge-detection algorithm and Disp is a estimator of disparity. The second one of DSE shows the total number of micro-explored designs of each task and the third of TC shows the required data transfer clocks of DMA communication. The value of date transfer clock TC is calculated as:

$$TC = \lceil \frac{IO}{BW} \rceil \tag{5.2}$$

where IO is the port bit of a task required and BW is the bit width of bus. Column S1-S8 indicate the number of task instantiations within system. The last row reports the total number of tasks involved in each system benchmark.

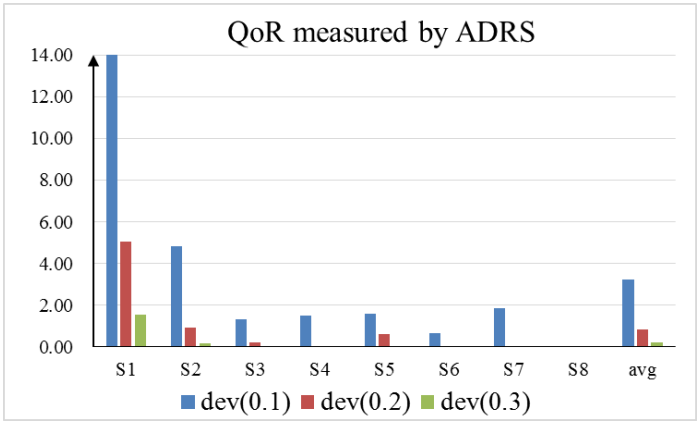
The main problem when comparing different multi-objective function optimization methods is how to measure the quality of the results. There are many unary quality measures for comparing the quality of the multi-objective function optimization methods. ADRS (Average Distance from Reference Set) and dominance (D) are used here as they are the most widely used one [46]. ADRS indicates how close a Pareto-front is to the reference front. The lower the value (ADRS) is, the more similar two Pareto sets are. Dominance is equal to the ratio between the total numbers of points in the Pareto set being evaluated, also present in the reference Pareto set. The higher the value, the better the Pareto set is.

Table 5.2: Experimental Results of macro-exploration for different CO

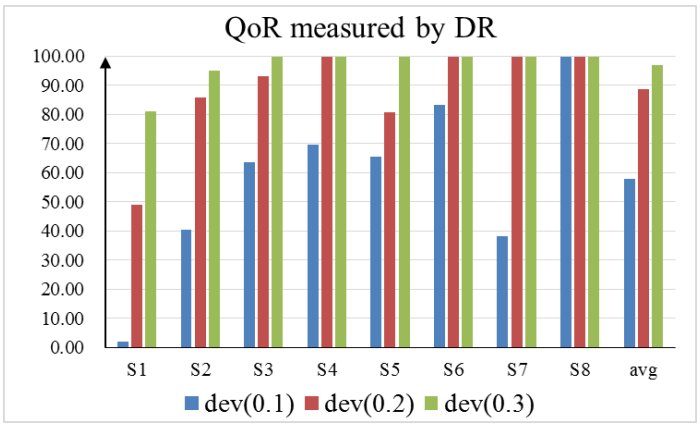
dev		∞	10%			20%			30%		
Bench	Task	Run	ADRS	DR	Run	ADRS	DR	Run	ADRS	DR	Run
S1	4	12	14.2	1.9	<1	5.1	49.1	<1	1.5	81.1	<1
S2	5	198	4.8	40.5	15	0.9	85.7	27	0.2	95.2	60
S3	6	672	1.3	63.6	47	0.2	93.2	119	0	100	271
S4	7	1601	1.5	69.6	121	0	100	331	0	100	914
S5	8	1378	1.6	65.4	7	0.6	80.8	295	0	100	1080
S6	8	97	0.7	83.3	5	0	100	43	0	100	54
S7	9	1439	1.9	38.1	8	0	100	810	0	100	1210
S8	10	28	0	100	28	0	100	28	0	100	28
Avg.		678	3.2	57.8	29	0.9	88.6	201	0.2	97	453

Table 5.2 shows the experimental results with 4 set of CO values in percentage of the generalized distance of reference set at each iteration. As mentioned in Chapter 5.2.3, CO value of infinity ∞ also means the exhausted research so that it provides the optimal solution set as reference set. The other three values are of 10%, 20% and 30%. All the values of *ADRS* and *DR* in Table

5.2 are measured in percentage and the running time value of label *Run* is measured in minutes. Thus, for the reference set ($dev = \infty$), the value of *ADRS* and *DR* should be always 0% and 100%, respectively, and of course, it should be the most time-consuming.



(a) QoR of ADRS



(b) QoR of DR

Fig. 5.3: Experimental results of macro-exploration: (a) QoR of ADRS; (b) QoR of DR

In particular, Fig.5.3 shows the results of Table 5.2in graph of (a) QoR of ADRS and (b) QoR of DR. It is visually found that with the value of CO increasing, QoR is getting better at the price of running time. Among three experiments, $dev = 10\%$ performs worst with maximum ADRS of value 14% and almost no domination solutions found DR at worst case, while $dev = 10\%$ provides a very good performance which could be accepted as optimal solution set but the running time is still significant. $dev = 20\%$ provides a good balance solution with QoR metrics of less than 1% to ADRS and almost 90% to DR in average and around $10\times$ speed-up.

One interesting issue revealed from the experimental results is that the running time does not always increase all the time as the system is getting more complex with more tasks involved. Look at the results of system benchmark of S8 with 10 tasks, it finishes very soon comparing to other benchmarks. One reason for it is that the proposed method and experiment are based on one shared bus with static scheduling, therefore once more tasks are involved, more data is required to transfer leading to heavy bus congestion and even bus saturation. Once bus is saturated, proposed scheduling fails marked as infeasible scheduling. Under this situation, proposed macro-exploration with static scheduling could find out an acceptable results effectively and the control offset CO adaptively control the QoR of exploration and running time.

5.4 Summary

This chapter proposes a complete adaptive exploration flow containing HW/SW partitioning, static scheduling and mapping targeting on re-configurable SoC architectures. It takes a set of independent tasks as input given in behavioral description and maps them onto a heterogeneous architecture with multi-core processor(s) coupling re-configurable HW accelerator connected through a shared bus. A control parameter called CO allows to flexibly generate solution with different QoR vs. running time.

6 Dynamic Schedule SoC Design Space Exploration

This chapter introduces a SoC design space exploration method for dynamic scheduled SoCs. Comparing to static schedule system in last chapter, system with standard bus architecture is explored with cycle-accurate model by optimizing timing diagram. Most commercial SoCs are build by stitching a set of IPs together through a bus or bus hierarchies. These allows companies to meet their tight schedules while focusing on the valued added parts of the SoC which differentiates their solution from others.

One of the most widely used on-chip bus is ARM's AMBA AHB/AXI buses. This bus has a typical master-slave architecture and makes use of an arbiter to determine which master can gain control over the bus and when each slave can return the data to the master. It is therefore important to analyze these dynamically scheduled systems and compare them with static scheduled ones, shown in the previous chapter.

One other aspect that this chapter will investigate is the optimization of BIPs mapped as slaves on these systems. When mapping a behavioral IP onto an MPSoC architecture, the problem for the designer is now to decide which micro-architecture for each behavioral IP is the best for a particular MPSoC configuration. It is therefore important to investigate methods to find smallest design which meets the performance constraints of a given MPSoC.

6.1 Motivations

Fig.6.1 shows the target MPSoC platform used throughout this chapter. Our MPSoC generator takes as inputs N BIPs in synthesizable ANSI-C or SystemC code for HLS and generates automatically different MPSoC configurations with M number of masters ranging from M=[1,N] and N slaves interconnected through an AMBA AHB bus. The BIPs are synthesized as slaves in the system, while the masters emulate processors executing different tasks.

[1] Intuitively having a system with a single master ($M=1$), is equivalent to a single processor which generates the data for all the slaves in the system. This configuration should lead to the slowest, but smallest system, while having a system with $M=N$, mapping each task onto individual masters, should lead to the fastest but largest system.

One of the uniqueness of this work is that it generates complete synthesizable C-based MPSoCs, by using a bus generator provided by the commercial HLS tool used in this work [45]. This allows our method to quickly generate new MPSoC configurations and simulate these to evaluate their performance using a cycle-accurate model generator included in the HLS tool as well.

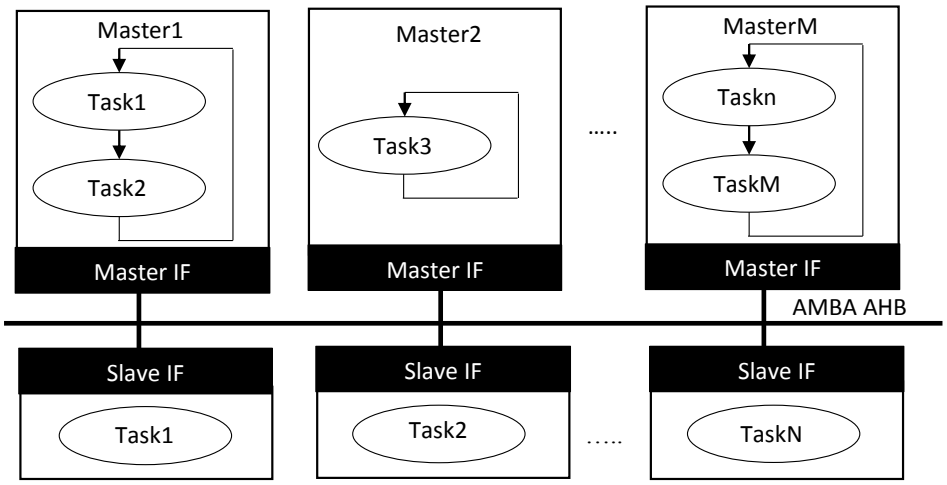


Fig. 6.1: MPSoC target platform

Fig.6.2 shows an example of multiple MPSoC configurations for 4 BIPs implemented as slaves on the MPSoC architecture described previously with one, two, three and four masters, M1/S4, M2/S4,M3/S4 and M4/S4 and Fig.6.2 (a), (b), (c) and (d) respectively. The area here only indicates the area of the slaves (excluding the masters' area). Each point on the diagram represents a unique task mapping within the specified number of masters.

The right side of Fig.6.2 also shows trade-off curves obtained for each BIP. Before our method is executed, a Design Space Exploration (DSE) for each BIP is performed as a pre-characterization step. The result of the DSE is a

[1] In this work the term master and processor will be used interchangeably to denote a system component which originates the data for the slaves and initiates the communication sequence. This work also makes use of the term slaves, BIP, HW kernel or HW accelerator interchangeably

trade-off curve with Pareto-optimal (dominating) designs with unique area vs. latency trade-offs for each BIP. One of the big advantages of BIPs over traditional RT-level IPs is that HLS allows the generation of micro-architectures with different area vs. performance trade-offs by only modifying the synthesis options. Thus, when choosing one design for each BIP from the trade-off curve, different mappings onto a system with the same number of masters will lead to systems with different performances, but same area. This is clearly shown in Fig.6.2(b) and (c), by a row of points with same area, but different throughput. Two special cases are the fastest designs of each of the BIPs, highlighted in their trade-off curves as a triangle and also the smallest ($D_{Mi}(small)$), highlighted as gray squares.

In all four cases, these designs lead to systems with highest throughput, but largest area (A_{max}) and systems of smallest area (A_{min}) but lowest throughput for a particular configuration. These designs corresponds to $D_{M1}(init)$, $D_{M2}(init)$, $D_{M3}(init)$ and $D_{P4}(init)$, for the single, dual, triple and quadruple master MPSoC configuration and $D_{P1}(small)$, $D_{P2}(small)$ and $D_{P3}(small)$, for the smallest configurations. The rest of the designs (black circles) represent other systems built from other designs taken from each of the BIPs' trade-off curves.

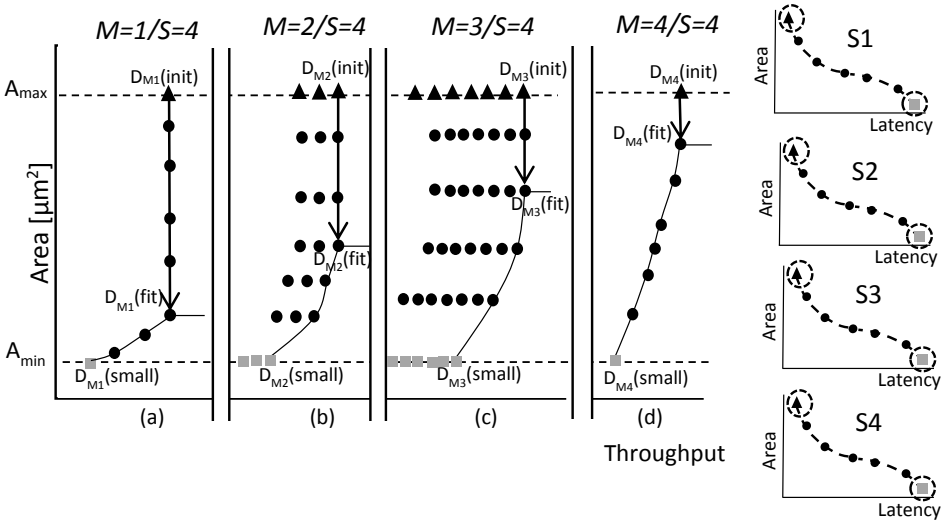


Fig. 6.2: MPSoC configurations example with 4 slaves over area vs. throughput: (a) 1 Master; (b) 2 Masters; (c) 3 Masters; (d) 4 Masters

Several observation can be made from these results:

Observation 1: Different task mappings for the same BIPs' implementations lead to different system performances, while consume the same area. Hence, there is a task mapping which dominates the others. This fact is mainly due to the fact that the master's cannot feed the slaves continuously with data due to bus congestion problems. The bus arbitration policy also affects this. In this work the bus arbiter is set in all cases to round robin arbitration.

Observation 2: Based on observation 1 it can be further observed that for each BIP, there are smaller designs, which can lead to the same performance of the entire system, while consuming less area than an equivalent system composed only by BIPs of highest performance and largest area. It is therefore not needed to fully parallelize the BIPs to achieve their highest performance. Hence a slower, but smallest version of these BIPs can be used in each of the MPSoC configurations. Designs $D_{M1}(fit)$, $D_{M2}(fit)$,$D_{M3}(fit)$ and $D_{M4}(fit)$ in Fig.6.2 show the smallest designs obtained after analyzing the amount of idle time of each slave. The smallest design for each BIP depends on the number of masters of the MPSoC and on the mapping of tasks on each master. It is therefore desirable to have an automated method which can generate automatically multiple MPSoC configurations and for each of them report the smallest design which will maximize performance.

Observation 3: The number of mappings follows the Stirling numbers of the second kind sequence. In this work we do not consider the task execution order once the tasks are mapped onto the same master. For the first case only a single task mapping is possible, because there is only a single master available. Similarly, only one task assignment is possible in the case that 4 masters are available as each tasks is mapped onto its own master. For the other two cases, there are 7 and 6 possible tasks mappings. The details of those mapping are displayed in Table 6.1 This will be explained in more detail in the next sections as this impacts the running time of our technique.

Table 6.1: Example for all possible tasks mapping with 4 tasks

PN	1	2	3	4
Combinations	{(1, 2, 3, 4)}	{(1), (2, 3, 4)} {(2), (1, 3, 4)} {(3), (1, 2, 4)} {(4), (1, 2, 3)} {(1, 2), (3, 4)} {(1, 3), (2, 4)} {(1, 4), (2, 3)}	{(1, 2), (3), (4)} {(1, 3), (2), (4)} {(1, 4), (2), (3)} {(1), (2, 3), (4)} {(1), (2, 4), (3)} {(1), (2), (3, 4)}	{(1), (2), (3), (4)}

Observation 4: From Fig.6.2 it can also be observed that the area savings are more pronounced for systems with less masters, as each accelerator (slave) has now to wait longer to receive and send data from and to the master.

The main tools that enable our work to identify the amount of performance degradation allowed by each HW kernel and the ability to generate smaller designs are: First the use of BIPs for each of the dedicated HW modules and (2) the ability to generate cycle-accurate models for the entire MPSoC to accurately estimate the idle time of each slave and the performance of the entire system. Other work make use of virtual platforms which model the communication part loosely through payloads. The problem with this approach is that the exact idle time of each HW modules cannot be accurately measured and hence previous work cannot exactly determine the idle time of each module. This combines with the fact that our method takes BIPs, which can be synthesized into different micro-architectures automatically, as inputs, and those are key differentiating elements in this work.

6.2 Proposed Exploration Method

The main aim of the work is to find a trade-off curve of Pareto-optimal systems with unique mappings and micro-architectures for each BIP to be mapped as a HWAcc on the system. The proposed method takes as inputs BIPs with design candidates pre-characterized in a HLS DSE are one of the inputs of our proposed method. A cycle-accurate model of the entire SoC is then used to estimate and measure the performance of the new system in order to find the optimal systems.

A complete flow diagram of the exploration method, called Optimization of Dynamic SoC (OPT_DSOC), is shown in Fig.6.3. BIP descriptions in synthesizable high-level language (e.g. ANSI-C or SystemC) as well as their corresponding testbenches are given as inputs to the flow. A system S contains M processors and N tasks to be mapped onto these processors, partitioned a priori into a synthesizable BIP, which is synthesized as a slave, and its testbench TB, which is always mapped onto the processor:

$$S = M + N$$

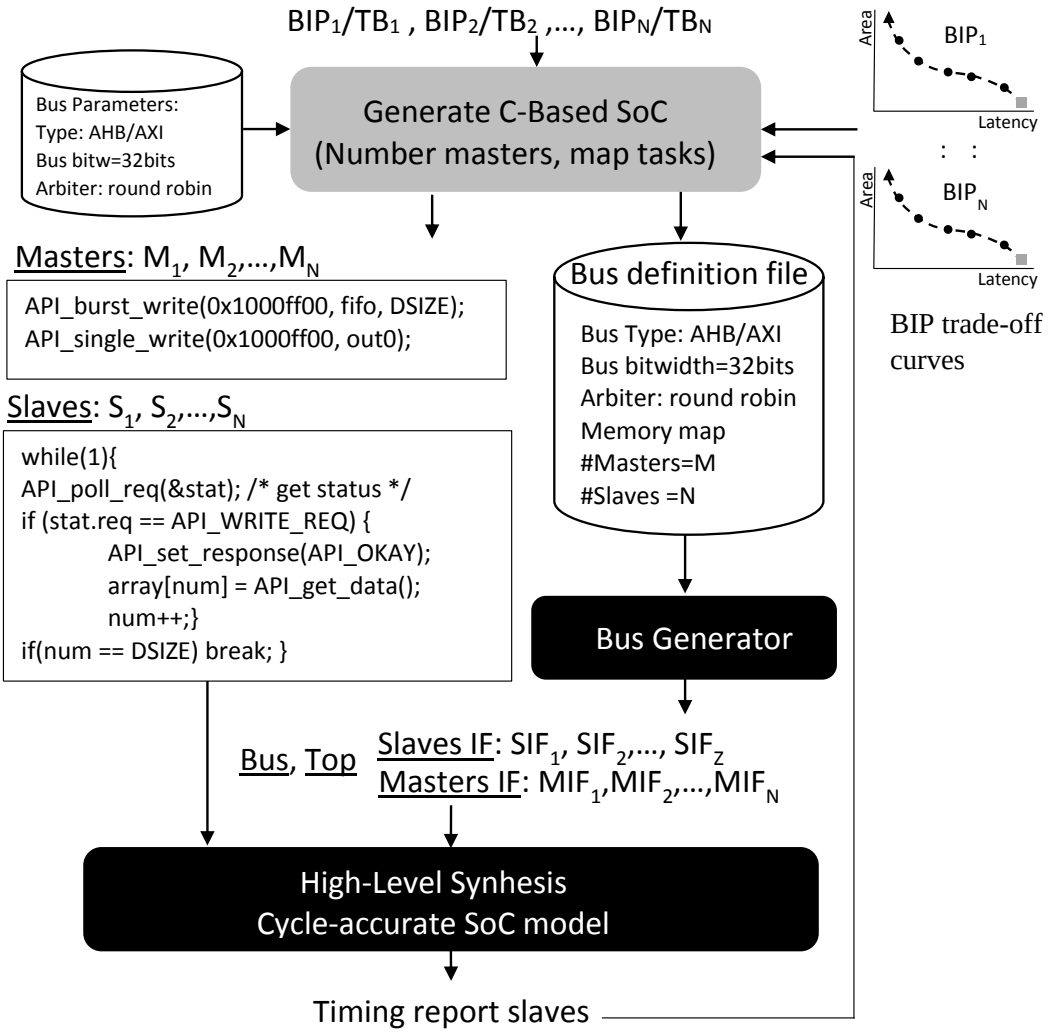


Fig. 6.3: Proposed method flow diagram

$$N = \{BIP_1, BIP_2, ..., BIP_N\}$$

$$M = \{TB_1, TB_2, ..., TB_N\}$$

In the system, the testbench is executed as master to produce data traffic (workload). In order to look at the effect of varying number of processors driving the HWAcc, the proposed method explores system architecture of different number of processors, ranging between $[1, N]$. OPT_DSOC is composed of 4 main steps and 1 pre-characterization. Algorithm 2 shows the a pseudo-code of the proposed method described in detail in the next subsections.

Pre-Step: BIP Design Space Exploration

Algorithm 2: MPSoC DSE algorithm

Input: $T_{list} = \{BIP_1/TB_1, ..., BIP_N/TB_N | BIP_n = \{D_1, ..., D_{M_n}\}\}$, M
T_{list}: Tasks list with size of N
BIP_n: N BIP dominating micro-architecture set with size of M_n
TB_n: N BIP simulation testbench
M: The number of processors involved in system architecture
Output: C_{Pareto}: Pareto-optimal solution list

```
/* Step 1:  architecture exploration                          */
S_list ← architecture_exploration();
S_list ← architecture_sorting(S_list);
C_Pareto ← S_list;
for (S ∈ S_list) do
    C_sys ← NULL;
    /* Step 2:  BIP optimization exploration                  */
    while (not_smallest_system(S) do
        if (is_BIP_optimized(S)) then
            S ← random_BIP_degrade(S);
        else
            S ← BIP_optimization(S);
        end
        cycle_accurate_simulation(S);
        C_sys ← C_sys + S;
    end
    /* Step 3:  Pareto-optimal update                         */
    C_old ← C_Pareto;
    C_Pareto ← Pareto_optimal_solution(C_old + C_sys);
    if (C_Pareto = C_old) then
        break;
    end
end
return C_Pareto;
```

As a pre-characterization step, HLS DSE is performed on each individual BIP to generate multiple micro-architectures in term of area vs. performance (i.e. latency here). Differently from the exploration in chapter 5, here the explorer exploits the effects on resource sharing. In resource sharing a single FU is shared among different operations in the source code by inserted multiplexers at its inputs and outputs. Initially the explorer starts by parsing each behavioral description and calling the HLS tool's resource allocator individually. The output of this step is a FU constraint file ($FCNT$) which contains the type and number of FUs needed in order to fully parallelize the behavioral description. The explorer continues by automatically reducing the number of FUs in the $FCNT$ file by a fixed rate Δ FU. Experimentally it was found that Δ FU=10%FU_{max} provides a good balance between running time and

exploration coverage. This method also bounds the search to $O(n)$, instead of having to generate all possible FU combinations. The explorer finishes when all FUs are set to a single FU, which should lead to the very smallest design. It is possible to use other exploration methods, which have also lead to good results [47]–[49], but in this work the explorer is an input to our system and is not one of the main contributions. The output of the explorer is a trade-off curve with dominating designs which are stored in a data base.

Step 1 :System Generation

This first step determines how many masters between 1 and N, the system should have and maps the different tasks to individual masters. Our method generates N number of SoCs consecutively (from 1 to N) and for each new configuration generates all possible mappings. The order in which the tasks are executed on each master is not considered as all tasks are completely independent from each other.

The number of mappings follow the Stirling numbers of the second kind sequence. The Stirling numbers of the second kind $S(n,k)$ count the ways to divide a set of n objects into k nonempty subsets. In our case $n = N$, and $k = [1, N]$ where N is equal to the total number of slaves (BIPs). Fig. 6.2 illustrated the effect of different tasks mappings on the area and overall system throughput as well as on the number of combinations. When the system only has 1 master ($M = 1$) only one mapping exists, which also leads to the slowest of all system configurations because the master now executes all the tasks. This case corresponds to $S(N,1) = 1$. By increasing the number of masters more tasks mapping combinations exists until $N/2$, which has the largest number of tasks mapping combinations ($S(N, N/2)$). Finally increasing the number of masters until $M = N$ leads again to a single task mapping as each tasks is mapped onto its own master,hence $S(N,N) = 1$. This configuration also typically leads to the fastest system. It should be noted that if the area of the masters is ignored, the total system area is virtually the same for all systems, as each system has the same number of slaves (although the bus complexity increases slightly with the number of masters and hence its area). In contrast, the performance will change with different mappings. The numbers of mappings in each case can be calculated as [50]:

$$S(n,k) = \frac{1}{k!}\sum_{i=0}^{k}(-1)^{k-i}\begin{pmatrix} k \\ i \end{pmatrix} i^n \tag{6.1}$$

where n is the number of slaves, which is always constant $= N$ and k is the number of masters (M)[51].

To generate valid mappings which can be simulated and synthesized, the original behavioral descriptions have to be modified to include a bus interface. For this purpose, commercial HLS tools provide a set of synthesizable APIs for different standard buses, i.e. AMBA's AHB and AXI. The tasks merged into a same master must write to the correct memory mapped slave, by calling the API with its assigned address, while the slaves *listen* until a master initiates the communication with them.

Fig.6.4 shows a snippet of these APIs for the master and the slave. The Masters can send data in burst mode or as individual data (when possible burst mode is chosen in this work), while the slaves wait for the masters to transmit the data. Normally, a complete communication iteration between the a master and a slave contains 5 steps:

1. master requests bus access for writing to the corresponding slave;

2. master writes data to corresponding slave when arbiter grants it access over the bus, otherwise it will wait until bus is free;

3. once slave receives data from master, it processes computation, and after finishing computation, it waits for master reading request;

4. master requests bus access for reading from the corresponding slave with unique address;

5. master reads data from corresponding slave if bus is idle, otherwise it may wait until bus is free or give up reading.

Because the entire system should be synthesizable, the testbenches should also be given in synthesizable C or SystemC code. The output is hence a list of synthesizable behavioral descriptions for the masters $MList = \{M_1, M_2, ..., M_P\}$ and for the slaves $SList = \{S_1, S_2, ..., S_N\}$.

This step also generates the bus definition file, which the bus generator in the next step takes as input in order to create a complete C-based SoC. This bus definition file includes: (1) arbiter protocol (fixed or round robin), (2)

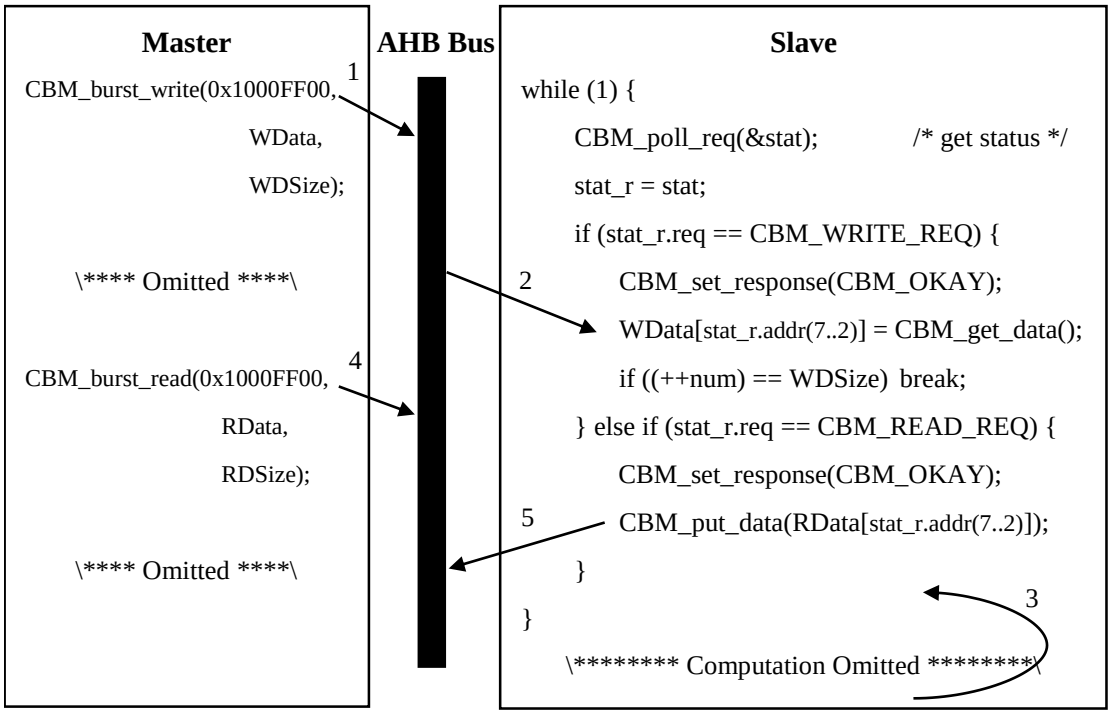


Fig. 6.4: APIs of read/write operations for master and slave

memory map, (3) number of masters and slaves, (4) bus type (AHB or AXI) and (5) bus bitwidth. By default the values for (1) (4) and (5) are set to round robin, AHB and 32-bits, but can be set to any other values externally. Fig.6.5 shows an example of this bus definition file, where Fig.6.5(a) describes the bus type, Fig.6.5(b) declares the the masters in the system and Fig.6.5(c) the slaves and their memory maps.

Step 2: Cycle-Accurate Model Generation

Once the system has been generated, each of the behavioral descriptions generated previously are synthesized. Because HLS is a single process synthesis method, each of them is synthesized individually with its own set of constraints. Once each of them are synthesized, a cycle-accurate model is generated in SystemC for the entire system. State of the art HLS tools also typically come with different model generators in order to verify the design at different levels of abstractions, e.g. behavioral-level (to verify the data type conversion) and cycle-accurate (to verify the timing). Once the system's cycle-accurate model is created, it is compiled using g++ and executed. All the BIPs used in this work were slightly modified to report the total time they remained in idle mode and the total time they were actively performing some computation. The results of

```
defbus AMBA_AHB {
    width address = 32;
    width data = 32;
    mode arbiter_rule = RoundRobin;
    module master =    {Master};
    module slave =     {Slave};
} ahbbus;
```

(a)

```
module AMBA_AHB_MASTER {
    mode burst = Enable;
    mode data_transfer = Direct;
    mode clock = Enable;
    mode reset = Enable;
} Master;
```

(b)

```
module AMBA_AHB_SLAVE {
    map address = 0x1000FF00-0x1000FFFF & 0xFFFFFF00;
    mode burst = Enable;
} Slave;
```

(c)

Fig. 6.5: Bus definition: (a) AHB BUS definition file example; (b) Master definition; (c) Slave definition

the execution is a timing report indicating the idle and computational time of each of the slaves.

Step 3: BIP Optimization

Based on the timing report obtained in the previous step, our method assigns to each slave a new micro-architecture from the trade-off curve generated for each BIP. Fig 6.6 graphically shows the report. It can be observed that at regular intervals the BIP receives the data from the master and computes it. It takes each BIP L_i cycles to finish the computation, where $L_i = \{L_{read} + L_{comp} + L_{write}\}$, with L_{read} the time required to read the data sent from the master, which is always constant once the communication has been established, L_{comp} the time taken to compute the new output and L_{write} the time taken to write the data back to the master. The only factor which

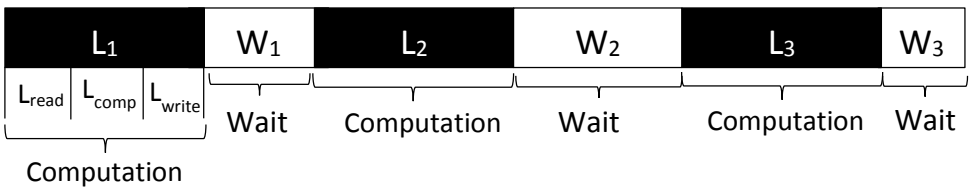


Fig. 6.6: One BIP execution schedule example

changes between two executions of the same task is L_{write}, as the slave has to retrieve the control over the bus, which changes between two executions. Moreover the main difference in the execution of the task is in the waiting time between two consecutive executions. Based on the number of other tasks being executed, their bandwidth required to send and receive data and the arbiter's priority, which in the round robin case keeps changing. In this case $W_3 < W_1 < W2$.

Our work considers these waiting cycles as positive slack, where the smallest waiting period (i.e. W_3) is the maximum slack. because the goal of our method is to find the smallest design which can sustain the same performance. This means that a micro-architecture with latency $L_{comp\_new} = floor(L_{comp} + W_{min})$ is chosen from the pre-characterized micro-architectural exploration trade-off curve and the BIP substituted. Because the dominating curve does not contain designs of all latencies the closes smallest value is chosen. This analysis is done for each of the slaves. Once all of the BIPs are substituted by their respective smallest designs, a new system is generated, re-synthesized and re-simulated to get accurate performance values. The same system choosing the smallest micro-architecture for each BIP is also generated as reference for each mapping in order to provide the user the range of systems that can be generated. In Fig.6.2 these smallest systems correspond to the $D_{Mi}(small)$, while the fastest smallest designs are represented by $D_{Mi}(fit)$.

Step 4: System Exploration

Until the previous step, the proposed method could optimize a single SoC configuration to find the smallest design for a given throughput (initially the system with maximum throughput). The method thus continues by finding all dominating designs on the exploration trade-off curve as shown in Fig.6.7. Once step for finding the smallest configuration (mapping and IP micro-architecture) for the design with highest throughput, the method continues by assigning to one of the slaves randomly a micro-architecture of worse performance that the current one, but also of smaller area. This will guarantee that a configuration of worse performance is now obtained, but also a smaller one.

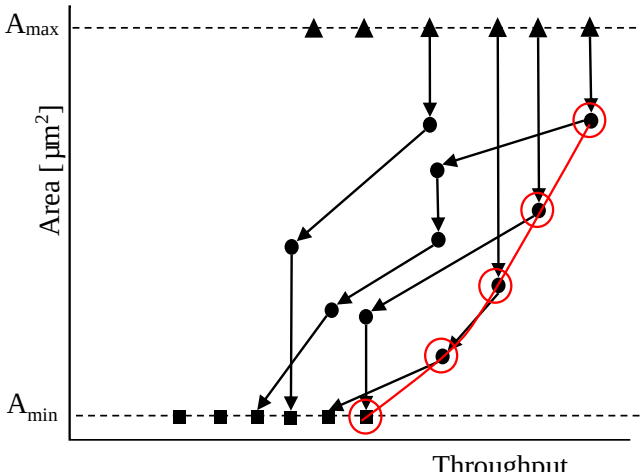


Fig. 6.7: Example of DSE with BIP optimization

6.3 Hybrid Exploration Method

As the search space for complex SoCs with larger number of HWAcc is so larger, apart from the proposed method,another method to avoid having to re-synthesize and re-simulate so many configurations was developed. The hybrid method is based on an analytical performance modelling method and greedy search.

According to the targeted MPSoC platform shown in Fig.6.1 shows, the tasks are executed consecutively and they are independent for each other. Also the operation frequency is constant and the arbitration policy in all cases was set to round-robin. Therefore a fitness function was analytically established as:

$$Fitness = \sum_{n=1}^{N} w_n Thp_n \tag{6.2}$$

$$Thp_n = \frac{IO_n}{L_n} \tag{6.3}$$

$$w_n = \frac{IO_n}{\sum_{m=1}^{M}(x_{nm}\sum_{i=1}^{N} x_{im} IO_i)} \tag{6.4}$$

with w_n the weighted throughput of each task and Thp_n the total system throughput. The throughput in Equation 6.3 is proportional to the actual throughput because the operation frequency is constant for each task. The weighted function is calculated from the throughput contributed by each of the specific tasks within the corresponding processor. If one processor has

many tasks assigned, the relative weight becomes smaller and thus the task performance degrades. In the other case that a processor only has a single tasks assigned to it, then the weight is 1. In this context $x_{nm} = 1$ indicates task n is mapped onto processor m, otherwise $x_{nm} = 0$. Thus, $\sum_{m=1}^{M} x_{nm} = 1$ for every n ranging between $[1, N]$.

Using this fitness function, the area and performance of the entire design space can be estimated without having to synthesize nor simulate anything. This method hence makes use of this analytical model to estimate the performance and area of every single configuration and extracts for each configuration (with unique number of processors) the estimated dominating designs.

After analytical estimation, a greedy search is applied iteratively. The estimated dominating designs are then synthesized and simulated in order to get the actual performance values. If the resultant configurations provide new dominated results among simulated trade-off curve, a new set of designs leading to an estimated Pareto-optimal design are chosen; otherwise iteration is finished and output final trade-off curve.

Although in theory an more accurate method, if fitness or cost function is well-defined ,than the previous, simulation based method, it has also some 3 main drawbacks: (1) the targeted architecture is fixed and should follow the one shown in Fig.6.1; (2) the tasks are considered independent for each others; (3) the communication is assumed as AHB bus with round-robin arbitration, while fixed arbitration systems could lead to inaccuracies as this method cannot handle priorities.

Nevertheless, this method and an exhaustive search can be used as reference methods to be compared in terms of running time an quality of results.

6.4 Experimental Results

Different computational intensive application, amenable to HW acceleration, were selected and grouped together into complex systems in order to test our proposed method. These designs were taken from the open source Synthesizable SystemC Benchmark suite (S2CBench) [44]. Table 6.2 shows how these complex benchmarks were formed. The first column indicates the

name of benchmark, the second column indicates the total number of dominating designs reported by the DSE for each benchmark. Columns S1-S8 indicate the number of instantiations of each test case used to build each complex benchmark. The last two rows report the total number of applications used in each system benchmark and total number of design candidates contained (adding up the results of the DSE of each application). The six BIP applications are: Kasumi of Block cipher used in mobile communication systems, MD5C of Message digest algorithm, Adpcm of adaptive differential pulse-code modulation, FIR of 9-tap FIR filter. Interp of 4-stage interpolation filter and qsort of quick sort algorithm.

The experiments were run on an Intel dual 2.40GHz Xeon processor with 16 GBytes of RAM running Linux Fedora release 19. The HLS tool tool used is CyberWorkBench (CWB) of NEC version of 5.6. The target architecture, as mentioned previously, is a multi-core processor system with as many masters as BIPs, with a 32-bit AMBA AHB bus using a round robin arbiter. The target technology is Nangate's 45nm Opencell technology aand the HLS target frequency for all of the processes in the system is set to 100MHz.

Table 6.2: Complex System Benchmarks

Bench	DSE	S1	S2	S3	S4	S5	S6	S7	S8
Kasumi	2		1		1	1	1		1
MD5C	4	1		1	1		1	1	1
Adpcm	3			1		1	1	1	1
FIR	7		1		1	1	1	1	1
Interp	8	1	1		1		1	1	1
qsort	4	1		1		1		1	1
BIP		3	3	3	4	4	5	5	6
Designs		16	15	11	21	16	24	26	28

Table 6.3 displays the experimental results comparing 3 methods: brute-force BF (exhausted search) method, analytic function $Analytic$ search and the proposed OPT_DSOC method. ADRS, average distance from reference solution, is used to measure the quality of result (QoR) of the obtained trade-off curve. The smaller ADRS is, the better QoR. Column 1, $Bench$, lists the system benchmark from Table 6.2 and column 2, BIP, shows the number of BIPs (tasks) involved in system. $ADRS$ is measured in percentage and running time Run in minutes. The last two rows indicates the average values of the results. The first row for S1 to S5 and the last row for S1 to S8. This helps

Table 6.3: Experimental Results

Bench	BIP	BF	Hybrid		OPT_DSOC	
		Run[min]	ADRS[%]	Run[min]	ADRS[%]	Run[min]
S1	3	192	0.66	82	2.20	32
S2	3	622	0.75	149	1.61	33
S3	3	297	0.64	106	3.23	40
S4	4	10518	0.44	340	3.55	118
S5	4	3472	0.57	229	0.34	84
S6	5	-	0.64	883	3.76	302
S7	5	-	0.65	1102	2.86	368
S8	6	-	0.85	3009	3.41	897
Avg.	4.125	3020	0.61	181	2.15	61
		-	0.65	738	2.62	234

comparing the average results, first isolating the case when the BF method was able to return a result and all the other cases. As the system get larger, $S6$-$S7$, brute-force method cannot be finished and there is no global optimal solution. Therefore the ADRS for these system is obtained from reference solution combining all results from different methods.

No ADRS results are shown for the BF method as it could always find the optimal solution. Hence the ADRS is 0% for the cases which it could finish the exploration. For S6 to S8 it could not finish as shown in the results table. According to the results of $Hybrid$, is very good as it provide results with similar quality to the BF (<1%) for S1-S5. On the other hand, our proposed method OPT_DSOC lead to result within 4%, which worse than the of 1% obtained from $Hybrid$, while the running is about $3\times$ faster than $Hybrid$. As mentioned previously, the drawback of the analytic model based on equations 6.2, 6.3, 6.4 is that a specific system architecture can only be target and hence it is not very flexible. On the other hand OPT_DSOC is much more flexible and can work with any system architecture, bus specification and different clock frequencies

6.5 Summary

This chapter presents a design space exploration method with dynamic schedule, based on maximizing the efficiency of each BIP mapped as a HWacc on a shared bus MPSoC system. In particular, the advanced features available in state of the art HLS tool allow the generation of complete MPSoC system in

C and simulation with cycle-accurate model. As BIPs often have to wait for data to be transferred and permission to access bus, it is unnecessary to maximum single BIP performance using more HW resources. The area can be optimized with the help of cycle-accurate simulations. Such optimization provides a efficient way to explore the search space. The results obtained are within reason compared to an exhaustive search. An analytic model method was also proposed in this chapter. It avoids having to re-synthesize and re-simulate new systems and has been shown to be quite accurate. The drawback is that the model only works for the master and slave system presented here, while the simulation based method is more generic and can be used for different types of systems (e.g. different bus structures).

7 Results Discussion

The previous chapters presented efficient methods to explore C-based SoCs for static and dynamic scheduled systems.

The developed methods have proven to lead to good results compared to exhaustive search methods which lead to the optimal solution. Although in chapter 5 and chapter 6, the HW and SW partition was given before-hand an automatic HW/SW partitioning method was developed and presented in chapter 4. This chapter also studied the effect of bus congestions on speed-ups. The main emphasis overall in this thesis, was in the practicality of the proposed methods as the optimization methods were all build on top of commercial SW and EDA tools.

The work in chapter 4 also helped extracting the most computationally intensive kernels of the main applications used in the following chapters. The granularity of each kernel of the partitioning method presented in this work is a function and hence is decided by the programming style in each application (also called coarse-grained partitioning). Other approaches use task graphs as inputs and can therefore create more optimal (finer) partitions (also called fine-grained partitioning). In this work this cannot be done because we use a commercial HLS tool during the pre-characterization stage. This allows our method to get very accurate area and timing results, but has the drawback of lack of internal controllability. The main results that can be extracted from this chapters is that bus congestion is a serious problem when accelerating computationally intensive tasks using HWAccs mapped on FPGA-based co-processors. As Moore's law continues, FPGAs have larger logic densities and allow complete systems to be mapped on them with minimal off-chip resources. This also means that multiple HWaccs can be now mapped onto the same FPGAs to speed-up multiple applications simultaneously. The main bottleneck as shown in this chapter is the communication overhead. It is therefore extremely important to consider the bandwidths required by each of these HWAccs mapped to the FPGA.

The next two chapters (chapter 5 and chapter 6) showed the importance of the scheduling and task mapping on these type of systems at the chip level.

In chapter 5 a static (off-line) task scheduling method was developed to study the generation of SoCs with unique area and performance. Because each HWAcc (BIP) mapped onto the system can have a variety of micro-architectures of different area and performance, SoC composed of these BIPs will also have different characteristics. Off-line algorithms have shown to be able to obtain superior results. These static methods to schedule multiple applications on a system can be used in order to reduce the complexity of the HW and hence reduce its area and power overheads as the bus structure is simplified (e.g. no arbiter is required). The drawback of these systems, are the lack of flexibility. The system might be required to be completely re-generated in case of any minor change. For dedicated HWAcc systems with very predictable access patterns this might not be a big issue, but with embedded systems with dynamic response behavior, these systems might not be adequate. The exploration method presented in this work, could find the optimal solution by using dynamic programming based on a tree structure with dynamic pruning technique. Additionally, the running time vs. the quality of results could be controlled be a single parameter called the Control Offset (CO).

Chapter 6 presented the results of the SoC exploration for dynamic scheduled systems based on a widely used on-chip bus standard: AMB AHB. The results showed the importance of tasks mapping and scheduling on the performance of the complete system. It was also observed that because most of the HWAcc have to wait for data to reach them and also to write data back to the master, they can be implemented using less logic resources and hence requiring less area and power without any performance loss.

The results obtained in this chapter were possible because of the advanced system-level design features available on the commercial HLS tool used in this work. In particular the bus generator which allows to create complete C-based SoCs and the cycle-accurate model generator, which allowed the generation of fast cycle-accurate system models. Although faster than RTL simulations, these complete system-level simulations are computationally very intensive as shown in Table 6.3.

Because the complex SoCs created in each of these chapters are different, it is not easy to use the results obtained to compare them directly. Hence Table 7.1 shows 4 different system created using 6 tasks from the S2CBench benchmarks explored using the static and the dynamic scheduling architectures used in the previous chapters. In both cases the number of masters is equal to the number of tasks. By creating the exact same designs it is possible to verify if some of the claims made by previous works regarding the benefits of each of this architectures is true.

The Design row in the table shows that many more unique configurations could be generated using the static scheduling architecture, showing that the search space is larger for static systems. In terms of area the static scheduled system is, as expected smaller, because it does not have any bus arbiter and the connection interfaces are simpler. Finally in terms of performance (throughput), the results are mixed. In most of cases, dynamic schedules lead to more accurate results as cycle-accurate model is used and larger area with arbiter and interfaces.

Table 7.1: Comparison with Static Schedule vs. Dynamic Schedule

Bench		S1	S2	S3	S4
Kasumi			1		1
MD5C		1	1	1	1
Adpcm				1	1
FIR			1	1	1
Interp		1	1	1	1
qsort		1		1	1
Task		3	4	5	6
Static	Design	83	64	58	52
	Area	68399	64931	86069	94028
	Throughput	448.8	1943.2	1645.9	2227.1
Dynamic	Design	3.7	5.3	5.2	4.7
	Area	71497	65419	90498	100433
	Throughput	1246.5	1759.8	1920.5	2430.4

According to the experiments of system exploration with static and dynamic schedule, there exist several problem. One is the dependency issue and the other is scaling issue. The interconnection among benchmarks is a big issue of a system but experiments in this thesis only consider the independent case. System scaling is one metric to measure proposed algorithm.

For the exploration with static schedule in Chapter 5, a self-defined model is used. For this model, dependency issue is not considered. With the size of system increasing, the chance of infeasible solution increase too till there is not expected result as we consider bus saturation is infeasible. Thus in order to consider dependency and scaling issue with static schedule a new model should be defined. For the dynamic schedule in Chapter 6, the dependency could be included as cycle-accurate model is used for simulation. Mapping decision and design combination are two factors affecting scaling. In this thesis design combination is reduced by OIP optimization. More works could be done, such as heuristics on mapping decision and using templates replacing the actual slaves, to overcome scaling problem.

In conclusion, method to explore C-based SoC designs were presented in this thesis. In particular off-line an online methods. Results show that our methods work well and lead to good results in a reasonable time.

8 Conclusions and Future Work

This section concludes the work done in this thesis and addresses future work directions.

8.1 Conclusions

This thesis focuses on the exploration and optimization of C-based SoCs using advanced system-level design tools included in state of the art HLS tools. The emphasis of this work is on the exploration and optimization of these systems.

The main enabler of this work is HLS, which can be defined as the process of converting behavioral descriptions (in this work ANSI-C) into RTL descriptions which can effectively execute these. Chapter 3 introduced the main steps required to synthesize behavioral descriptions into RTL.

SoCs are currently being designed in a mixture of top-down and bottom-up approach. The first steps involve the virtual prototyping of the entire system in order to get preliminary performance, power and area estimates. This allows design teams to figure out the best overall architecture. The next step involves designing each of the individual blocks separately.

These SoCs are being created from a combination of legacy Register Transfer Level (RTL) blocks, Intellectual Properties (IPs), newly developed RTL and C/SystemC descriptions synthesized using behavioral synthesis. This re-use of existing legacy code allows design teams to focus only on the new features that need to be implemented, thus reducing time to market considerably. Nevertheless, the fact that each individual process is optimized separately means that global system-level optimization is neglected and that not the most efficient architecture is created. In particular, it is common practice not to modify any hardware (HW) block that has been fully verified, even if a more efficient architecture could be achieved in subsequent designs, due to the cost of having to re-design and especially re-verify the new implementation.

With the advent of new design methodologies and EDA tools, which further raise the level of abstraction, it is now possible to develop entire complex SoCs using only behavioral descriptions. These work makes use of these tools and show that it is easier, faster and more convenient to design these system a the C-level. Most of the work done in this thesis would not have been possible at the RT-level.

Although it will still take time until the industry fully adopts C-based VLSI design to design complete systems, the adoption has started at different levels in almost all companies and will only further continue.

8.2 Future Work

Future work includes the exploration of systems with multiple-clock domains/frequencies. Complex SoC now make aggressive use of DVFS (Dynamic Voltage and Frequency Scaling) to reduce the power consumption. In this work, the clock is assumed to be constant and equal for all the components in the SoC.

Moreover, all the tasks in this work are consider independent of each. In more complex systems, inter-dependencies between tasks should also be considered.

Finally, the results could be evaluated on real silicon by prototyping them on configurable SoCs, e.g. Altera's Cyclone V SoC or Xilinx's Zynq FPGA. It would be interesting to compare the simulation based results with the prototyped ones.

References

[1] I. T. R. for Semiconductors, "Www.public.itrs.net/links/2013itrs/2013chapters", 2013.

[2] J. Teich, "Hardware/software codesign: The past, the present, and predicting the future", *Proceedings of the IEEE*, vol. 100, no. Special Centennial Issue, pp. 1411–1430, 2012.

[3] M. López-Vallejo and J. C. López, "On the hardware-software partitioning problem: System modeling and partitioning techniques", *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 8, no. 3, pp. 269–297, 2003.

[4] R. Niemann and P. Marwedel, "An algorithm for hardware/software partitioning using mixed integer linear programming", *Design Automation for Embedded Systems*, vol. 2, no. 2, pp. 165–193, 1997.

[5] R. K. Gupta and G. De Micheli, "Hardware-software cosynthesis for digital systems", *Design & Test of Computers, IEEE*, vol. 10, no. 3, pp. 29–41, 1993.

[6] R. Ernst, J. Henkel, and T. Benner, "Hardware-software cosynthesis for microcontrollers", *Readings in hardware/software co-design*, pp. 18–29, 2002.

[7] D. Hendry and D. Sananikone, "Hardware/software partitioning of embedded systems with multiple hardware processes", *IEE Proceedings-Computers and Digital Techniques*, vol. 144, no. 5, pp. 285–294, 1997.

[8] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli, "Hardware/software partitioning with iterative improvement heuristics", in *Proceedings of the 9th international symposium on System synthesis*, IEEE Computer Society, 1996, p. 71.

[9] J. Harkin, T. M. McGinnity, and L. P. Maguire, "Hardware-software partitioning: A reconfigurable and evolutionary computing approach", in *Field-Programmable Logic and Applications*, Springer, 2001, pp. 595–600.

[10] J. T. Olson, J. W. Rozenblit, C. Talarico, and W. Jacak, "Hardware/software partitioning using bayesian belief networks", *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, vol. 37, no. 5, pp. 655–668, 2007.

[11] J. Noguera and R. M. Badia, “A hw/sw partitioning algorithm for dynamically reconfigurable architectures”, in *Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings*, IEEE, 2001, pp. 729–734.

[12] I. Ouaiss, S. Govindarajan, V. Srinivasan, M. Kaul, and R. Vemuri, “An integrated partitioning and synthesis system for dynamically reconfigurable multi-fpga architectures”, in *Parallel and Distributed Processing*, Springer, 1998, pp. 31–36.

[13] O. Lindtjorn, R. Clapp, O. Pell, H. Fu, M. Flynn, and O. Mencer, “Beyond traditional microprocessors for geoscience high-performance computing applications”, *Ieee Micro*, no. 2, pp. 41–49, 2011.

[14] C. Liem, F. Naçabal, C. Valderrama, P. Paulin, and A. Jerraya, “System-on-a-chip cosimulation and compilation”, *IEEE Design & Test of Computers*, no. 2, pp. 16–25, 1997.

[15] V. Živojnovic and H. Meyr, “Compiled hw/sw co-simulation”, in *Proceedings of the 33rd annual Design Automation Conference*, ACM, 1996, pp. 690–695.

[16] D. Bernstein, M. Rodeh, and I. Gertner, “On the complexity of scheduling problems for parallel/pipelined machines”, *Computers, IEEE Transactions on*, vol. 38, no. 9, pp. 1308–1313, 1989.

[17] T. Wiangtong, P. Y. Cheung, and W. Luk, “Comparing three heuristic search methods for functional partitioning in hardware–software codesign”, *Design Automation for Embedded Systems*, vol. 6, no. 4, pp. 425–449, 2002.

[18] E. A. Lee and D. G. Messerschmitt, “Static scheduling of synchronous data flow programs for digital signal processing”, *Computers, IEEE Transactions on*, vol. 100, no. 1, pp. 24–35, 1987.

[19] F. Ferrandi, P. L. Lanzi, C. Pilato, D. Sciuto, and A. Tumeo, “Ant colony heuristic for mapping and scheduling tasks and communications on heterogeneous embedded systems”, *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 29, no. 6, pp. 911–924, 2010.

[20] J.-J. Hwang, Y.-C. Chow, F. D. Anger, and C.-Y. Lee, “Scheduling precedence graphs in systems with interprocessor communication times”, *SIAM Journal on Computing*, vol. 18, no. 2, pp. 244–257, 1989.

[21] G. C. Sih, E. Lee, *et al.*, “A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures”, *Parallel and Distributed Systems, IEEE Transactions on*, vol. 4, no. 2, pp. 175–187, 1993.

[22] M. Bauer, J. Bosko, and E. Rogstad, “A dynamic approach to thread scheduling in hardware (dash)”,

[23] T. Hagras and J Janeček, “Static vs. dynamic list-scheduling performance comparison”, *Acta Polytechnica*, vol. 43, no. 6, 2003.

[24] Y. Markovskiy, E. Caspi, R. Huang, J. Yeh, M. Chu, J. Wawrzynek, and A. DeHon, "Analysis of quasi-static scheduling techniques in a virtualized reconfigurable machine", in *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, ACM, 2002, pp. 196–205.

[25] K. Lahiri, A. Raghunathan, and S. Dey, "Design space exploration for optimizing on-chip communication architectures", *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 23, no. 6, pp. 952–961, 2004.

[26] T. Givargis, F. Vahid, and J. Henkel, "System-level exploration for pareto-optimal configurations in parameterized system-on-a-chip", *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 10, no. 4, pp. 416–422, 2002.

[27] A. Gerstlauer, C. Haubelt, A. D. Pimentel, T. P. Stefanov, D. D. Gajski, and J. Teich, "Electronic system-level synthesis methodologies", *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, no. 10, pp. 1517–1530, 2009.

[28] C. Erbas, S. Cerav-Erbas, and A. D. Pimentel, "Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design", *Evolutionary Computation, IEEE Transactions on*, vol. 10, no. 3, pp. 358–374, 2006.

[29] F. Bellard, "Qemu, a fast and portable dynamic translator", in *USENIX*, 2005, pp. 41–41.

[30] T. Austin, E. Larson, and D. Ernst, "Simplescalar: an infrastructure for computer system modeling", *Computer*, vol. 35, no. 2, pp. 59–67, 2002.

[31] OVP. (2015), [Online]. Available: `www.ovpworld.org`.

[32] M. Lis *et al.*, "Scalable, accurate multicore simulation in the 1000-core era", in *ISPASS*, 2011, pp. 175–185.

[33] Y. Corre, V.-T. Hoang, J.-P. Diguet, D. Heller, and L. Lagadec, "Hls-based fast design space exploration of ad hoc hardware accelerators: a key tool for mpsoc synthesis on fpga", in *DASIP*, IEEE, 2012, pp. 1–8.

[34] S. Gheorghita *et al.*, "System-scenario-based design of dynamic embedded systems", *ACM Trans. Des. Autom. Electron. Syst.*, vol. 14, no. 1, 3:1–3:45, Jan. 2009.

[35] P. van Stralen and A. Pimentel, "Scenario-based design space exploration of mpsocs", in *ICCD*, 2010, pp. 305–312.

[36] K. Wakabayashi and B. C. Schafer, ""all-in-c" behavioral synthesis and verification with cyberworkbench", in *High-Level Synthesis*, Springer, 2008, pp. 113–127.

[37] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis", *IEEE Design & Test of Computers*, no. 4, pp. 8–17, 2009.

[38] C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu, "A formal approach to the scheduling problem in high level synthesis", *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 10, no. 4, pp. 464–475, 1991.

[39] S. Govindarajan, "Scheduling algorithms for high-level synthesis", *Term paper ECE*, vol. 834, 1995.

[40] Z. Baruch, "Scheduling algorithms for high-level synthesis", *ACAM Scientific Journal*, vol. 5, no. 1-2, pp. 48–57, 1996.

[41] A. Canis, J. Choi, B. Fort, R. Lian, Q. Huang, N. Calagar, M. Gort, J. J. Qin, M. Aldham, T. Czajkowski, *et al.*, "From software to accelerators with legup high-level synthesis", in *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, IEEE Press, 2013, p. 18.

[42] M. Lin and Y. Ma, "K-server optimal task scheduling problem with convex cost function", in *Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks, 2005. WIOPT 2005. Third International Symposium on*, IEEE, 2005, pp. 345–350.

[43] S. Martello, D. Pisinger, and P. Toth, "Dynamic programming and strong bounds for the 0-1 knapsack problem", *Management Science*, vol. 45, no. 3, pp. 414–424, 1999.

[44] B. C. Schafer and A. Mahapatra, "S2cbench: Synthesizable systemc benchmark suite for high-level synthesis", *Embedded Systems Letters, IEEE*, vol. 6, no. 3, pp. 53–56, 2014.

[45] NEC. (2015). Cyberworkbench, [Online]. Available: `www.cyberworkbench.com`.

[46] E. Zitzler, L. Thiele, M. Laumanns, C. M. Fonseca, and V. G. Da Fonseca, "Performance assessment of multiobjective optimizers: An analysis and review", *Evolutionary Computation, IEEE Transactions on*, vol. 7, no. 2, pp. 117–132, 2003.

[47] B. C. Schafer and K. Wakabayashi, "Divide and conquer high-level synthesis design space exploration", *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 17, no. 3, p. 29, 2012.

[48] H.-Y. Liu and L. P. Carloni, "On learning-based methods for design-space exploration with high-level synthesis", in *Proceedings of the 50th Annual Design Automation Conference*, ACM, 2013, p. 50.

[49] A. Mahapatra and B. C. Schafer, “Machine-learning based simulated annealer method for high level synthesis design space exploration”, in *Electronic System Level Synthesis Conference (ESLsyn), Proceedings of the 2014*, IEEE, 2014, pp. 1–6.

[50] H. Sharp, “Cardinality of finiate topologies”, *Journal of Combinational Theory*, vol. 5, no. 1, pp. 82–86, 1968.

[51] D. E. Knuth, R. L. Graham, O. Patashnik, *et al.*, “Concrete mathematics”, *Adison Wesley,*, 1989.