Contribution à une approche de modélisation et à un flot d'exploration destinés à des architectures MPSoC hétérogènes basées sur des processeurs configurables

H. Shen

INSTITUT POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque
978-2-84813-129-30

T H È S E

pour obtenir le grade de

DOCTEUR DE L'Institut polytechnique de Grenoble

Spécialité : Micro et Nano Électronique

préparée au laboratoire **TIMA**

dans le cadre de **l'École Doctorale d'Électronique, Électrotechnique, Automatique et Traitement du Signal**

présentée et soutenue publiquement

par

Hao SHEN

Le 11 Mars 2009

Contribution to a modeling approach and an exploration flow targeted to MPSoC architectures based on configurable processors

Directeur de thèse : Frédéric PÉTROT

JURY

M. François CHAROT	Président
M. Daniel ETIEMBLE	Rapporteur
M. Pascal SAINRAT	Rapporteur
M. Frédéric PÉTROT	Directeur de Thése
M. Rainer LEUPERS	Examinateur

Résumé

Dans le domaine de l'électronique pour la consommation de masse, les concepteurs sont tenus de fournir des systèmes embarqués qui doivent satisfaire des exigences de performance, de consommation, de coût et de temps de mise sur le marché. Pour satisfaire toutes ces exigences, nous nous concentrons sur les systèmes sur puce multi-processeurs (MP-SoCs) à base de processeurs configurables. Dans cette thèse, les architectures hétérogènes sont définies comme des architectures multiprocesseur à base de processeurs qui sont basées sur le même jeu d'instructions avec des extensions différentes. Cette thèse tente de résoudre certaines des difficultés causées par l'utilisation de processeurs configurables et les architectures hétérogènes. Dans de telles architectures, le champ des solutions d'implémentation devient extrêmement large et inclut aussi bien des optimisations logicielles que des optimisations matérielles. C'est pourquoi nous présentons 4 niveaux d'abstraction différents avec des niveaux de détail et des vitesses de simulation différentes pour faciliter l'exploration des différentes solutions d'implémentation. Une méthode de simulation hybride est également intégrée à ces niveaux d'abstraction pour éviter les efforts d'adaptation du logiciel dépendant du matériel durant l'exploration. Pour que l'implémentation choisie soit hautement performante et flexible, nous proposons un schéma de migration de tâches dans lequel une tâche peut être exécutée sur plusieurs processeurs compatibles avec différentes extensions d'instructions. Une application décodeur Motion-JPEG a été utilisée pour valider ces travaux.

Mots clés: Système sur Puce, multi-processeurs, processeurs configurables, hétérogénéité, niveaux d'abstraction, budget, flot d'exploration des solutions d'implémentations, simulation hybride et migration de tâches.

Abstract

In the consumer electronics domain, we propose to use the multiprocessor system-on-chip solution with configurable processors and heterogeneous architectures to satisfy all the requirements of performance, power consumption, cost and time-to-market. In this thesis, the heterogeneous architectures are defined as a group of processors which are based on the same core instruction set with different extensions. Because of the configurability and the heterogeneousness, the design space becomes extremely large and includes both software and hardware optimizations. To solve this problem, we build a design space exploration flow using 4 abstraction levels with different details and simulation speed. As we find that the *Hardware-dependent Software* **(HdS)** is the bottleneck in this flow, a hybrid simulation method is introduced to avoid these HdS adaptation efforts. To give realization high performance and flexibility, we propose one task migration framework in which one task can be executed on several compatible processors with different extended instructions. A Motion-JPEG case study is used to validate all these works.

Key words: System-on-Chip, multiple processors, configurable processors, heterogeneousness, abstraction levels, budget, design space exploration flow, hybrid simulation and task migration.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the advance of micro-electronics, in the consumer electronics domain, designers are demanded to provide embedded system solutions which should satisfy the following 4 requirements:

Performance: It means that a successful product should meet performance requirements for all supported applications. With the evolution of latest video compression/decompression algorithms such as MPEG 4 [26] and H.264 [4], high-speed wireless communication protocols [57] and 3D video games [53], performance requirements of these embedded systems even exceed abilities of most desktop computers. Fig.1.1 shows the performance requirement grows with the Moore's law [79]. To satisfy the huge performance requirement, the gap of Fig.1.1 can potentially be solved by increasing the number of general processors and application specific *Processing Elements* **(PEs)** [1].

Power Consumption: It means that embedded systems should provide enough available time with limited battery power supply. Until now, almost all portable devices such as smart mobile phones, MP3 players and video cameras rely on batteries for power supply. Because the growth of battery technology is much slower than Moore law of the micro-electronic domain, the power consumption is one of key constraints for the portable devices development in the future. Even for other embedded systems such as TV-boxes and 3D video games consoles [53] with alternating current power supply, the power consumption reduction can help avoid noise of coolers and improve reliability. Fig.1.2 shows that the trend of power consumption has to continuously growing in the next ten years. The *International Technology Roadmap for Semiconductors* **(ITRS)** report [1] provides several solutions for SoC consumer portable including architecture optimization at high-level design stages based upon power consumption analysis, and MPSoC realization with customized PEs.

Cost: It means that designers should provide competitive and profitable products for the embedded system market. With the ITRS prediction [2], the mask cost will increase almost 20 times in the next ten years (to 2018). Meanwhile, the software development fee grows even faster than that of the hardware development. To continue getting profits with this trend and reuse the same mask for different applications, *Application-specific integrated circuit* **(ASIC)** based design methodologies are gradually replaced by the *System-on-Chip* **(SoC)** based ones. By using the SoC design methodologies, the architectures with multiple

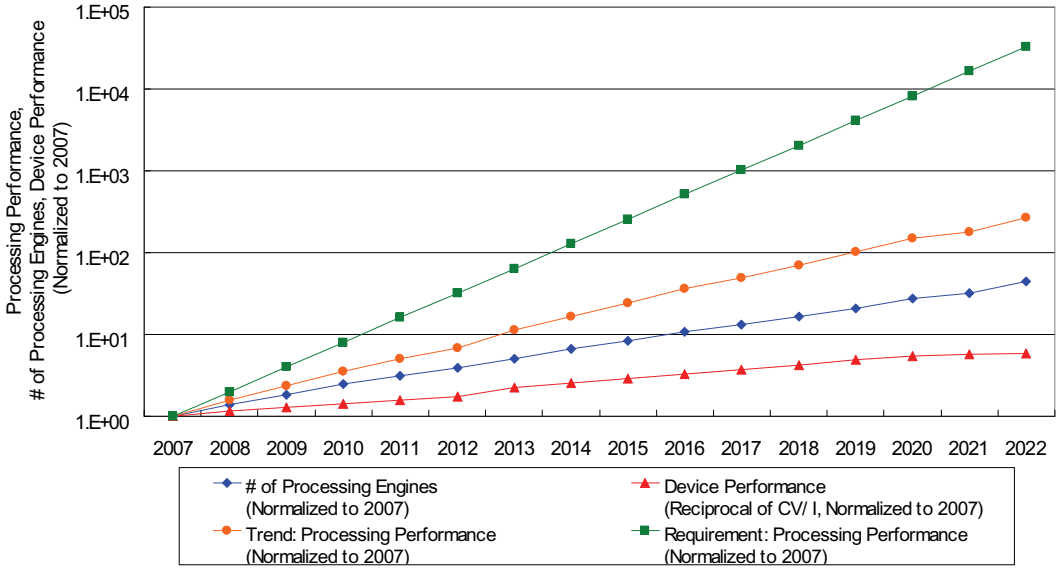

Figure 1.1: SoC Consumer Portable Processing Performance Trends from ITRS 2007 [1]

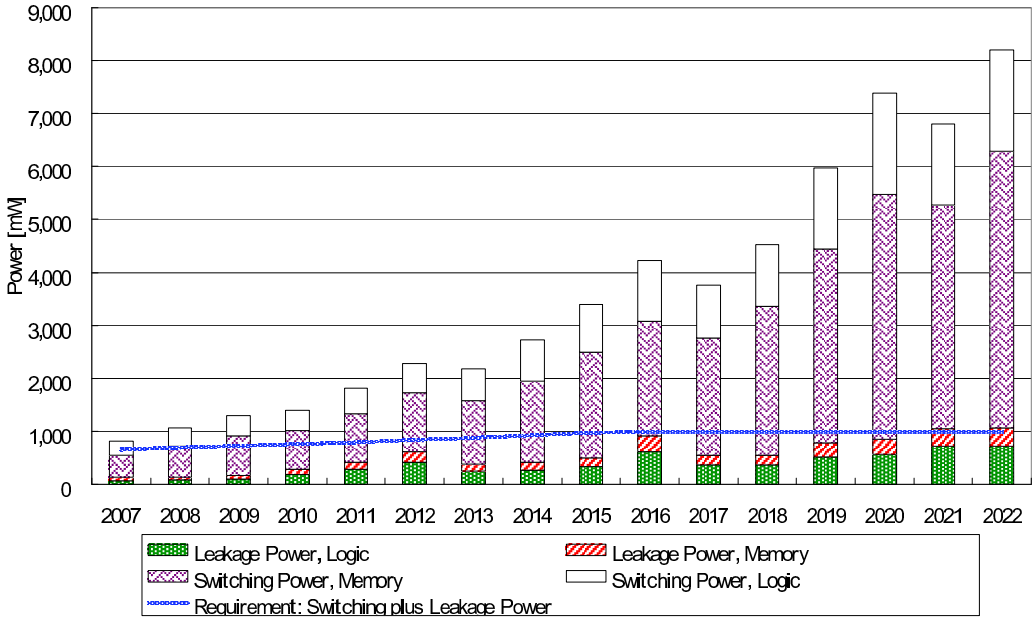

Figure 1.2: SoC Consumer Portable Power Consumption Trends from ITRS 2007 [1]

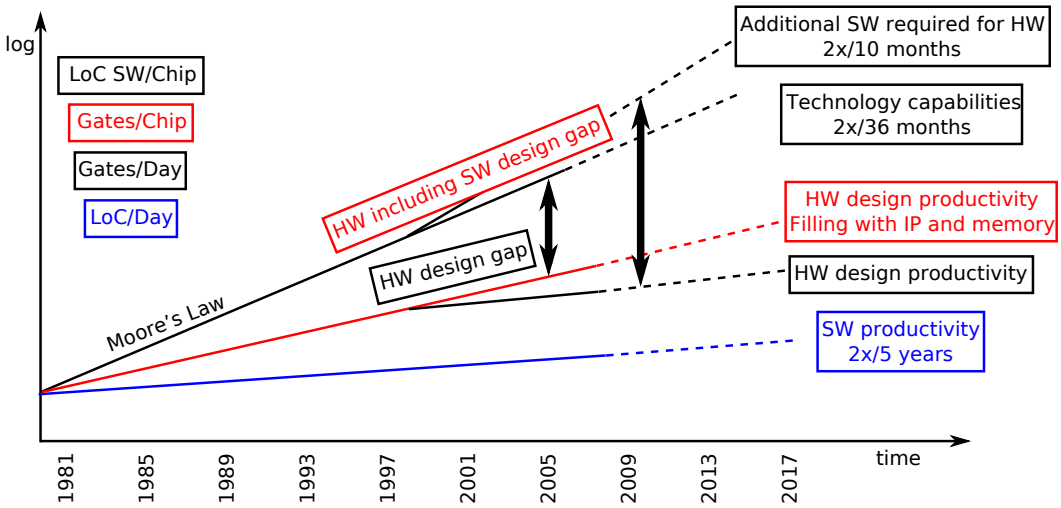

Figure 1.3: Hardware and Software Design Gaps Versus Time from ITRS 2007 [2]

customized processing units can provide more flexible and reusable solutions for a group of similar applications.

Time-to-market: It means that designers should shorten as much development time as possible for one product to success in the market. Fig.1.3 shows huge gaps between the Moore's law for the integrated circuits growth and the productivity of the hardware/software development. In this figure, these productivity gaps make the final product very hard to keep up with the Moore's law and the market requirements even by increasing development resources. Reusable design platforms, flexible hardware architectures and system level design methodology could be possible solutions to shorten both design and verification time to meet the time-to-market requirement.

These 4 requirement trends existed more than ten years ago and drove the development of the SoC design methodology. SoC integrates technology and design elements from other system driver classes into a wide range of high-complexity, high-value semiconductor products. In SoC design, the goal is to maximize the reuse of existing mature and verified blocks to shorten the design phase and protect the quality of new products. As a kind of SoC, *Multi-Processor System-on-Chip* **(MPSoC)** extends the SoC idea and integrates multiple *General Purpose Processors* **(GPPs)** and special execution units such as *Digital Signal Processor* **(DSP)**. As embedded applications exhibit natural parallelism called *Thread-Level Parallelism* **(TLP)** [52], these multiprocessors platforms can profit from this TLP to increase computation performance with generality and flexibility. To meet all performance, power consumption, cost and time-to-market requirements, these MPSoC based platforms could be the best solution and are already well accepted by the market of both the embedded computing domain and the general computing domain. Fig.1.1 shows the number of execution elements inside one MPSoC growths with the performance and Fig.1.4 presents the design complexity growths with the number of execution elements.

For the execution elements discussed in the ITRS reports, they can have fixed instruction sets or configurable instruction sets. A standard processor has a fixed instruction set and its software is compiled by a standard compiler. In contrast to this standard processor idea,

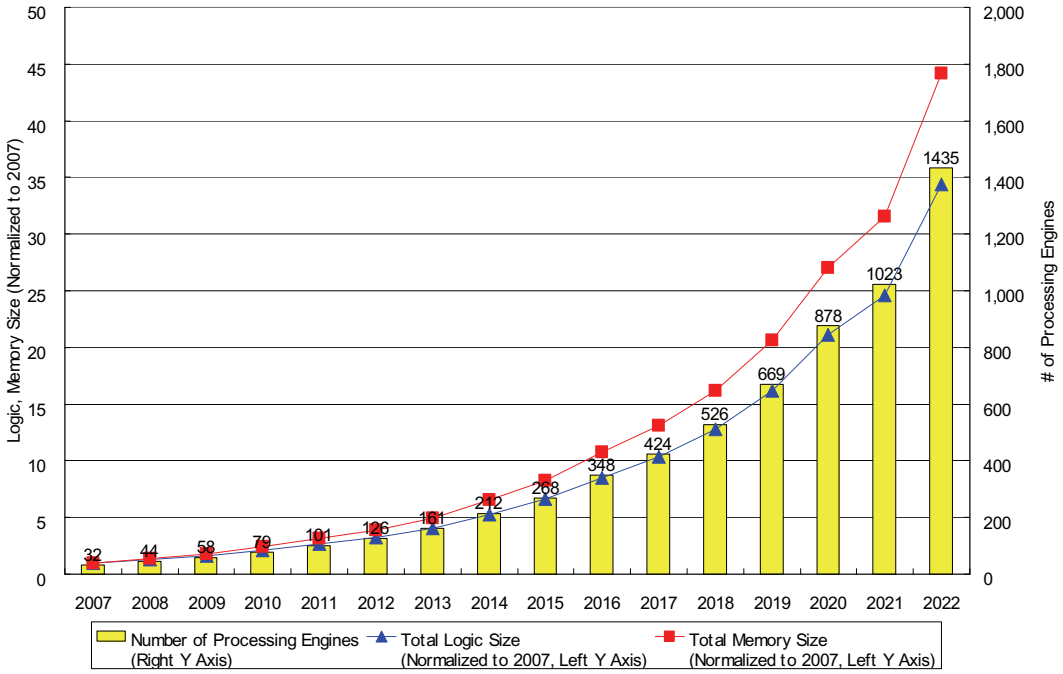

Figure 1.4: SoC Consumer Portable Design Complexity Trends from ITRS 2007 [1]

some existing technologies allow designers to customize both instruction sets and compilers. This kind of processors is normally called configurable processors. With this configuration process, configurable processor can achieve specific speed-up for a class of applications. As normally embedded systems only need to support a very small group of applications, they can profit from design specific instruction set to achieve higher performance without much cost and power consumption lost.

In this thesis, we will discuss the exploration flow specifically designed for the heterogeneous MPSoC based on configurable processors. This study will require multiple abstraction levels. The rest of this thesis is organized as follows:

Chapter 2 presents the background of configurable processors and defines three classes of problems which we focus on in this thesis: the definition and modeling of multiple abstraction levels for heterogeneous MPSoC based on configurable processors, the exploration flow for this kind of heterogeneous multiple configurable processors SoC architectures and the complex *Hardware/Software interface* **(HW/SW interface)** for them. At the end, we propose several open questions which will be solved in the following chapters.

Chapter 3 analyzes the related works for all these questions proposed in chapter 2. At the end of this chapter, we conclude that there are no existing works which can well handle all these questions for the configurable processors and the heterogeneous MPSoC architectures.

Chapter 4 proposes 4 abstraction levels for the MPSoC modeling. These abstraction levels are defined and analyzed based on different software programming interfaces, software simulation technologies and hardware modeling technologies. A high level design space exploration flow is created based on these abstraction levels to solve the huge design space problem.

Chapter 5 provide a hybrid simulation platform to solves the *Hardware-dependent Soft-*

ware **(HdS)** adaptation difficulties. This platform try to realize all hardware related operation with SystemC module instead of the traditionally assembly code and hard-coded I/O addresses to make the adaptation process simpler.

Section 6 analyzes the Hardware/Software interface complexity of heterogeneous MPSoC based on configurable processors. This section proposes a modeling method for Hardware/Software interface automatic generation. A task migration method is also provided to make the hardware/software interface much efficient and flexible for the heterogeneous architectures.

Section 7 is the experiment section which uses the Motion JPEG decoder case study to validate our design methods for configurable processors.

Section 8 gives conclusion and possible follow-ups of this thesis.

Chapter 2

Problem Definition

Contents

Since trends of embedded systems are discussed in the chapter 1, we would like to introduce one of the possible solutions, the heterogeneous MPSoC platform based on configurable processors. This thesis focuses on some problems related to the configurable processors and the Hardware/Software interface of this heterogeneous MPSoC platform.

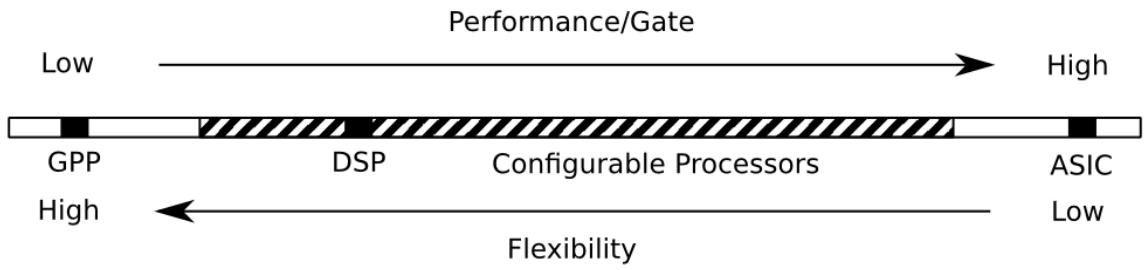

Figure 2.1: Trade-off between performance/gate ratio and flexibility

In the first section, we introduce the background of configurable processors and show advantages of the MPSoC architecture based on this kind of processors. Though this platform is not new [49], we would also like to show its advantages and drawbacks. After that, we give out the motivation of this thesis which is to provide higher performances without losing much flexibility and reduce the design effort for this complex MPSoC platform. In the third section, we try to explore this MPSoC architecture based on configurable processors by using the abstraction modeling of MPSoC platforms. After that, we present some existing problems of the abstraction modeling technology specifically for configurable processors. The HW/SW interface is especially complex with configurable processor based heterogeneous MPSoC systems. The fourth section provides an hybrid MPSoC simulation platform which can be used to solve the HW/SW interface modeling and the *Hardware-dependent Software* **(HdS)** adaptation problem found in the third section. In the fifth section, we point out the inflexibility and low efficiency problem of existing HdS for this kind of systems. At the end, we conclude this chapter by outlining several unsolved problems which will be partly solved in the following chapters.

2.1 Configurable Processors and Heterogeneous Multi-Processor System-on-Chip (MPSoC)

2.1.1 Background of configurable processors

To evaluate one processing unit, there are two important parameters which are the *performance/gate ratio* and *flexibility*. The performance/gate ratio represents how efficient each gate is used for one hardware system, while flexibility shows how easy this hardware system can be adapted to different applications. In Fig.2.1, several well used processing unit types are compared with these two parameters. Because of generality, the *General Purpose Processor* **(GPP)** normally has the least performance/gate ratio with the highest flexibility. In contrast to GPPs, ASIC systems normally can fully utilize hardware resources but become rigid and inflexible for different applications. Among all possible execution units listed in Fig.2.1, the configurable processor is a special kind of application specific instruction set processors and good at the trade-off between the performance/gate ratio and flexibility for embedded system.

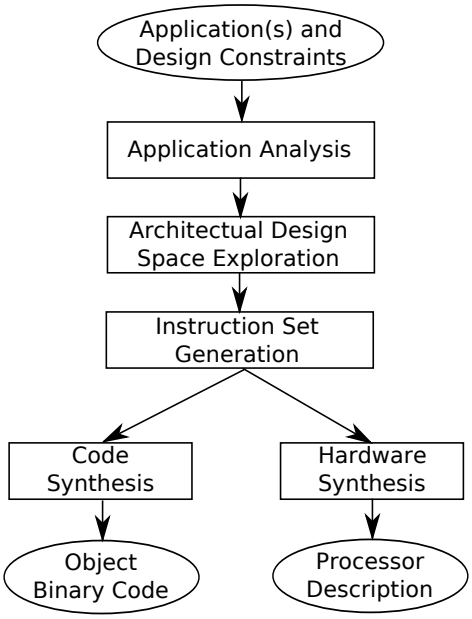

Figure 2.2: Flow diagram of ASIP design methodology from [58]

Application Specific Instruction set Processor (ASIP)

Different from general computation environments, in the embedded system domain, most processors only need to support a very small group of applications. With this constraint, the idea of designing a new processor with the instruction set specifically targeted to a group of applications for higher system performance became mature and appeared almost 30 years ago [103]. The most important reason for popularity of ASIP is the trend of embedded system complexity discussed in chapter 1. With the increase of complexity, it becomes harder and harder to realize the whole system with only ASIC hardware modules because of the design and verification cost and inflexibility for different applications. To follow the embedded system requirement trends, ASIP requires less hardware development resources and provides a more flexible solution for a group of applications [64].

Fig.2.2 shows the flow diagram of a general ASIP design methodology. In this flow, the **application analysis** uses a group of target applications for one embedded platform. Based on these analysis results, there are several possible methods to help generate the application specific instruction set.

- Build an ASIP by directly abstracting suitable instructions from candidate applications. There is much research work focusing on this method such as [103].
- Another possible solution is to select the more appropriate instructions from a set of predefined instructions or generate candidate instructions with predefined templates. Some researchers use this method to build their ASIP solutions such as [93] and [97]. This method is commonly used in latest ASIP automation processes.
- To achieve higher performance with lower cost and power consumption, researchers have created a trade-off solution which can both support user defined instructions and

easy for OS porting and software compiler generation. Configurable processor is a possible solution in which an existing general purpose processor instruction set is extended by adding and removing application specific instructions. As we take account of software development environments such as compiler building, *Operating System* **(OS)** porting and development effort, the configurable processors with a fixed instruction set core becomes a much more mature solution for industry projects. That is why many research and industry works focus on this method [43] [51] [95] [48] [5] [90] [41]. In the following thesis, all configurable processors mentioned belong to this group.

General structure

A general configurable processor should include both the basic core instruction set and the basic core register file. All these instructions and registers should appear in any configuration instances. Normally, this basic core instruction set includes the following groups of instructions.

- **Arithmetic and Logic Instructions**: which are used for arithmetic and logic operations.
- **Memory Access Instructions**: which are used for data transferring between memories and registers.
- **Program Flow Control Instructions**: which are used for changing the programming execution flow based on the processor status.
- **Concurrent Access Control Instructions**: which are used to avoid non-coherence shared memory access cases in multi-processors execution environments.

Meanwhile, four classes of registers should be included in the basic core register file.

- **General Purpose Registers**: which are used for the storage of the data and address information.
- **Program Counter Registers**: which are used to indicate the current program address.
- **Program Status Registers**: which are used to store current processor statuses and include the exception status, the interrupt status and so on.
- **Program Stack Register**: which is used for the current stack address and can be realized using a general purpose register.

As we use specific extended instructions to speed up the application execution, we define extended instruction sets and the relationship between them. Most extension instructions can be divided into two classes:

- **SIMD Instructions**: which can process in parallel multiple data inside one instruction with the same operation.

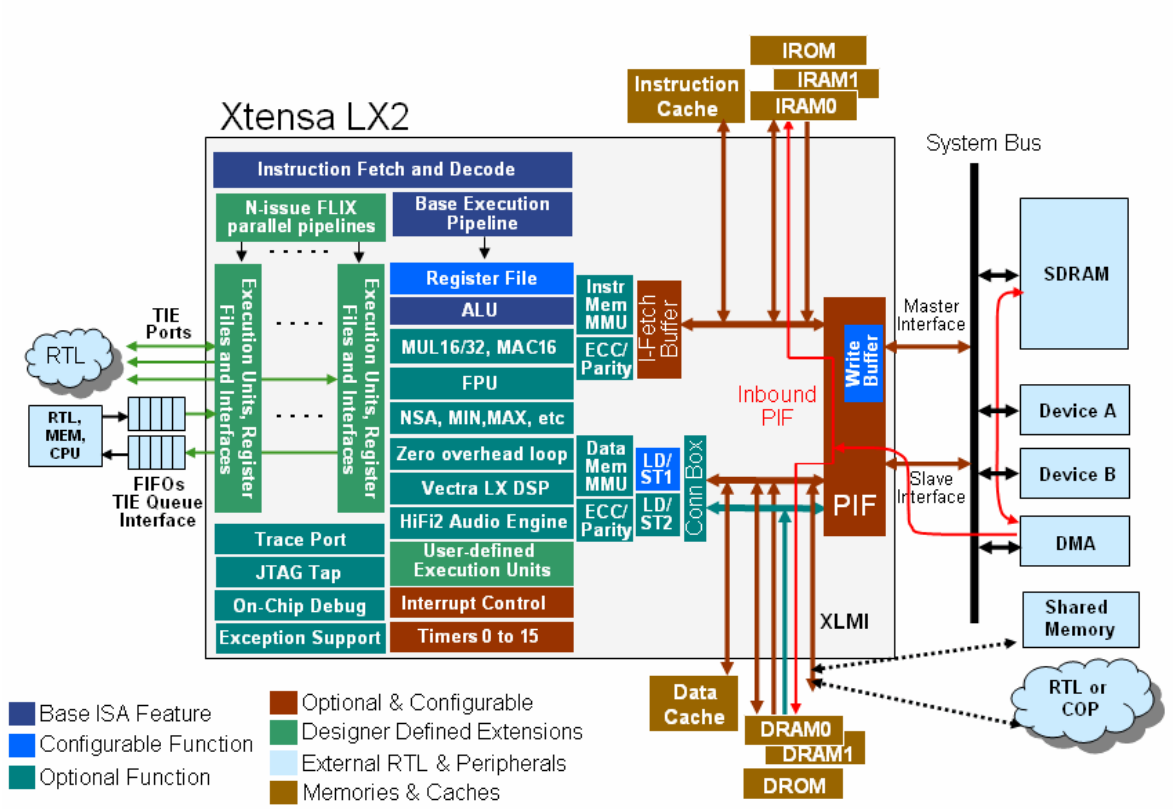

Figure 2.3: Tensilica LX2: a configurable processor architecture example from [3]

- **MIMD Instructions**: which can process in parallel multiple operations without dependencies inside one instruction.

Besides extended instructions, we can also extend register files to improve the performance.

- **Very Wide Registers**: which are used to improve SIMD instructions performance, we add very wide register files to store operands and results.
- **Special Internal Registers**: which are used for some special operations such as accumulator registers for the multiply-accumulate operation.

To well explain internal architectures of configurable processors, the LX2 configurable processor from Tensilica [3] is presented in Fig.2.3 as a case study. The left part of this figure shows the configuration property which uses two different data paths for the instruction execution. The right path is the path for the execution of the predefined instruction set, while the left one is specifically designed for user defined instruction set which can support several kinds of specific bit level operations, SIMD instructions and MIMD instructions.

Instruction Level Parallelism (ILP)

Instruction level parallelism is a measure of how many operations in one computer program can be performed simultaneously. For one application, the ILP can be increased by

removing *data dependences*, *hazards* and *control dependences* [52]. To really profit from this ILP, the processor should have the ability to execute multiple instructions simultaneously. If all simultaneously executed instructions share the same operation code, this kind of ILP is called *Single Instruction Multiple Data* **(SIMD)**. Meanwhile, if these simultaneously executed instructions have different operation codes, this kind of ILP is called *Multiple Instruction Multiple Data* **(MIMD)**. Both SIMD and MIMD instructions can be added to configurable processors for the ILP exploration.

To improve overall performance, embedded systems need to improve ILP also. One possible solution is to replace the traditional RISC processors with ASIP to improve ILP for one embedded SoC. For example, the ARM Cortex processor [6] is one kind of latest embedded processors which support several kinds of extensions to improve ILP such as NEON Media Processing Engine [7] and Jazelle extension for Java virtual machine [8].

Different from these predefined instruction set solutions, configurable processors can have specific extended instructions and have higher ILP with lower cost and power consumption. To show the ILP advantage of configurable processors, we have one block of C code which is a small part of the *Inverse Discrete Cosine Transform* **(IDCT)** task.

```
for(k = 0; k < 8; k++)
  for(l = 0; l < 8; l++)
    Y[k][l] = input[k][l] << S_BITS;
```

To speed up the execution, we add extended instructions to the core instruction set. This extended instruction set includes 128 bit burst load and store instructions `load_32x4`, `store_32x4` and 128 bit parallel shift instruction `shift_32x4`. With this extension, we have the following refined C code with the assembly language style to realize the same function:

```
Addr AddrInput = &(input[0][0]);
Addr AddrOutput = &(Y[0][0]);
for(k = 0; k < 16; k++) {
  load_32x4(AddrInput, 4);//AddrInput+=4
  shift_32x4();//Use the implicit register
  store_32x4(AddrOutput, 4);//AddrOutput+=4
}
```

With the extended processor, we shorten the execution time from the original 345 cycles to the optimized 71 cycles. That means we achieve almost 5 times higher execution speed with three extended instructions. Though the speed up advantage is attractive, we should also notice that extended instructions consume extra chip areas and power. The extended instruction set used for this example consumes approximately 5k extra gates.

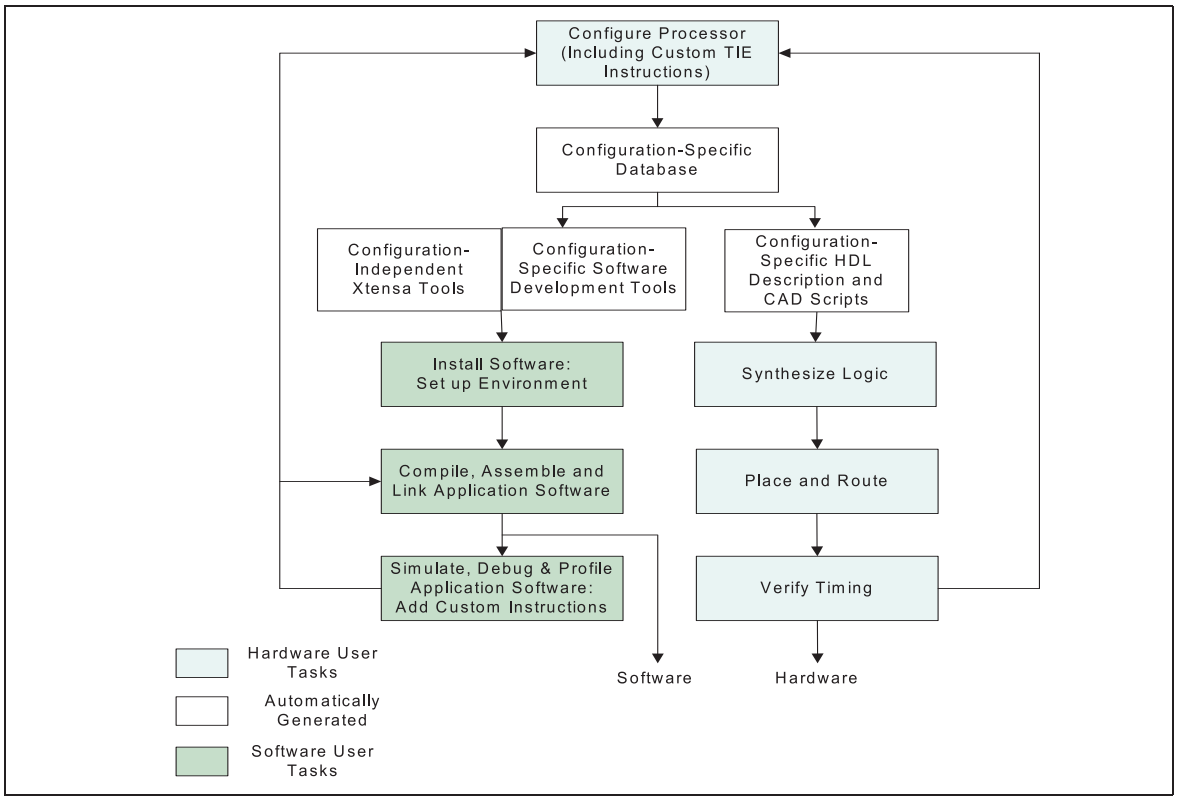

Figure 2.4: Tensilica Tools Set for both Software and Hardware Development [3]

Compiler support

Normally, a configurable processor needs a series of software for both software and hardware development. In Fig.2.4, we have the following two catalogs:

- **Software development tools**: A configurable processor generally has its own compiler which can adapt to a specific extended instruction set automatically. For the Tensilica Xtensa processor case, developers can get a corresponding compiler after modifying the processor instruction set. With this compiler, developers can use the extended instructions by including some corresponding predefined functions in the source code. The compiler can automatically transfer these functions into extended instructions. This work avoid much assembly coding for application development.

- **Hardware Synthesis tools**: Hardware developers can get the synthesized configured processors from this tools set. For the Tensilica Xtensa processor case, the extended instructions are described in an RTL level language which can be synthesized, placed and routed for tape-out. The hardware synthesis tools can help integrate the extended instruction set with the basic processor core for the real hardware implementation.

Different from these commercial tools, there are also some academic configurable processor solutions such as [75] which extends the basic core instruction set by hand code RTL code and without much compiler support. All these solutions are suitable for following thesis.

In contrast to the configurable processors, the reconfigurable processors are also well accepted as a possible way to improve ILP. There are two kinds of reconfigurable processors and both of them are based on the reconfigurable fabric (for example FPGA). The first one has the static configuration which does not modify during the system execution. For example, the research work [99] [56] and [9] belong to this group. Different from the static method, there are also the run-time on the fly reconfigurable processors [69] which can modify the configuration during the system execution. Fundamentally, the most important interest of the reconfigurable processors is the reconfigurable fabric which can help increase the system performance by hardware implementation of some functions or extended instructions for the dedicated embedded applications. The configurable processors share the similar basic idea but are implemented with the traditional fabrication processes.

2.1.2 Heterogeneous MPSoC platform based on configurable processors

The multiple configurable processors idea can be used to achieve both TLP and ILP for the whole embedded system. As both TLP and ILP related works were discussed in previous sections, we would like to focus on how to use the heterogeneousness property to improve the overall performance and reduce the manufacture cost for embedded systems.

Heterogeneousness

In contrast to *Symmetric Multiple Processor* **(SMP)** architecture, heterogeneousness is frequently used to design an MPSoC system composed with different processors. In the embedded system domain, heterogeneous MPSoC appears more than one decade ago and at the commercial market, there are many products such as the TI OMAP platform [10], the TI Davinci platform [11], the ST Nomadik platform [12] and the Atmel SHAPES platform [82] which integrate both RISCs and DSPs together to meet complex application requirements.

The advantage of heterogeneousness is to map different kinds of tasks onto different processors to fully profit from execution properties of each task. Generally speaking, RISCs have better performance and power consumption for OS and control related tasks while DSPs are designed mainly for intensive mathematical computation tasks. The appropriate usage of heterogeneousness can help achieve better flexibility and satisfactory performance with lower power consumption and costs. Fig.2.5 (a) shows the traditional heterogeneous MPSoC architecture in which each CPU subsystem uses SMP structure and shares the same HdS image.

As this thesis focuses on configurable processors, the heterogeneousness can be realized much easier with configurable processors. Fig.2.5 (b) shows the heterogeneous MPSoC architecture in which each CPU subsystem uses the asymmetric structure based on the same base core and uses different extended instruction sets for different processors. Meanwhile, different CPU subsystems can use different base cores to take more variety. Compared with the traditional architecture, configurable processor can provide much more heterogeneous-

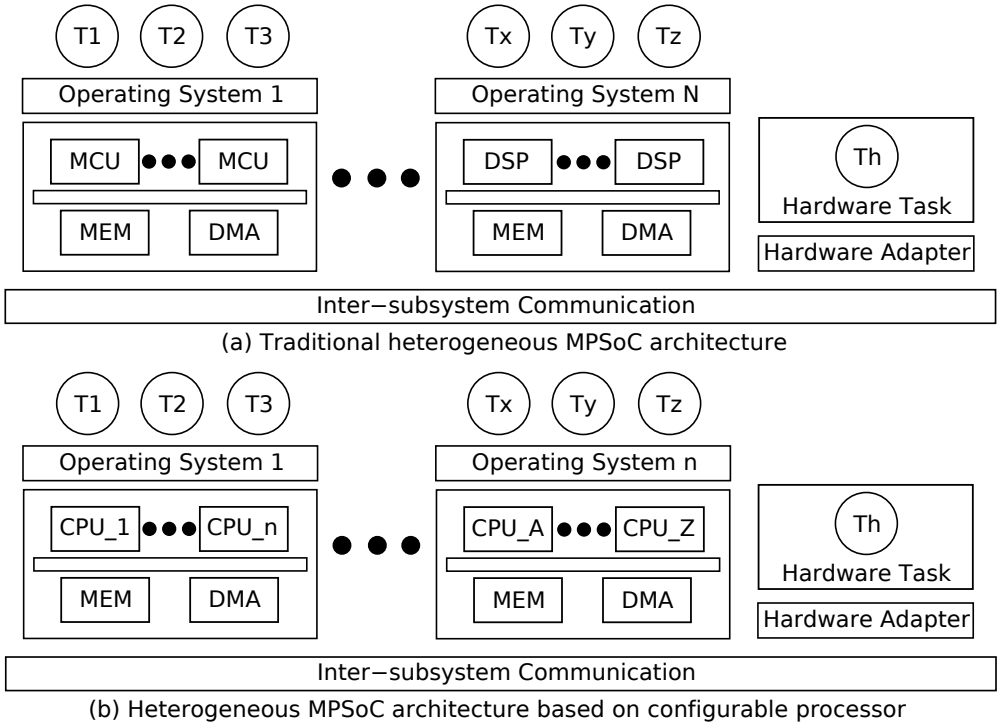

Figure 2.5: Heterogeneous MPSoC architecture based on configurable processors.

ness and flexibility. But we should notice this kind of architectures requires much sophisticated HdS and application development environments. We try to meet these requirements in the following chapters.

Performance and cost advantages

Performance advantage is one of the most important reasons why heterogeneous MPSoC architecture is chosen for embedded systems. To improve performance, there are several possible solutions:

- **Adding extended instructions**: As discussed in the configurable processor section, by adding application specific instructions, the whole embedded system can achieve higher performance.
- **Adding multiple processors**: With more processors, multi-tasks and multi-threads system can achieve higher overall performance with both task level or thread level parallelism.
- **Changing task mapping**: By using the heterogeneousness property for one embedded system, it is possible to improve system performance by changing the task mapping or migration algorithm to fully use extended instruction sets of different processors.
- **Refine system communication**: System communication sometimes becomes a bottleneck of one MPSoC system. Different communication architectures can be explored to meet both the delay and throughput constraints.

Power consumption advantage is also one interesting point for the heterogeneous MPSoC architecture used in embedded systems. To reduce power consumption, there are several possible solutions:

- **Multiple processors instead of high frequency**: For the same computation requirement, using more processors solution requires less power consumption than the high frequency solution because of lower circuit voltage [50].
- **Extended instruction set instead of high frequency**: For some specific applications and tasks, power consumption can be reduced by using specific extended instructions. This method can help reduce the processor frequency to save power consumption.
- **Heterogeneousness instead of symmetric**: For different types of applications and tasks, the heterogeneous property can be useful to reduce power consumption with the same performance. This method is especially useful to reduce the processor frequency for power consumption saving.

The *cost* advantage is also obvious to this kind of heterogeneous MPSoC platforms. With extended instructions, it is possible to reduce the processor number, the cache size and so on while keeping a fairly good flexibility and re-programmability. Meanwhile heterogeneousness can provide more specific acceleration for various applications.

2.2 Motivation

As both configurable processors and the heterogeneousness property have all of the performance advantage, the power consumption advantage, the cost advantage and the flexibility advantage, it would become a suitable candidate for the next generation embedded system. But the configurable processor is still a relatively new idea for embedded system designs in industry. A general heterogeneous MPSoC architecture and its associated design flow for embedded system design is still under research. Compared to the traditional single core embedded system based on off-the-shelf processors, all of configuration, heterogeneousness and multiprocessor properties provide much higher flexibility for system designers. Until now, there are no well accepted design flows to well handle the flexibility provided by this kind of systems. The focus of this work is to build a possible design flow for heterogeneous MPSoC platforms based on configurable processors.

Because of the complex and huge design space presented by configurable processors and heterogeneous MPSoC, a modeling method and a corresponding simulation platform are necessary for this flow. In this thesis, both the modeling method and simulation platforms are based on the abstraction level idea. With abstraction levels, one complex system can be more easily modeled at higher abstraction levels and details are described at lower abstraction levels. At higher abstraction levels, the simulation platform is much faster than the one at lower abstraction levels with less accuracy. Based on these models and simulation platforms, a design space exploration flow can be built to help designers find the most suitable processor

configuration and heterogeneous MPSoC architecture. In this design space exploration flow, multiple abstraction levels modeling technology can help fix different levels of detail and the multiple abstraction level simulation platform can help verify both function and performance at each exploration step. This flow with multiple abstraction levels can also close the huge gap between requirements and realization details by adding several intermediate models and dividing big design loops into smaller ones.

In this design space exploration flow, we find that there are two problems unsolved. The problem of the *Hardware/Software interface* becomes extremely complex because of the growth of complexity and flexibility of configurable processor based heterogeneous MPSoC platforms. The HW/SW interface becomes a time consuming and error-prone part of the whole design flow. In fact, this HW/SW interface adaptation and HdS porting work is the base of the design space exploration flow proposed before.

Beside this problem, another one is to have a software task management system which can get full profit from this kind of heterogeneous MPSoC platform. A well designed task management system can make workload balance between multiple processors and have better performance/cost ratio for embedded systems.

These 3 sections are necessary to establish configurable heterogeneous MPSoC as a mature and viable solution to embedded applications in the future.

2.3 Design Space Exploration Flow with Multiple Abstraction Levels

Abstraction is the process or result of generalization by reducing the information content of a concept or an observable phenomenon, typically in order to retain only the information which is relevant for a particular purpose. In design space exploration flow, abstraction is useful to help balance between simulation speed and result accuracy in term of performance or even function. This property needs to be especially emphasized for configurable processor based embedded systems.

Based on the abstraction levels, we would like to build a design space exploration flow which can fully profit from the abstraction idea.

2.3.1 Classification of abstraction

Different use cases

There have been many abstraction level definitions before. For all these works, different abstraction levels target at different users. For example, application software developers may need to know that interruptions are triggerable and what interrupt handlers are available. While HdS developers need to realize these handlers according to features of hardware configurations. At the end, hardware developers implement these features by grouping and combining logic circuits. In this thesis, three groups of users are focused:

- **Application software developers**: This group of developers uses simulation models to help the design of application software. Instead of taking care of very detailed implementation information, they just need a very gross view of the hardware architecture and use already existing operating system, drivers and libraries.
- **Hardware dependent software developers**: This group of developers uses simulation models to accelerate and verify hardware dependent software porting works. For these developers, the detailed processor architecture and I/O devices behavior are really important for their works. Besides the user interface related porting works, most HdS porting works do not require very high simulation speed.
- **Hardware developers**: This group of developers uses simulation models for hardware development and verification. These designers take simulation models to develop test vectors and use the simulation platform as the golden reference model for the test results comparison and the hardware system verification. With these requirements, the simulation model should be very accurate to real hardware platform and cycle/byte accurate. The simulation speed is not important for test vector execution and comparison.

For all these three groups of developers, it is clear that a single simulation platform cannot meet all these different use cases. Abstraction levels are the solution to model the same hardware platform with totally different software programming interfaces, simulation speed and result accuracy.

Different interfaces

For these three groups of users, different *application programming interfaces* **(APIs)** should be taken into account. For example, application software developers would like to have a much higher level interface than hardware developers.

This thesis would like to answer this question: *What interfaces should be provided to different kinds of users by simulation platforms at each abstraction level?*

Different simulation technologies

As different groups of users have different speed and accuracy requirements, to satisfy all user requirements, each simulation platform should have different software simulation technologies to satisfy user defined requirements.

This thesis would like to answer this question: *What simulation technologies should be provided to different kinds of users by simulation platform at each abstraction level?*

Different internal modeling

Besides the software simulation part, hardware modeling is also important for a simulation platform. With different user requirements, different levels of detail should be presented at different abstraction levels.

This thesis would like to answer this question: *What internal modeling details should be provided to different kinds of users by simulation platform at each abstraction level?*

2.3.2 Design Space Exploration (DSE)

In the embedded system design domain, for a group of applications, there are more than thousands of possible solutions with different realization details. All these possible solutions make up a huge design space. Among this design space, designers need to choose the one which can meet all user defined requirements such as performance, power consumption, cost and so on. This comparison and choosing process is normally defined as *Design Space Exploration* **(DSE)** in the academic domain. Especially for embedded systems, this process is really important and unavoidable for a successful system.

User constraints

In this thesis, all user defined requirements are transferred into constraints. These constraints can include some very high level ones such as the total chip performance should be able to finish 25 frames of video decoding in one second and the total chip size should be smaller than 1 cm^2. Besides these high level constraints, there are many low level ones such as the cache miss rate of one processor and so on. It is natural to bind all different granularities constraints with different abstraction levels, but the binding and refinement method is still an open question.

This thesis would like to answer this question: *How to define constraints of each abstraction level and refine all high level constraints to low level ones?*

Flow with abstraction levels

There are many possible ways for this multi-processors design space exploration. Until now, some most accepted methods are simulation based ones which are mature and flexible. As speed and accuracy are the two most important properties of one simulation platform, abstraction levels are used to balance these two properties. The design space exploration flow should well utilize all these provided abstraction levels and avoid much design loop cost existing in most traditional flows.

Heterogeneousness and processor configuration is specially focused in this thesis. These two properties lead to the question how to well integrate these properties with different abstraction levels. Because these properties make the design space much huger than traditional MPSoC architectures, we also want to solve the problem of how to control these two properties at lower abstraction levels for the huge design space problem and how to use these two properties to avoid some design loops during the DSE process.

This thesis would like to answer this question: *How to integrate all these abstraction levels together into a design space exploration flow?*

2.3.3 Questions

- How many abstraction levels are necessary to meet different groups of user requirements?
- What interfaces should be provided to different kinds of users for each abstraction level?
- What simulation technologies should be provided to different kinds of users by simulation platform at each abstraction level?
- What internal modeling should be provided to different kinds of users by simulation platform at each abstraction level?
- How to define constraints of each abstraction level and refine all high level constraints into lower ones?
- How to integrate all these abstraction levels together into a single design space exploration flow?

2.4 Hardware/Software Interface Modeling and Hybrid MPSoC Simulation Platform

Hardware/software interface **(HW/SW interface)** is used to describe the connection between software application modules and hardware functional modules in embedded SoC platforms. Fig.2.6 shows more and more details of HW/SW interfaces from left to right.

As this thesis focuses on the design space exploration flow based on configurable processors and heterogeneous MPSoC architectures, the HW/SW interface difficulty due to configuration and heterogeneousness should be solved to simplify the whole design flow .

2.4.1 Hardware/software interface modeling

Hardware/software interface Complexity

With the complexity of configurable processors and heterogeneous MPSoC architectures, the *Hardware/software interface* becomes critical in the whole system development process. During the DSE process, processors of the HW/SW interface need to be configured to meet requirements of the application software. Even worse when architectures and configurable processors are modified during the design space exploration process, the underlying part of one HdS should be modified frequently and leads to maintenance and portability problems.

Bottleneck of automatic generation

A possible solution for this complex HW/SW interface problem is to generate one HW/SW interface for a specific application platform automatically. In the embedded system domain,

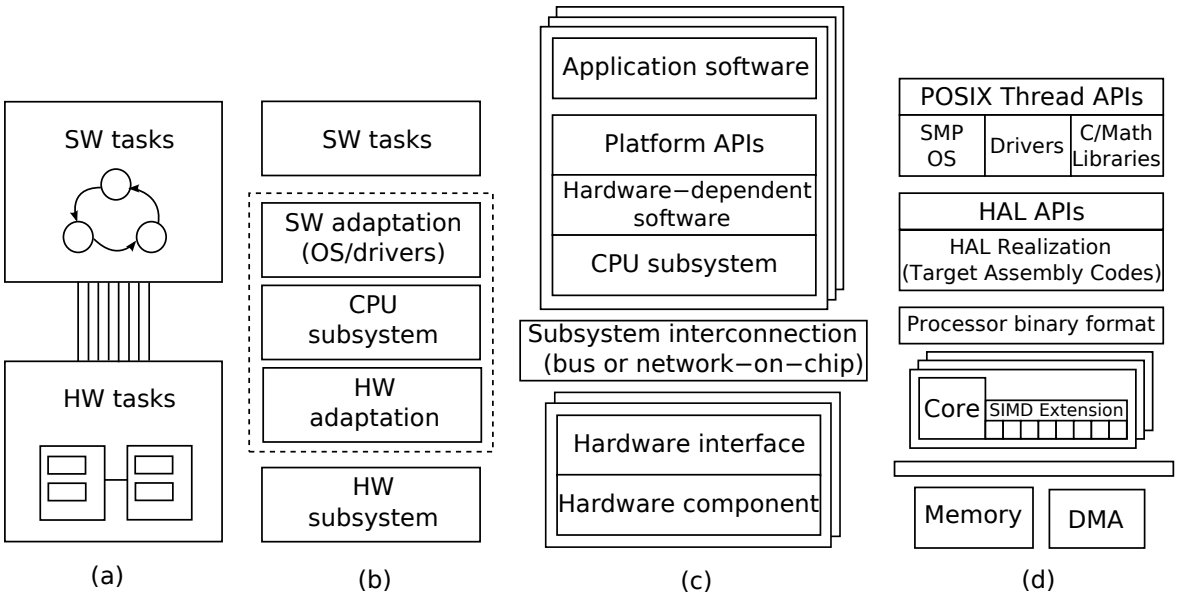

Figure 2.6: The hardware/software interface idea for a general MPSoC platform [61]. (a) A high level HW/SW interface description which uses some simple lines to connect between software tasks and hardware tasks. (b) The less abstract HW/SW interface squared with dot-lines. This interface uses CPU subsystems to execute software tasks. Abstract hardware and software adaption can help integrate the whole system together. (c) The software adaption is refined into platform APIs and HdS. The hardware adaption is refined into hardware interfaces for connections of hardware components. Subsystem interconnection is used to integrate all these hardware and software subsystems together. (d) The most detailed HW/SW interface is shown in this figure. As this thesis focuses on configurable processors, detailed HdS and CPU subsystem are shown with POSIX Thread APIs, HAL APIs and the processor binary format. From (a) to (d), these four figures express the same HW/SW interface idea from very abstracted ones to detailed ones. In (c), multiple CPU subsystems and hardware subsystems are connected with interconnection for a general MPSoC platform. Different from HW/SW interfaces of traditional MPSoC platform, in (d), SIMD extensions of each processor can be different for one CPU subsystem.

the automatic generation of the HdS and the hardware platform is called *System Level Synthesis* process. Unfortunately, the formulation of this problem is not well formalized and is not easy to formalize as a whole.

Though most synthesis ideas are extensions of *Register Transfer Level* **(RTL)** synthesis, system level synthesis lacks the background of Boolean algebra and digital logic. Without strong mathematic background and relationship, the system level synthesis remains a design job.

This thesis would like to answer this question: *How to model the HW/SW interface to facilitate the automatic generation and support complex architectures such as the configurable processor based heterogeneous MPSoC platform?*

2.4.2 High level Application Programming Interfaces (APIs) realization

Automatic generation of the HW/SW interface is not the only solution, we may have other methods to make the integration of both the hardware and software easier. During early stages of the exploration process, these modification requires many updates of HdS implementation to validate both user applications and the underlying MPSoC hardware architecture. For example, the assembly HdS code realization consumes much effort and is error prone for early design stages. Meanwhile, we also notice that normally the HdS does not occupy many of the whole platform computation resources. How to make the HdS realization suitable for the design space exploration at early stages becomes an open question for high abstraction level platform designers. Beside this question, how to well integrate high level software interfaces with different simulation technologies is also a question this thesis tries to solve.

2.4.3 Questions

- How to model the HW/SW interface to facilitate the automatic generation and support complex architectures such as the configurable processor based heterogeneous MPSoC platform?

- How to build high level simulation models with high level programming APIs to avoid HdS porting works?

2.5 Task Migration Framework for Heterogeneous MPSoC Architectures

Besides the complex HdS modeling and generation problem, it is also a difficult problem to design a high performance and flexible task migration enabled HdS for heterogeneous MPSoC.

2.5.1 Rigid heterogeneous MPSoC platform

Though heterogeneous MPSoC platforms have many advantages which were discussed before, they are not as flexible as existing SMP based MPSoC platforms. Normally, in heterogeneous MPSoC platform, each task is mapped onto one specific processor instead of a group of processors like on SMP platforms. This rigid mapping requirement makes either the task partitioning very complex or task load unbalanced among multiprocessors.

2.5.2 Mapping model of computation

There are several models of computation for multiprocessor embedded systems such as *Synchronous Data Flow* **(SDF)** model [72] and *Kahn Process Networks* **(KPN)** model [62]. As these models are not dependent on the underlying MPSoC platform, the mapping between tasks/processes and each processor is not defined. For heterogeneous MPSoC platform based on configurable processors, because processors are different among them, existing mapping solutions are almost fixed for all models of computation.

This thesis would like to answer this question: *How to make heterogeneous MPSoC more flexible by providing lightweight task migration between different processors?*

2.5.3 Questions

- How to make heterogeneous MPSoC more flexible by enabling the task migration function?

2.6 Conclusion

In this chapter, we proposed 6 questions which need to be solved for design space exploration flows based on configurable processors.

- How many abstraction levels should we have and how to integrate software programming interfaces, software simulation technologies and internal modeling together for one abstraction level?
- How to define constraints of each abstraction level and refine all high level constraints into lower ones?
- How to integrate all these abstraction levels together into a single design space exploration flow?
- How to model the HW/SW interface to facilitate the automatic generation and support complex architectures such as the configurable processor based heterogeneous MPSoC platform?
- How to build high level simulation models with high level programming APIs to avoid HdS porting works?

- How to make heterogeneous MPSoC more flexible by enabling the task migration function?

Chapter 3

Related Works

Contents

For most of the questions proposed at the previous chapter, there are already many research works related to them. In this chapter, we present some of these works and use some parts of them as the background of our solutions. Meanwhile, the differences of our solutions are also analyzed to show our contributions.

3.1 MPSoC Abstraction Levels

For MPSoC abstraction levels, we take a look at three aspects which are *software simulation technologies*, *hardware modeling technologies* and *software programming interfaces*.

3.1.1 Software simulation technologies

As the trend is to move hardware functional modules into software realization for embedded systems, software simulation technologies are important for both the performance and the accuracy of one MPSoC simulation platform.

Basic software simulation technology

To simulate software, *Instruction Set Simulator* **(ISS)** is the most well known technology which mimics the behavior of a mainframe or microprocessor by "reading" instructions and maintaining internal variables which represent the processor's registers. In most embedded system simulation platforms, the ISS is grouped with peripherals and memories to model the whole SoC. Runtime instruction translation technology is the simplest and most robust solution for both the software development and the hardware verification. The most serious drawbacks of this solution are the simulation speed and the lack of visibility of memory accesses. The second drawback can be avoided if the ISS solely interpret instructions, as the rest of the system is modeled accurately [54].

Software annotation

Software annotation is used in some high level MPSoC simulation approaches to provide performance informations. Some of them require explicit manual source annotation by programmers [102][78][100]. There are also partially automatic solution for the annotation process [101][36][89][55]. All these methods provide the possibility to get the performance of one MPSoC platform with a simulation speed close to the native execution.

As the software performance relies much on the processor configuration, existing software annotation methods have still hard problems to well model this instruction set modification effect.

Dynamic binary translation

Dynamic binary translation, known also as *Just-In-Time* **(JIT)** compilation, can convert objects into target binaries during the runtime. In contrast to static translation, dynamic one can avoid long translation time during download and startup. There is much research work focusing on this method such as [40], [42], [92] and [30]. Some of these works provide a software translation kernel while some others model the whole hardware platform and provide just the same simulation properties as the real one. Many machine emulations are based on this approach.

The advantage of this technology is the high speed simulation ability while the drawback of this method is hard to express the pipeline effect (control and data dependencies) and other detailed hardware properties such as the instruction fetch. In this thesis, we focus on how to take the advantage of this technology to build our multiple abstraction levels and related exploration flow.

3.1.2 Hardware modeling technologies

Traditional hardware simulation

In the digital system domain, the *Register Transfer Level* **(RTL)** is used to describe a digital circuit by defining the flow of signals between hardware registers and the logical operations

performed on these signals. As this modeling technology provides the synthesis ability on libraries of standard cells, it ensures a path to layout implementations. It is therefore well accepted as a hardware modeling solution.

SystemC

At beginning, SystemC [13] is a hardware description language which focuses more on system description than VHDL and Verilog. SystemC includes a set of data types, macros and library routines implemented in C++ language to well support event driven hardware simulation. Instead of focusing on the gate level synthesis as the RTL level, SystemC is designed to provide a system description of complex digital platforms. Compared to synthesizable RTL level models, the SystemC models are much easier to build and comprehend. Simulation speed of SystemC models is also higher than RTL level models, but it doesn't target to the synthesis ability.

Transaction Level Modeling

With the SystemC 2.0, SystemC extends the original hardware description with *Transaction Level Modeling* **(TLM)** [14]. TLM is a high-level approach to model digital system where details of communication among modules are separated from the details of the implementation of functional units or the communication architecture. Since this modeling technology simplifies the communication interface, the compatibility and higher simulation speed are two obvious advantages.

3.1.3 Exploration Flow

Design space exploration is a huge domain which includes topics such as hardware architecture refinements, hardware/software partitioning, software tasks mapping and scheduling.

Automatically extracting the candidate extended instruction set from one specific application is focused by both academia and industry. L. Pozzi et al. [90], F. Sun et al. [95] and Xtensa Processor Extension Synthesis (XPRES) Compiler [3] are such works. They try to extract the most efficient instruction set to meet timing, cost and power requirements. As these works focus on automatic generation for the standalone processor only, in some cases, extension parts are several times bigger than original cores. A better solution could be using multi-processor architectures and configurable processors at the same time. Meanwhile, most of them target pruning the huge search space [90][95]. In our design flow, designers can group and map a series of tasks with similar functions onto one *Symmetric Multi-Processing* **(SMP)** subsystem at higher abstraction levels. The effort spent on choosing the most suitable instruction set from huge candidate instructions can be reduced.

On the thread level parallelism side, designers would explore the multi-processor architecture for higher performance. Several simulation platforms such as SoCLib [15], StepNP [83] and CoWare [16] are developed to accurately evaluate the MPSoC architecture. Most of these platforms use cycle-accurate model or TLM [35] for hardware modeling and ISS

for software simulation. As these platforms need very detailed architecture information, it can only be useful at the very late design stage. Meanwhile, as these platforms include many architecture parameters and ISS, simulation speed becomes slower than design space exploration requirements.

To trade-off among simulation speed, accuracy and flexibility, improving the abstraction level of simulation seems to be a good way to solve these problems [59]. Recently several MPSoC design space exploration platforms such as Metropolis [28], CASSE [91], MESH [38], MILAN [77], Koski [63] and Sesame [86] use similar high abstraction level ideas as our method for design space exploration. Metropolis focuses on the framework for design space exploration and different models of computation. CASSE uses Kahn Process Networks [62] and abstract hardware components to build their simulation-based platform. MESH proposes the layer-based approach for modeling and performance simulation. MILAN is a model-based, extensible framework which supports multi-abstraction level and multi-granularity simulation. Koski uses trace file to annotate the UML state charts and achieve very high exploration speed at this abstraction level. Sesame also use trace file to gradually refine the application model and focuses more on their Y-chart methodology.

With these related works, we find that most configurable processor exploration works focus on standalone processor only while most MPSoC architecture exploration works provide no extra support for configurable processors.

3.2 Hardware/Software Interface

In this section, we introduce the idea of automatic generation of Hardware/Software interface and limitations of existing Hardware/Software interface modeling works. After that, we also introduce the existing HW/SW interface adaptation and HdS porting works.

3.2.1 Hardware/Software interface modeling

With the growth of embedded system requirements and configurable processors, Hardware/-Software interface becomes more complex than before. ECos [17] is an embedded operating system which provides graphic interface for users to configure and port this OS. CoWare [16], Virtutech [18] and ARM RealView System Generator [19] provide user friendly interfaces to help generate glue logics for the MPSoC platform design. Though the efficiency of these industry tools is well verified by the market, these works still need much user effort during the MPSoC design process.

Automatic generation of HW/SW interface seems to be a promising solution to simplify the HW/SW interface design problem [31]. But the system description model is the critical problem to improve the quality of generated MPSoC platforms. Because UML is well defined and used in software development, some research works extend the UML to support the whole embedded system description [67]. Compared with UML, our solution is relatively concise in representation and focuses on the HW/SW interface automatic generation. ROSES [98] is another HW/SW interface automatic generation tools. As this tool need a Co-

lif [39] tools for HW/SW interface modeling, the generation process should also be defined as semi-automation tools. The ability of all these automatic generation works are constrained by the HW/SW interface modeling. In this thesis, we would like to provide a new modeling technology which may provide higher flexibility for the HW/SW interface modeling.

3.2.2 Hybrid hardware/software interfaces

Several simulation platforms such as SoCLib [15], StepNP [83] and CoWare [16] are developed to accurately evaluate MPSoC architectures. Most of these platforms use cycle-accurate model or TLM [35] for hardware modeling and ISS for software simulation. Because these platforms need very detailed architecture information and processor configurations, it can only be used at the late design stage when the choice of the overall hardware architecture has already been made. With these platforms, OS should be ported and the porting work should be debugged before simulating user applications.

To save all these OS porting works, some researchers propose the automatic generation of operating systems for specific architectures [44][32]. As these generation methods heavily depend on available templates and libraries, they cannot automatically adapt to new instruction sets and new I/O devices. The *Hardware Abstraction Layer* **HAL** [17] can separate hardware related details from platform independent functions and ease the OS porting work. Unfortunately, designers still need to realize all these HAL APIs with assembly codes for different processor configurations and system architectures.

Simulation speed is also an important parameter for a virtual prototype platform. To speed up the simulation, researchers have built a host environment to locally execute software with annotation data [55][70]. These solutions can provide very high simulation speed, but have accuracy drawbacks (especially for configurable processors which performance heavily relies on user extended instructions).

The idea of hybrid simulation is created to take both advantages of host and target simulation platforms. Software ISS and FPGA hybrid simulation technologies are created to improve the speed of the energy estimation [80] and the functional verification [81]. There are also hybrid platforms which combine the ISS with the host execution [66], but the target of these works is to speed up the simulation with the host execution.

3.3 Task Migration Framework

In these section, we analyze the task migration problem which appears because of configurable processors.

3.3.1 Task migration on configurable heterogeneous MPSoC

At the commercial market, there are many heterogeneous MPSoC products such as the TI OMAP platform [10], the ST Nomadik platform [12] and the Atmel SHAPES platform [82] which integrate MCUs, DSPs and ASIC modules to meet complex application requirements.

Because of the instruction set difference, it is impossible to schedule tasks between different kinds of processors in one heterogeneous MPSoC.

One interesting heterogeneous MPSoC example is the *Cell* processor [53] [85]. The Cell processor includes a general dual-threaded PowerPC processor element as the main processor and 8 synergistic processor elements (SIMD processors) as coprocessors. Software applications are compiled into either the PowerPC binary format or the SIMD coprocessor format. Normally, the task scheduling between coprocessors is controlled by the PowerPC and the task migration between the PowerPC and SIMD coprocessors is not possible.

Different from heterogeneous MPSoC platform, the SMP architecture [52] is composed with two or more identical processors and connected to share main memories. Because the SMP architecture can provide a uniformed execution environment, there are much more scheduling researches and realization systems for this kind of platforms [34]. SMP architecture is well suited to the general purpose computing, since it is easier to program. However it cannot profit from differences of various application requirements. In the embedded system domain, this SMP solution is unfortunate to have weak points in term of power and performance [74].

As there is a huge gap between the homogeneous MPSoC and the heterogeneous one, some researchers try to provide a trade-off solution which can take both advantages. R. Kumar et al. [68], M. Becchi et al. [29], S. Balakrishnan et al. [27] and S. Ghiasi et al. [47] suggest to integrate several processors which implement the same instruction set but with different costs and performances. This limited heterogeneous platform can achieve higher performance than homogeneous one with similar costs. Based on this platform, some scheduling algorithms [37] are designed to achieve higher performance with less power consumption.

3.4 Conclusion

With all presented works, we can find that

- Most existing multi-abstraction levels works do not focus on configurable processors.
- Most existing exploration flows do not really handle configurable processors.
- HW/SW interface is more complex with configurable processors and no specific solution focus on it.

To handle the processor configuration and heterogeneousness of our MPSoC platform, we will present our solution in the following chapters.

Chapter 4

Design Space Exploration Flow with Multiple Abstraction Levels

Contents

Many researchers focus on abstraction levels and propose different definitions from different points of view. Especially, configurable heterogeneous MPSoC architectures are more complex than traditional ones. To make abstraction levels easier to define, in this chapter, we propose 3 axes to help classify MPSoC abstraction levels. With these three axes,

this chapter will present 4 different MPSoC modeling abstraction levels. These 4 abstraction levels are targeted for different groups of users with different software programming interfaces. Besides interfaces, simulation platforms at all these abstraction levels use different simulation technologies which can propose a trade-off between the simulation speed and performance result accuracy. Hardware modeling methods are also discussed in this chapter for different abstraction levels.

By using these abstraction levels, we would like to help walking through the huge design space of configurable heterogeneous MPSoC to determine a HW/SW architecture meeting the system design constraints. As mentioned at the chapter 2, with the flexibility of configurable processors and the heterogeneity property, there is an infinity of ways to refine an MPSoC architecture. All these possibilities lead to a huge design space for designers to explore. To be able to target a specific solution to the system design problem at hand, we present a budget based refinement policy which can transfer custom defined constraints into budgets and refine all gross budgets through the defined four abstraction levels.

In the following chapter, we try to answer the following questions:

- How many abstraction levels is necessary to meet different groups of user requirements?
- What interfaces should be provided to different kinds of users for each abstraction levels?
- What simulation technologies should be provided to different kinds of users by simulation platform at each abstraction levels?
- What internal modeling should be provided to different kinds of users by simulation platform at each abstraction levels?
- How to define constraints of each abstraction level and refine all high level constraints into lower ones?
- How to integrate all these abstraction levels together into a single design space exploration flow?

4.1 Classification of Abstraction

As there are different requirements with different groups of users, we would like to classify all these requirements along three axes. In this thesis, these three axes are very important to differentiate 4 MPSoC abstraction levels. They are:

- Software programming interfaces
- Software execution methods
- Hardware modeling technologies

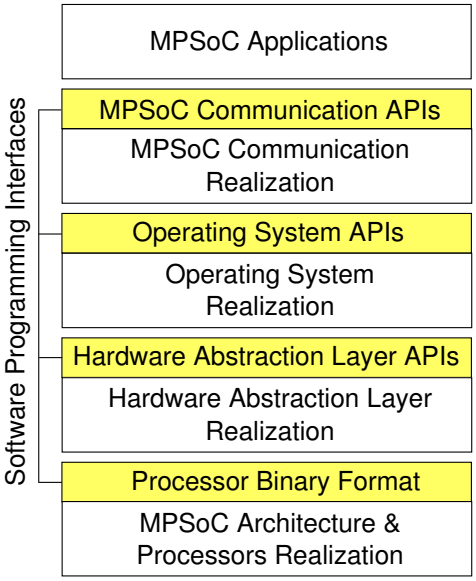

Figure 4.1: 4 different software programming interface.

4.1.1 Software programming interface

The software programming interface is one of the most important axes as it defines the interface between user developed software programs and the MPSoC simulation platform. In Fig.4.1, we have 4 different kinds of software programming interfaces which are *MPSoC Communication APIs*, *Operating System APIs*, *Hardware Abstraction Layer APIs* and the *Processor Binary Format*. These 4 interfaces represent different abstraction levels for software developers and have different abstraction of hardware details.

- **MPSoC Communication APIs**: This software programming interface is at the highest abstraction levels. With this interface, most hardware realization details are transparent to software developers by using inter-processes communication APIs. The advantage of this interface is that the very high level modeling does not rely on the detailed hardware architecture and the HdS realization. Many computation models such as KPN [62], SDF [72] are compatible with these communication APIs. For example, with the KPN communication model, we use two main functions which are `ChannelRead` and `ChannelWrite` to communicate between two different processes. At this level, simulation platforms with this interface can be used to verify the realization of high level computation models to avoid deadlock and resource starvation cases. So this software programming interface is suitable for high level application validation.

- **Operating System APIs**: This software programming interface is lower than communication APIs. At this level, the operating system is fixed and all OS APIs are provided as the software programming interface for application designers. For example, the POSIX Thread OS APIs interface is an example of the OS APIs interface and can be grouped into the following classes:

  - **Thread Manipulation APIs**: This group of APIs includes `pthread_create`,

`pthread_join`, `pthread_exit` and so on. These APIs are necessary for thread management.

  - **Synchronization APIs**: This group of APIs supports two kinds of thread level synchronization APIs which are related to *Mutual exclusion* and *Condition variables*. These APIs guarantee a certain sequence of threads execution and handshake at a certain point to enforce the semantic of parallelism execution.
  - **Thread Local Storage APIs**: This group of APIs handles threads specific data during system execution.
  - **Utility APIs**: This group of APIs includes some utility functions which are useful when working with threads such as `pthread_equal` and `pthread_self`.

With this software programming interface, the realization of all OS APIs can be modeled by simulation platform and made transparent to application designers. So application designers do not need to take care of OS realization details such as spin locks and the thread scheduling.

- **Hardware Abstraction Layer APIs**: This software programming interface is even lower than the OS APIs one. The HAL APIs interface is defined to isolate hardware realization details and upper general operating system functions. We take the eCos HAL APIs interface [17] as our example and these APIs can be divided into the following groups:

  - **Architecture Characterization APIs**: This group of APIs contains basic CPU architecture definitions. The CPU context save format, context switching, endianness, stack size and memory address translation make up architecture characterization APIs.
  - **Interrupt Handling APIs**: This group of APIs includes definitions of exception and interrupt numbers, interrupt enabling and masking, and real-time clock operations.
  - **I/O Access APIs**: This group of APIs provides I/O devices register accessing ability.
  - **Cache Control APIs**: This group of APIs contains definitions for cache structures and operations.
  - **SMP APIs**: This group of APIs supports *Symmetric Multi-Processing* **(SMP)**. They include spinlocks, the processor number and atomic instructions.

  Details about interrupt and context switch are transparent to OS and application designers. This interface is suitable for OS developers to realize and verify OS based on these APIs.

- **Processor Binary Format**: This interface is the most traditional software programming interface used for simulation platforms. X86 [20], SPARC [65] and MIPS [76]

instruction sets are three typical examples of the processor binary format. A general processor binary format should include the following groups of instructions:

- **Arithmetic Instructions**: This group of instructions is used for arithmetic operations such as `add` and `subtract`.
- **Logic Instructions**: This group of instructions contains logic operations such as `and`, `or` and `not`.
- **Data Instructions**: This group of instructions supports data operations for the register-register, the register-memory and the memory-memory data movements such as `move`, `load` and `store`.
- **Control Flow Instructions**: This group of instructions can change the execution flow of programs such as `jump`, `call` and `return` (may be conditional or not).

With this interface, software designers should understand the very detailed computer architecture of the target platform and be responsible for all HdS realization. All user applications, OS and drivers should be well defined and compiled into target binary formats. This interface can be well suitable for HAL developers and the hardware verification usage.

4.1.2 Software simulation/execution methods

Besides the software programming interface, we should also take a look at internal simulation or execution methods of software programs.

Formally, we have two different instruction sets which are $\mathbb{T}$ and $\mathbb{H}$. $\mathbb{T}$ represents the instruction set of the target processor to be modeled. While the whole development environment is based on the computer with the instruction set $\mathbb{H}$ processor.

We separately define the following three words for clear expression in the following parts.

- **Simulation**: The program is compiled into the binary file with the target instruction set $\mathbb{T}$. The instruction set simulator running on the host computer with instruction set $\mathbb{H}$ translates and executes the binary with $\mathbb{T}$.
- **Execution**: The program is compiled into the binary file with the target instruction set $\mathbb{T}$. In this case, we should have the condition that $\mathbb{T} = \mathbb{H}$ which means that the development platform and the target system share the same instruction set and the compiled software can be executed directly.
- **Emulation**: The program is also compiled into the binary file with the target instruction set $\mathbb{T}$ and we have the condition that $\mathbb{T} = \mathbb{H}$. In the emulation case, the binary file is executed by using the experimental board or the FPGA directly.

As software becomes much more important in today's MPSoC platforms, four software simulation technologies are proposed in Chapter 3 and we summarize them here:

- **Host execution**: This is the most direct simulation technology which compiles all software into the binary format of host machines. The simulation speed of target software can be as high as the speed of the directly host execution if all simulation and hardware modeling costs are excluded. But we should also notice that the host execution cannot provide any timing information which may be important for developers to verify the performance of target application software and MPSoC platforms.

- **Timing annotation**: To avoid the weak point of timing information with the host execution method, timing annotation technology is created to add small timing information into each basic block to improve the host execution method. There are several different realization ways of timing annotation discussed in related works. But most of these methods are still far from cycle accurate.

- **Instruction interpretation**: This method is the most traditional one which interprets target binary code at run-time. This method can model the whole instruction set execution, the register files and all peripherals. Some simulators model pipelines of processors to achieve very high accuracy. Simulation speed is the drawback of this method.

- **Dynamic binary translation**: To improve simulation speed, dynamic binary translation method can translate target binaries into binaries of the host machine. As this process is dynamic, self modification programs can also be well supported. Because the cache effect and some communication operations cannot be modeled with this method, the performance result accuracy is lower than the traditional instruction interpretation method.

In this thesis, because only software based simulation platforms are discussed, we do not deal with FPGA emulation methods. These methods require a very accurate description of the HW, outside of the scope of the current work.

4.1.3 Hardware modeling methods

To model hardware components, there are also several possible methods which are different in both accuracy and efficiency. In this thesis, we have three possible hardware modeling technologies:

- **Transaction Level Modeling**: which is a high-level approach to model digital systems where details of communication among modules are separated from the details of the implementation of functional units or of the communication architecture. All clock driving communication requests are replaced by functional calls to improve simulation performance. Though transaction level modeling, as such, does not produce accurate results, the simulation speed is very good and allows functional validation.

- **Cycle Approximate Bit Accurate Modeling**: which models the digital systems and guarantees all data transferred between modules are bit accurate. Internal hardware

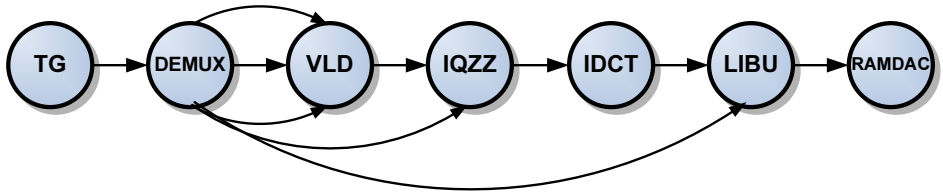

Figure 4.2: Motion-JPEG decoder application modeled with KPN

modules realization is only cycle approximate to balance both performance and accuracy.

- **Register Transfer Level Modeling**: which is commonly used to describe the operation of synchronous digital systems. Compared with the TLM modeling and the CABA modeling, RTL models both hardware modules and communication components with cycle and bit accuracy. This modeling is the most accurate one with the lowest performance among all these three modeling technologies. It usually allows producing the real hardware by automatic synthesis.

4.2 4 Abstraction Levels for Configurable Heterogeneous MPSoC

By combining multiple software programming interfaces, different software simulation and hardware modeling methods, 4 abstraction levels are proposed in paper [59] and used by this thesis for the configurable processor based heterogeneous MPSoC platform.

- System Level
- Virtual Architecture Level
- Transaction Accurate Level
- Cycle Approximate Level

4.2.1 System level

In this thesis, the *System Level* **(SL)** is the highest abstraction level used in our design flow. Because the key point in the MPSoC design is the application parallelism, this abstraction level is designed to provide a parallel execution environment. In Fig.4.2, the application is described as a set of communicating processes exchanging data exclusively through blocking lossless point-to-point *First In First Out* **(FIFO)** channels, known as *KPN*. This figure shows a KPN example for the Motion-JPEG decoder application. Beside the KPN computation model, we have a SDF one shown in Fig.4.3. Both of these computation models and many others can be supported at this system level for parallel software validation.

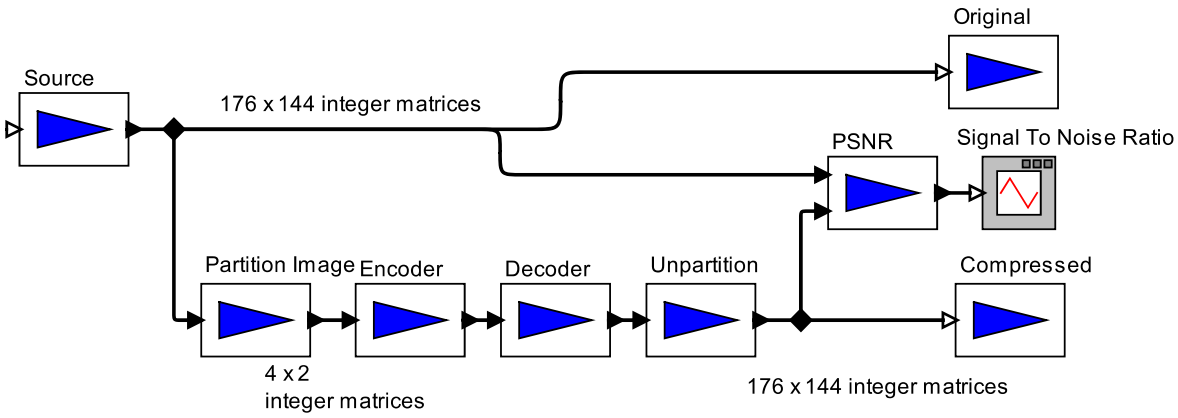

Figure 4.3: A simple vector quantizer for images from Ptolemy II demo. Author: Steve Neuendorffer.

At system level, there is no difference between software processes and hardware modules. All of them are general high level language code threads and communicate with each other with communication APIs. Several commonly used communication APIs are `channelInit`, `channelRead` and `channelWrite`. `ChannelInit` API is called by the main process to create a new channel for process communication. `ChannelRead` and `channelWrite` are used by processes to send and receive data. Here is the converter module of the Motion-JPEG example:

```
/* Use Channels for KPN communication */
void threadConv(Channel *from_idct, Channel *to_idct) {
  uint8    MCU_Y[64], MCU_Cr[64], MCU_Cb[64], MCU_RGB[64 * 3];
  int      R, G, B, Y;
  uint8    index, i, j;

  while (1) {
            /*-- Channel read operations --*/
    channelRead(from_idct, MCU_Y,sizeof(MCU_Y)/from_idct->cellSize);
    channelRead(from_idct, MCU_Cb,sizeof(MCU_Cb)/from_idct->cellSize);
    channelRead(from_idct, MCU_Cr,sizeof(MCU_Cr)/from_idct->cellSize);

              /*-- process computation --*/
    for (i = 0; i < 8; i++) {
      for (j = 0; j < 8; j++) {
        index = i * 8 + j;
        Y = (int)MCU_Y[index]<<SCALEBITS;
        R = (MCU_Cr[index] - 128) * c_RCr + Y + ONE_HALF;
        B = (MCU_Cb[index] - 128) * c_BCb + Y + ONE_HALF;
        G=Y-(MCU_Cb[index]-128)*c_GCb-(MCU_Cr[index]-128)*c_GCr+ONE_HALF;
        if(R < (1<<SCALEBITS)) R = 0; else if(R> (0xff<<SCALEBITS))
            R = 255; else R = R >> SCALEBITS;
```

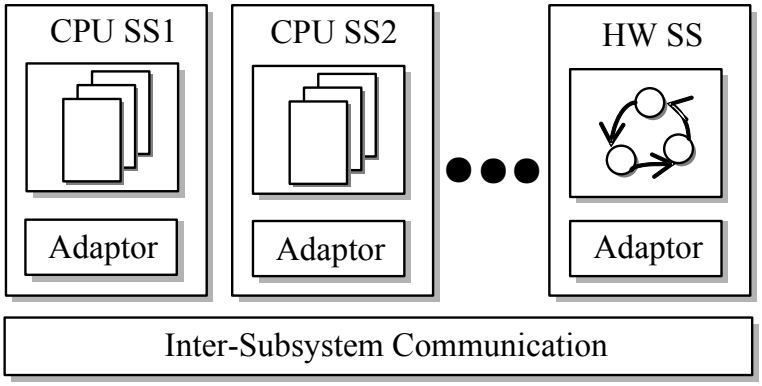

Figure 4.4: System Architecture at VA Level

```
        if(G < (1<<SCALEBITS)) G = 0; else if(G> (0xff<<SCALEBITS))
            G = 255; else G = G >> SCALEBITS;
        if(B < (1<<SCALEBITS)) B = 0; else if(B> (0xff<<SCALEBITS))
            B = 255; else B = B >> SCALEBITS;
        MCU_RGB[(i * 8 + j) * 3] = R;
        MCU_RGB[(i * 8 + j) * 3 + 1] = G;
        MCU_RGB[(i * 8 + j) * 3 + 2] = B;
      }
    }
             /*-- Channel write operations --*/
    channelWrite(to_libu, MCU_RGB,sizeof(MCU_RGB) / to_libu->cellSize);
  }
}
```

The converter process is a typical KPN process which gets data from other processes and send computed data back. From Table.4.1, the application is executed natively and no timing information is given for communications. The whole simulation platform is compiled to host machines and uses POSIX Thread APIs for thread operations. The channel communication is realized as a library with sending and receiving data calculation for communication optimization at system level. The implementation of these functions can for example be done on top of the Unix pipe system calls. The advantage of system level is to provide a fast high level validation environment for one computation model with communication profiling abilities.

4.2.2 Virtual Architecture (VA) level

The VA level is a more detailed abstraction level compared with the system level. The most important difference is that the VA level includes some architecture information. At this level, the architecture is coarse grain and abstracts the real system into *Subsystems* and *Inter-subsystem Communications*. Fig.4.4 presents a general VA architecture. In this figure, the system is composed with several subsystems which can be divided into *CPU Subsystems*

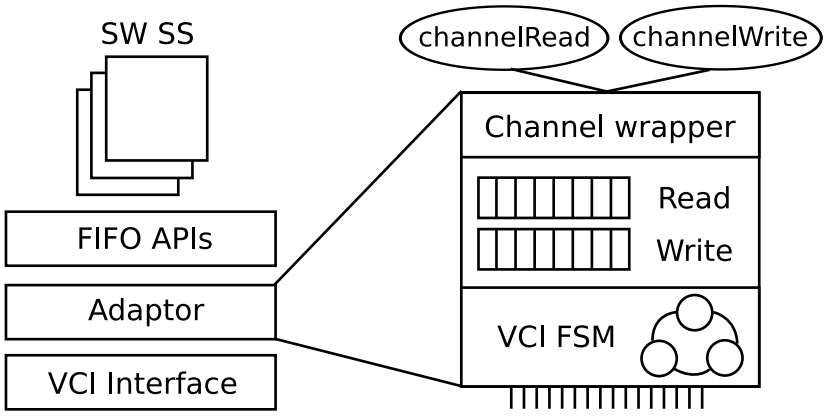

Figure 4.5: VA Adaptor Realization

(CPUSS) and *Hardware Subsystems* **(HWSS)**. All these subsystems are connected with each other by using the inter-subsystem communication module.

The aim of the VA level is to provide an execution model which can give more detailed inter-subsystem information with high simulation speed. From Table.4.1, we find all the subsystems are timeless models and compiled to host machines to significantly speed up system simulation. But the inter-subsystem communication can be cycle accurate or timed TLM to provide detailed timing information for communication exploration.

With all these requirements, we build the VA level platform based on SystemC. We replace the original timeless communication module at system level with the timed communication module, and we add adaptors to connect subsystems to the communication module with the *Virtual Component Interface* **(VCI)**[33]. The overall objective of VCI is to obtain a general interface, such that *Intellectual Property* **(IP)**, in the shapes of virtual components of any origin, can be connected to SoC of any chip integrator. In this manner, VCs are not limited to one-time usage by their designers. They can be re-used over and over. Fig.4.5 shows realization details of an adaptor which connects the high level FIFO interface with this VCI interface. The upper part of this adaptor is the channel wrapper which transfer read/write commands into buffer operations. By using data of the read and the write buffer, the VCI *Finite State Machine* **(FSM)** can communicate with the inter-subsystem communication module with the VCI interface.

4.2.3 Transaction Accurate (TA) level

TA level is created for subsystem internal modeling with timing information. As shown in the left side of Fig.4.6, one general software subsystem includes a multi-threaded application, an operating system, CPUs, memories, inter-subsystem communications and so on. At the right side of Fig.4.6, the TA level targets at modeling all these internal components with TLM to speed up simulation at the cost of accuracy. The loss of accuracy has to be quantified by experiments.

The TA execution model has to handle two different interfaces: one at the software side using HAL APIs or POSIX Thread APIs and one at the hardware side using hardware pro-

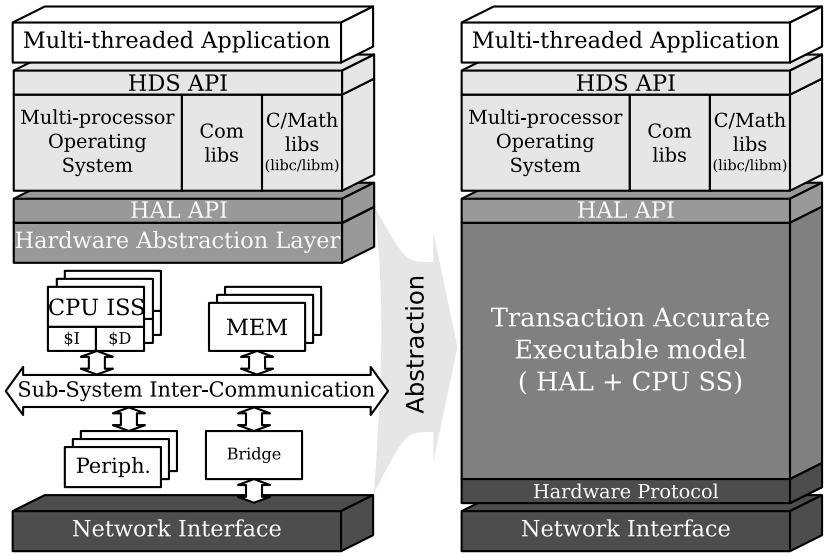

Figure 4.6: TA Level Abstraction with the HAL APIs Interface

tocols as shown in Fig.4.6. Over the HAL APIs, the operating system, the specific I/O communication library and the multi-thread application software are added. With the same HAL APIs, the effort of rewriting low level assembly programs for exploring different CPUs and extended instructions sets is saved. Because upper operating system and application software are timing annotated and executed directly with the TA level model, the MPSoC simulation speed is higher than traditional ISS based virtual platforms.

At TA level, software and hardware components of one MPSoC are easy to integrate for an executable platform with high simulation speed and relatively accurate performance estimates [45]. All these properties make TA level a suitable abstraction level for the CH-MPSoC architecture exploration.

4.2.4 Cycle Approximate (CA) level

Cycle Approximate **CA** level is the most detailed abstraction level in our multiple abstraction levels design space exploration flow. The target of this abstraction level is to provide highly accurate simulation results at the cost of a low simulation speed. That means almost all detailed architecture information should appear in this simulation model. Fig.4.7 gives out a general model of this architecture.

At the hardware part, we have cycle approximate *Instruction Set Simulator* **ISS** for software execution. With the ISS, we have the cache model with cache size, cache line size and associativity parameters. As the software programming interface of the CA level is processor binary format, all HdS and applications should be compiled into target binaries. To connect each processor, we propose two possible connection methods which are bit accurate SystemC connection modeling and *Transaction Level Modeling* **(TLM)**. Hardware components should be implemented with SystemC modules to provide accurate timing information during simulation processes.

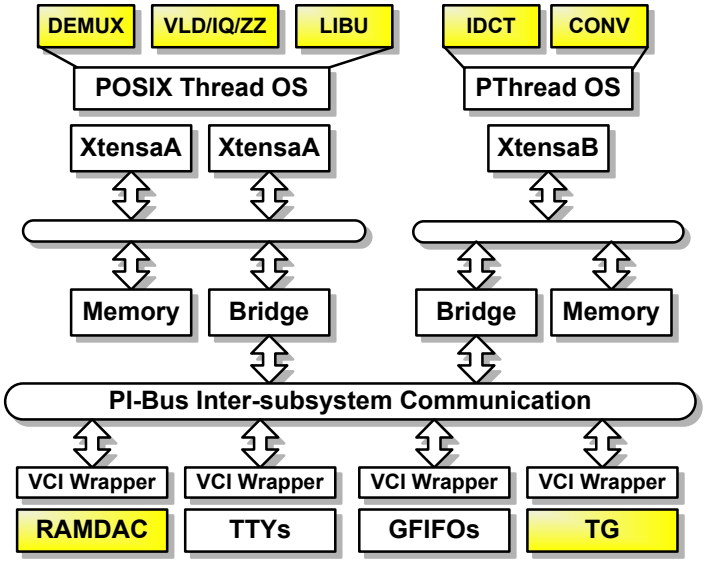

Figure 4.7: Heterogeneous MPSoC Software/Hardware Architectures.

4.2.5 Comparison of abstraction levels

Table.4.1 summarizes this chapter and presents the comparison of the four abstraction levels. This table is also useful for the multi-abstraction level design space exploration flow presented following.

4.3 Budget Based Exploration Flow

Budget generally refers to a list of all planned expenses and revenues. In the logic synthesis domain, the timing budget is used to select suitable logic components for predefined timing requirements [46]. In DSE flow, budget means given resources each module or subsystem can use to build the overall system. For example, the power consumption is an important parameter for most MPSoC architectures. During the DSE flow, the power budget for the whole chip will be divided into small budgets for each subsystem and each module. With this refinement process, details of each subsystem and each module can be fixed at the end of this DSE flow. Though budget is supposed to work on additive constraints, this property will unfortunately not always be true. For example, adding bandwidth requirement leads to exponential growth of delay with some kinds of communication architectures.

This budget based exploration idea is not new for system level design space exploration [71][96][38]. As this approach is in practice efficient and compatible with the multiple abstraction levels we proposed, in following parts, we will give details of this budget based DSE flow.

4.3.1 Refine Budgets with Multiple Abstraction Levels

Our multi-abstraction levels exploration flow follows the budget based problem partitioning approach. Fig.4.8 shows the whole flow from the most abstract level (SL) to the most de-

Table 4.1: Comparison of Different Abstraction Levels

	System level	Virtual architecture level	Transaction accurate level	Cycle approximate level
Software tasks	C/C++ code	C/C++ code	C/C++ code	C/C++ code
Software Programming interface	Communication APIs	Communication APIs	POSIX Thread APIs or Hardware Abstraction Layer APIs	Processor Binary Format
Software Simulation Tech	Execution natively	Execution natively	Dynamic binary translation	Run-time Instruction interpretation
Hardware components	Host threads	SC_Thread	SystemC modules	SystemC modules
Hardware communication interface	Communication APIs	Communication APIs	TLM or VCI	TLM or VCI
MPSoC architecture	Not yet defined	Inter-subsystem communication	Subsystem details	Fixed instruction sets
Performance accuracy	No timing information	Inter-subsystem Communication	Cycle approximate	Cycle approximate

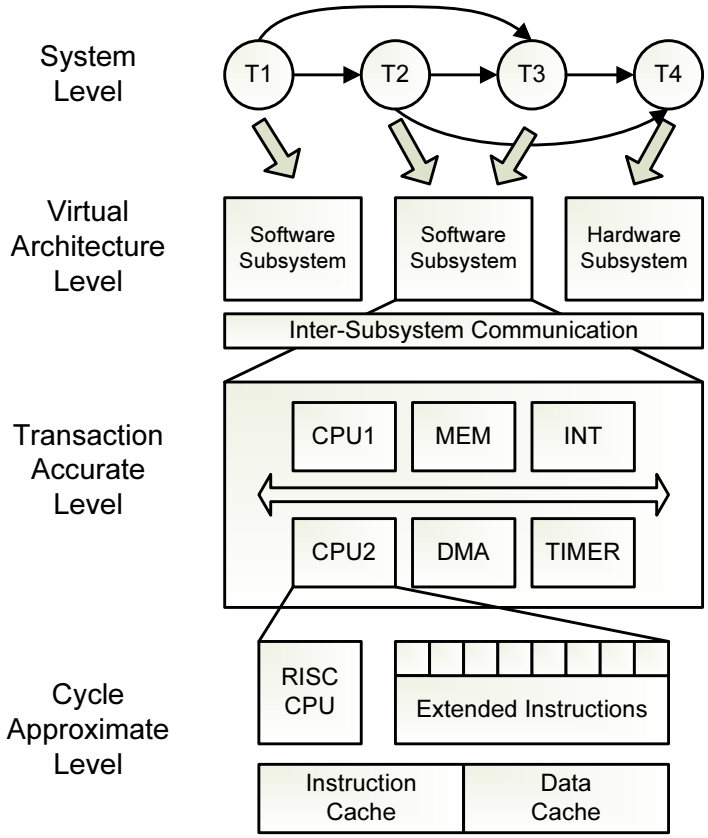

Figure 4.8: Multi-Abstraction Levels Design Space Exploration Flow

tailed level (CA). At system level, designers optimize computation models such as KPN or SDF for communication related properties. By using executable environments provided by SL models, designers can minimize the inter-task communication size and optimize communication properties to improve system performance. At VA level, designers create basic multi-subsystem architecture and map all processes of SL into each subsystem. After exploring the inter-subsystem communication with high simulation speed, designers can get the communication size and transfer pattern of the inter-subsystem communication. By using these statistical data, designers can fix inter-subsystem communication type selecting from bus, crossbar and NoC and get the usage rate of this communication module.

After the inter-subsystem communication is well optimized at VA level, we continue exploring each subsystem at TA level. Because a general software subsystem includes CPUs, memories, buses and other modules, the architecture is more complex than that of the hardware subsystem. At TA level, all complicated modules in one *SoftWare SubSystem* **(SWSS)** are modeled and composed together. The software performance data is based on the back annotation from CA level with some statistical parameters. The simulation results provided at this level help designers to direct the automatic generation of configurable processors and modify hardware peripherals of one subsystem. After all components are fixed for each subsystem, we can automatically generate configurable processors and modify cache size to meet TA level constraints. As CA level is the most detailed abstraction level, all hardware parameters of this configurable heterogeneous MPSoC solution are fixed and HAL APIs are

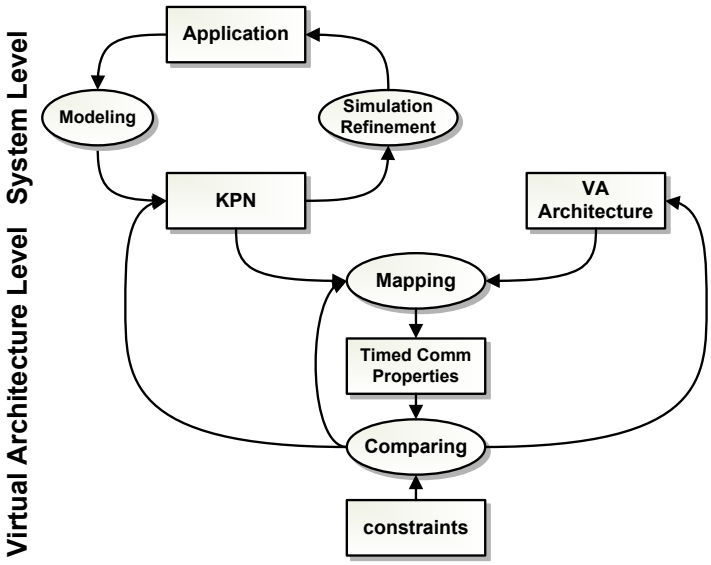

Figure 4.9: The System Level and Virtual Architecture Level Communication Exploration Flow

carefully realized based on instruction sets of final processors. The whole exploration flow is finished when a CA model is obtained to meet the budget constraints.

Summarily, our budget based design space exploration flow can handle the following refinement processes with 4 MPSoC abstraction levels.

- Communication refinement at *System Level* **(SL)**.
- Subsystem refinement at *Virtual Architecture* **(VA)** level.
- Internal subsystem refinement at *Transaction Accurate* **(TA)** level.
- CPU extended instruction and cache refinement at *Cycle Approximate* **(CA)** level.

4.3.2 Communication refinement at system level

The *System Level* is the highest abstraction level in our design flow. Because the key point in the MPSoC design is the application parallelism, this abstraction level is designed to provide a parallel execution environment. At system level, there is no difference between software and hardware. From Table.4.1, the application is executed natively and no timing information is given for communications.

There are several advantages of exploring the communication architecture at this level. Though compared with more detailed abstraction levels, application tasks at system level are timeless, the functional simulation can give out the accurate number of transfer and the data size of each transfer. The information can be used to optimize the communication behavior and estimate the required bandwidth for real time requirements. We should notice that the simulation speed at system level is much higher than that of more detailed levels. This advantage is really useful for the communication exploration of complex applications.

The system level exploration flow is shown in Fig.4.9. Square blocks represent exploration statuses while ellipses represent actions. From this exploration flow, we first model the original application into, for example, a KPN. The system level simulation model is based on KPN and relies on existing tasks and FIFO libraries. For communication exploration requirements, we design a specific channel module which captures the number of transfers and transfer sizes. After each channel in the KPN model is implemented, we obtain the executable system with the detailed transfer burst size and transfer numbers. The statistical communication information can be displayed in figures which designers can use for bus usage optimization and latency minimization. By modifying the application source code and the KPN modeling, communications can be optimized at this level. The data shown in experimental section gives detailed information of the system level exploration. Because the simulation speed is really high, design loops at this level are relatively low-cost.

4.3.3 Subsystem definition at VA level

After communications are optimized at system level, we minimize the communication transfer number and utilize the communication burst mode. At VA level, we continue exploring communications by fixing the type and protocol of the inter-subsystem communication. For communication types, we can choose one type from {bus, crossbar and Network-On-Chip}. After we fix the communication type as bus, we may continue to select one specific protocol from several kinds of buses {APB, PI and so on}.

These exploration works are divided into two steps. The first step is to map application tasks into subsystems. Then we can simulate the whole system at VA level and get inter-subsystem communication performance results. If results are not compatible with system constraints, designers should change the tasks mapping based on inter-subsystem communication performance results and choose another communication type and protocol. The exploration flow is shown in Fig.4.9 and we give details of these two methods separately.

Because computation and communication parameters are related, task mapping becomes really complex. At RTL/binary level, it is a time consuming process to simulate and find a suitable mapping solution. At VA level, there is no computation information because all CPUSS and HWSS are timeless. The designer needs only to take into account the communication data during the mapping process. With a specific number of subsystems, we map application tasks from KPN to these subsystems based on the inter-subsystem communication. So a good mapping solution is the one uses all subsystems and requires less inter-subsystem communication. This is the guideline for the VA level communication mapping.

Concerning computation requirements, we satisfy them at the next abstraction level. The performance of one CPUSS can be improved by increasing the number of CPUs, changing the CPU instruction set and adding co-processors. These computation refinement approaches are discussed in TA and CA levels of the DSE processes.

One obvious drawback of this method is that we can only get the overall data communication size of the target communication architecture. As there is no computation related timing information, all send/receive communication data is taken into average during the ex-

ecution. This assumption may cause some inaccuracy for this VA level simulation platform and it can be corrected at TA can CA levels.

4.3.4 TA level subsystem exploration

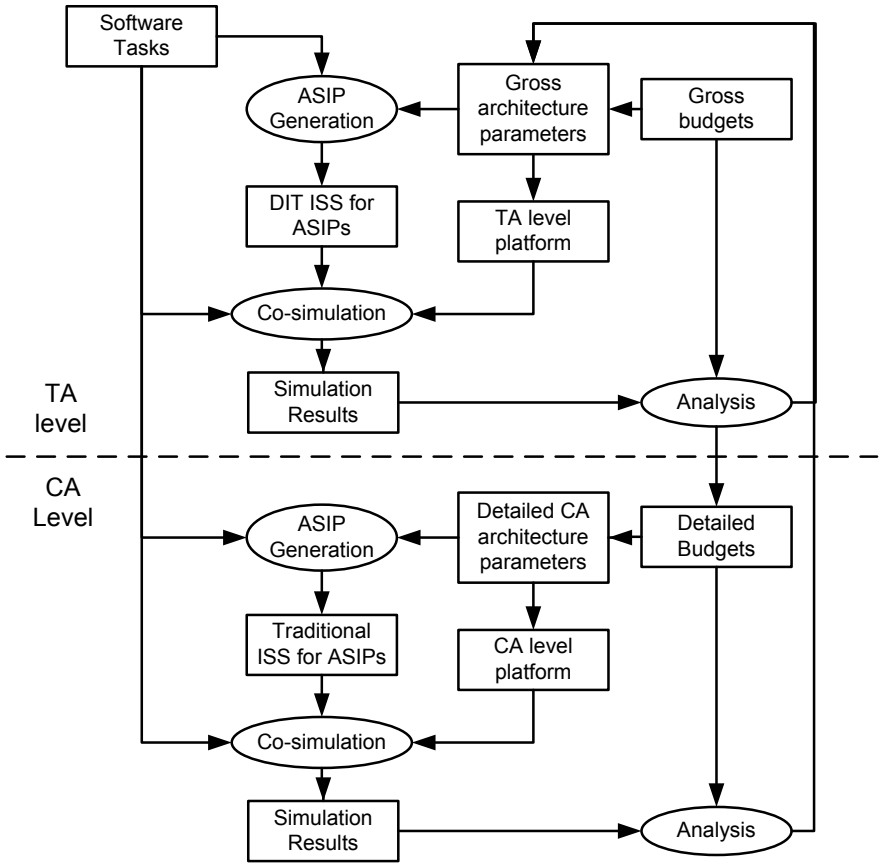

Figure 4.10: TA Level Exploration Flow

With the hardware architecture and either native realization [45] the dynamic binary translation simulator [30] for subsystems, the TA level becomes a suitable platform for design space exploration. Fig.4.10 presents the exploration flow for the TA level and the CA level. We can see that the TA level exploration flow extends the classical Y-chart[87] flow with separated budgets and ASIP generation. These parts help this design flow become suitable for the design space exploration of configurable heterogeneous MPSoC based architectures.

Gross budgets used in this flow are derived from upper-layer exploration results. They may include performance constraints, chip size (cost) constraints and power constraints for this subsystem. With gross budgets, we can get gross architecture parameters for TA level simulation. After we get TA level simulation results, we can refine gross budgets into detailed ones for each component. With the budgets refinement process, the designers make progress in the exploration flow.

Dynamic binary translation method is also important for this exploration flow. At TA

level, the application software and HdS are simulated with this method. Though there are no cache and instruction execution dependency information in this method, this TA level platform has relative fast speed and is sensitive to extended instructions. These properties can help designers optimize both MPSoC architectures and application software to meet user defined budgets. For extended instructions, we can use the ASIP automatic generation tools XPRES[3] to generate a new processor with constraints of budgets. If designers want to optimize the extended instruction set directly, the extended instruction set can be used at CA level directly. So the optimization effort is retained during this flow.

4.3.5 Fix all details at CA level

With the MPSoC architecture parameters and extended instruction set fixed at CA level, the MPSoC simulation platform can provide accurate performance results with relatively low simulation speed.

Table 4.2: Parameters: TA to CA

	Cache	Performance	Subsystem communication
TA	• Miss rate	• Execution time	• Throughput • Latency
CA	• Cache size • Line size • Cache associativity	• Extended instruction set • Frequency	• Bus/crossbar/NoC • Protocol: AHB/PI-Bus • Frequency

As the design space exploration process is a parameters and budgets refinement process, we can, at CA level, continue refining some abstract parameters of TA level. Table 4.2 shows several examples of this process. At TA level, we only have the `Miss rate` parameter for the whole simulation and we need to refine it into `Cache size`, `Line size` and `Cache Associativity` at CA level. It is the same case for *Performance* and *Communication*. Statistical parameters used at TA level can save much computation and adaptation effort.

4.4 Conclusion

In the past, there have been much abstraction level definition research works and only a few of them are still in use now. Because these abstraction levels should relate application software, system software and hardware, modification of these aspects will make the abstraction level definition lose its interests. Underlying simulation platform implementation technologies may also change with the time. These four abstraction levels described in this chapter try to provide meaningful information for current requirements of application software developers, hardware dependent software developers and hardware developers.

By integrating software programming interfaces, hardware modeling technology and software simulation technology, we can build an efficient and fast multi-abstraction level design space exploration flow. With the budget based refinement methodology, many huge design loop cases become smaller ones between consequence abstraction levels. Though the

detailed implementation of this methodology is still dependent on real applications, the use of multiple abstraction levels is a way to help designers reach their target design easier.

Chapter 5

Hardware/Software Interface Modeling and Hybrid MPSoC Simulation Platform

Contents

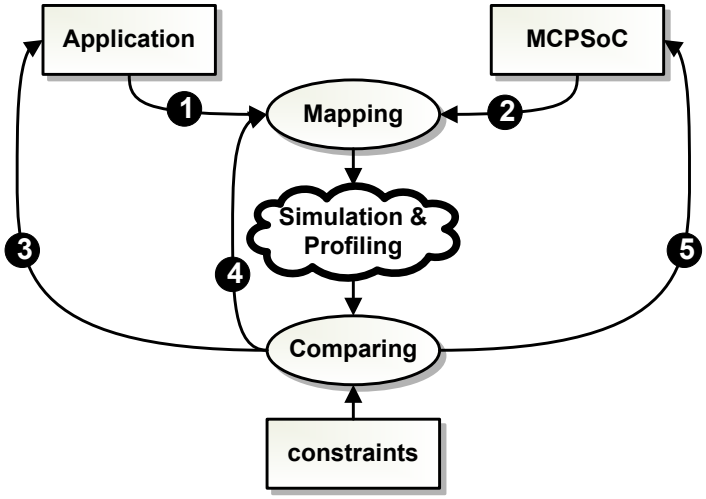

Figure 5.1: Design Space Exploration Flow with Configurable Processors.
1 and 2. Mapping applications on the configurable MPSoC architecture. 3. Refine application to satisfy predefined constraints. 4. Refine mapping to satisfy predefined constraints. 5. Refine both the processor configuration and MPSoC architecture to satisfy predefined constraints.

As presented in chapter 4, the HW/SW interface modeling and adaptation become very complex because of the configurable processors and the heterogeneous multiprocessor architectures. As the HW/SW interface becomes a bottleneck of the whole design flow, this chapter proposes a modeling method and an automatic generation flow that help solve this problem. Meanwhile, we also propose an hybrid MPSoC simulation platform which can realize HAL APIs and I/O device drivers with SystemC to avoid much adaptation efforts for design space exploration flow discussed in chapter 4.

In this chapter, we try to answer the following questions:

- How to model HdS to facilitate the automatic generation and support complex architectures such as the configurable processor based heterogeneous MPSoC platform?

- How to build high level simulation models with high level programming APIs to avoid HdS porting works?

5.1 Motivation

Configurable processors, being nowadays widely available, it is a complex process to find a suitable architecture for user defined requirements. In the embedded system domain, this process is generally called *Design Space Exploration* **(DSE)**. In Fig.5.1, a Y-chart [87] liked exploration flow is given. We believe that the *Simulation* and *Profiling* processes gradually become the real bottleneck of the whole exploration flow because of the SW adaptation requirement. Three kinds of modifications cause much HdS porting work:

- **Processor Configurations**: To improve the performance evaluated during the DSE process, the configuration of processor may need to be modified by adding and removing some instructions/registers. Besides that, the addition or removal of exception causes, traps, interrupt numbers and priority may also impact on the HAL realization of the HdS.
- **MPSoC Architectures**: Beside processor level changes, the modification of MPSoC architectures may also change the I/O mapping, the processors number and the processors connection topology. These changes need modifications of HdS accordingly.
- **Hardware modules realization and interfaces**: During the DSE flow, both the realization and the interface of hardware modules may need to be modified according to performance and architecture requirements. This modification may require modifying the driver at the same time because of register mapping and function specification changes.

Without well adapted system software, it is impossible to simulate the application and profile the performance data in a reasonable time frame. Meanwhile, normally some parts of these components are realized with assembly code directly which may need to be partly re-implemented when the instruction set is changed.

As the automatic generation method could be a possible solution for the system software adaptation, in this chapter, we try to solve the problem of HW/SW interface modeling which is regarded as one of the bottlenecks of the automatic generation process. With a well defined HW/SW interface modeling, the generated interface can be flexible for different kinds of applications and have a good balance between cost and system performance.

Beside the automatic generation methods, we have created a hybrid simulation platform which can realize most of HAL APIs with SystemC codes to avoid much adaptation work. Though there are already several hybrid simulation platform research works in Chapter 3, we use the similar *Semi-Hosting* functional call technology for simulators, but the aim of our hybrid simulation platform is to avoid OS porting and modification works at early DSE stages of the CMPSoC development. Though the system software is important for the whole DSE flow and time consuming for development, it does not really change much performance of the whole system. Because normally the system software does not perform heavy computations and consumes much less time than application codes, we can conceal the internal realization of these functions without a large impact on the simulation result.

To the best of our knowledge, our hybrid simulation platform is the first work which uses semi-hosting functional calls for HAL APIs and drivers realization to simplify OS adaptation and speed up the configurable MPSoC DSE process. Details of our hybrid simulation platform are given in following sections.

5.2 Hardware/Software Interface Modeling

In section 2, we conclude that there is no strong mathematic background for the concurrent and detailed hardware/software system level synthesis. To support automatic HW/SW inter-

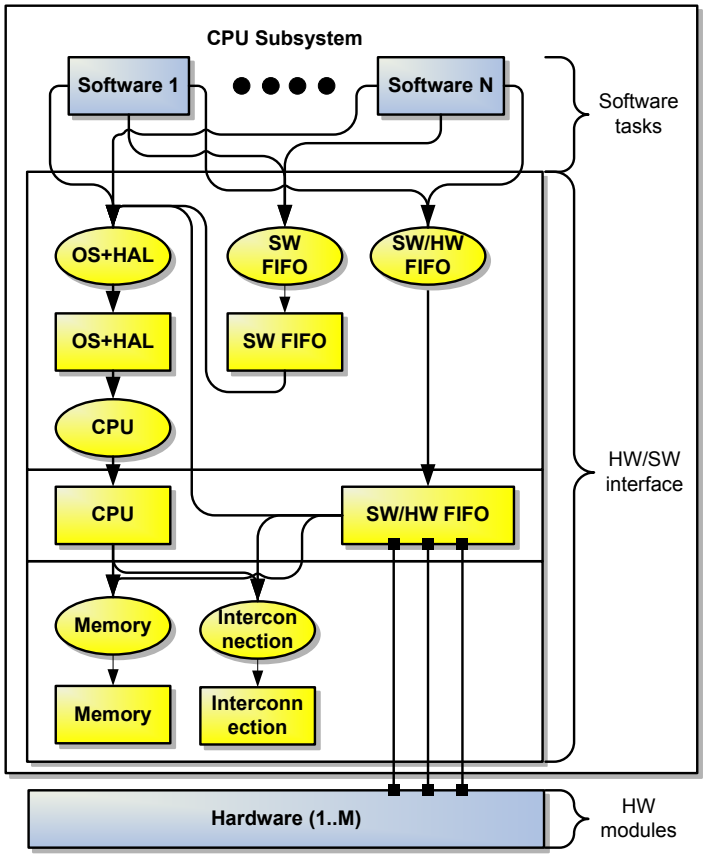

Figure 5.2: Components in SDG

face generation, we need to find a method to describe relationships between all components used in one MPSoC. The *service* ideas are commonly used in software community for enterprise architectures. In that context, the term service refers to discretely defined sets of contiguous and autonomous business or technical functionalities. We extended this definition and use the service idea for all functional components used in MPSoC architecture. By using this service idea, both software and hardware components used in one MPSoC platform can be modeled as a service object. All these service objects can be put into a library for automatic generation tools. Service relationship among all these service objects can be used for the system level synthesis.

Interface and implementation

Actually, the service objects can be grouped into either *Interface Components* or *Implementation Components*.

- **Interface components** are designed elements that present the exported information of the services. By using the same idea of interface as in Java and other object oriented languages, interface components abstract information about an exported interface without any realization details. For the software service, each interface only includes the *function name*, *parameters* and *return type*. As for the hardware service,

each one includes the *port name* and *signal information*. In the diagram representation, the interface components are represented with an ellipse and the function name noted inside. As show in the example SDG of Fig.5.2, services of the HW/SW interface include the CPU service, the OS service and so on. Because there are 80 services in the SDG of this case study and we want to show the information clearly, we pack the services into several groups, such as OS related services and local FIFO related services.

- **Implementation components** are the second kind of component which realizes the interfaces of the SDG. For every interface component in the SDG, there should be at least one implementation component to realize it. More than one implementation of a specific interface component is encouraged. The advantage is to provide the support for architecture exploration to meet the different performance and resource requirements. With the same exported APIs in one service library, the SDG gives architects more freedom to explore the system architecture without any modification of existing application codes. For the diagram representation, implementation components are represented as rectangles. The OS and CPU implementation components are shown with rectangle in Fig.5.2.

5.2.1 Service Dependent Graph (SDG)

An SDG is a graph which vertices are *Service Objects* and arcs are *Service Dependencies*. A *Service Dependency* is the connection between one interface component and one implementation component. The connection has two slightly different meanings. When it originates from an interface component and point to an implementation component, it means that the interface component needs a realization. If it goes from an implementation component to an interface component, it means that the implementation requires the specific API to finish its realization. In Fig.5.2, the *requirement dependency* is represented with an arrowed line while the *realization dependency* is represented with a line.

All the information relative to the service components and the service dependencies will be used to build a service library. Using the service library, an SDG tool could construct a suitable SDG automatically for a specified HW/SW interface. By using this suitable SDG, the simulation model could also be generated automatically.

5.2.2 Automatic Generation Process Definition

As we use the SDG to model the HW/SW interface, it is possible to generate it automatically. *Service libraries* and *constraint files* are the crucial parts for this process. The service libraries contain service interface components and implementation components. In each library, there may be several implementation components that realize one interface with different performance properties and at different abstraction levels. The constraint files include system requirements such as the chip size requirements, the software code size requirements

and timing/performance requirements. Given service libraries and constraint files, the SDG can be generated for the HW/SW interface automatically with specific exported APIs.

The generation process can be divided into several steps.

- The architects choose the exported APIs and give out the constraint files.
- Build the SDG by traversing the service library and extracting the paths that implement the APIs.
- Generate the simulation models (for hardware) and real executable code of HW/SW interface (for software) by using the SDG model.
- Get the performance results. If the results are not satisfied, the designers should change the constraint files and run the process once again.

5.2.3 Constraints and limitation

Though this service based automatic generation process can handle all modules for HW/SW interfaces, this process still has the limitation of performance estimation. As the relationship between application software and processors are not easy to model analytically, simulation becomes the only solution for MPSoC performance estimation. Because it is still time-consuming to simulate a group of applications for a generated MPSoC architecture, until now, this system level automatic generation process is still far from the RTL level synthesis process.

We have modeled a small OS kernel using this approach which leads to a very complex dependency graph (shown in the chapter 7). As the complexity grows with the OS and hardware components, this method is still far from real usage.

5.3 Hybrid simulation platform

To simplify the configurable MPSoC design space exploration flow, especially to separate the operating system adaptation process to the architecture and application optimization, we create a new hybrid simulation platform which can provide much higher simulation speed, accurate enough performance results without having to dig into many unrelated implementation details.

In both Fig.5.3 and Fig.5.4, we have the traditional simulation platforms at the left, while at the right side, we propose new hybrid ones. We have the hybrid simulation platform with PThread OS APIs at the right of Fig.5.3. In this figure, we provide the same POSIX Thread OS APIs to multi-threaded applications, so there is no difference for the high level application development. Meanwhile, at the right of Fig.5.4, we show the hybrid simulation platform with the HAL programming interface. In this platform, we share the same operating system, communication library, C library and applications as traditional ones. In both figures, the most important difference is that this hybrid simulation platform replaces the

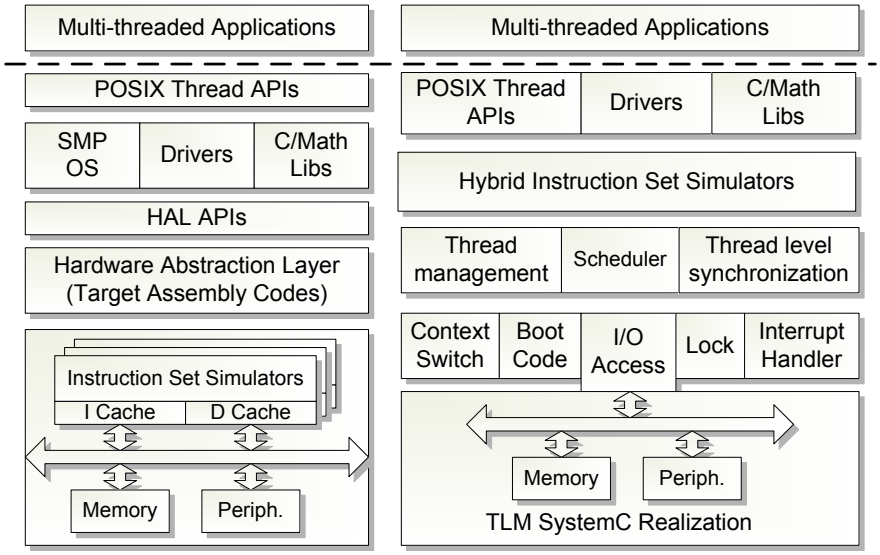

Figure 5.3: Hybrid Simulation MPSoC Simulation Platform.
Left: the traditional ISS-based simulation platform. Right: the hybrid simulation with the local POSIX Thread OS execution.

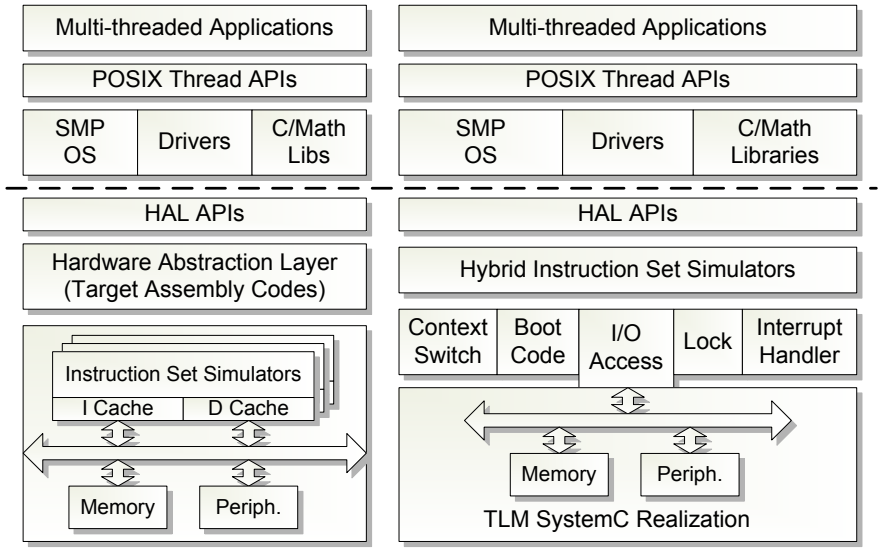

Figure 5.4: Hybrid Simulation MPSoC Simulation Platform.
Left: the traditional ISS-based simulation platform. Right: the hybrid simulation with the local HAL execution.

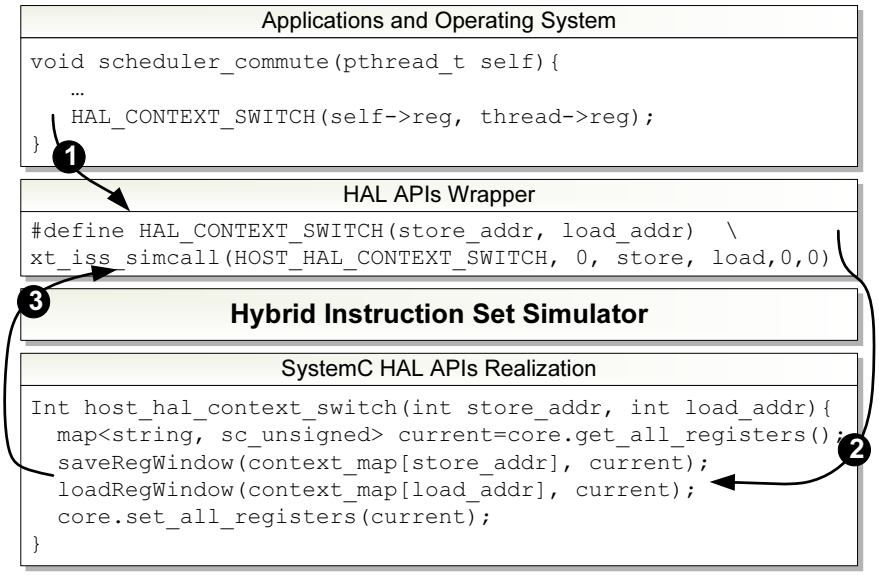

Figure 5.5: 3 Step of Semi-Hosting HAL Function Call.
1. The OS calls the HAL API: HAL_CONTEXT_SWITCH which transfers to the semi-hosting call xt_iss_simcall (this functional call will be compiled into a special trap instruction). 2. When this trap instruction is executed, the ISS is bypassed and the corresponding functional call of a host SystemC module is invoked. 3. The host SystemC module finishes the task and returns the control right.

HAL assembly codes realization or PThread OS implementation on the ISS with SystemC modules realization on the host machine.

Technically, we need to have a method to let the HAL or PThread OS APIs bypass the ISS and reach the underlying SystemC modules. In our hybrid simulation platform, we propose the usage of a specific *Semi-Hosting Interface* [3] for instruction set simulators. The semi-hosting interface can be realized with a specific instruction which can be used only in the simulation process. When the ISS fetches this instruction, instead of decoding and executing it directly, the ISS calls a specific host hooking function to process it. Hooking is a technique employing so-called hooks to make a chain of procedures as an event handler. After the handled event occurs, control flow follows the chain in specific order. Traditionally, most simulators use this way to realize some system function calls which are not realized in target embedded environments such as SPIM MIPS simulator [21] and so on. In our platform, this hook function invokes the corresponding host module which can modify the processor status and memory data. After the host module returns, the embedded application can have the same execution flow as the traditional realization.

In following experiments, we use the Tensilica Xtensa XTSC environment [3], Xtensa simulator has a specific `SIMCALL` instruction which can directly call hook functions for all HAL APIs and PThread OS APIs accesses. Fig.5.5 shows details of this method with the `HAL_CONTEXT_SWITCH` API. We have built SystemC modules to realize most of these APIs on the host machine directly. Realization details will be discussed in the next section.

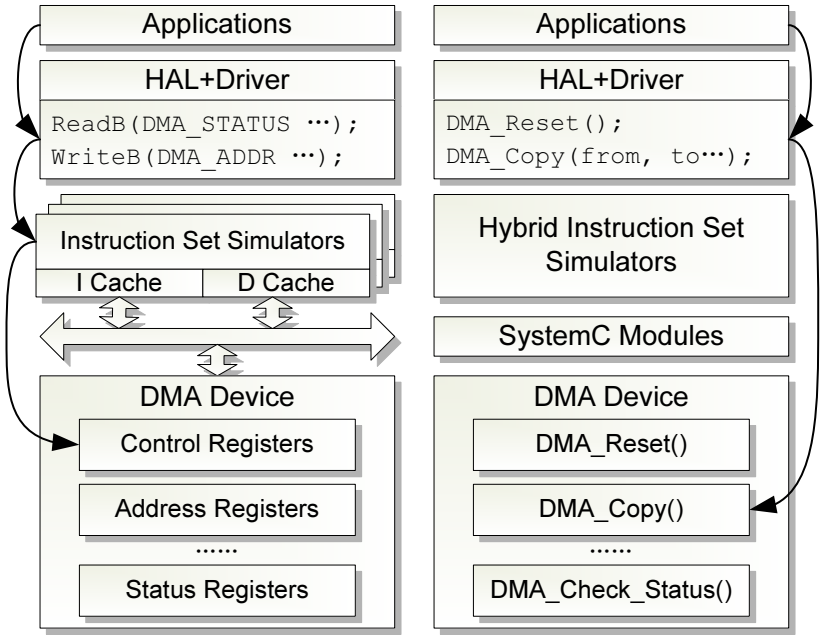

Figure 5.6: Difference between traditional device realization and TLM realization. Left: Traditional realization with access memory mapped I/O registers. Right: TLM DMA with directly functional calls.

5.3.1 Simulation of peripherals by Transaction-Level Modeling (TLM)

Transaction-Level Modeling **(TLM)** is a high-level approach to modeling digital systems where details of communication among modules are separated from the details of the implementation of functional units or of the communication architecture. As existing TLM 2.0 Draft 2 [14] only focuses on the communication architecture modeling, the method to model and integrate peripherals with operating systems and applications is still an open problem. Traditionally, this integration process requires well defined control registers and data registers specification for each device. All operations between the OS and the device should follow this specification and be triggered by a series of I/O registers accesses. As the design space exploration process does not mainly target to validate drivers and HAL APIs, we abstract these details using a much easier SystemC functional call approach.

In Fig.5.6, we show these two different realization methods. On the left, we demonstrate the traditional register based modeling technology for the definition of one DMA device. A group of control and data registers is defined for the driver realization. On the right, we propose a more proper method in our context which does not rely on memory mapped registers at all. This new method provides only a series of function calls which can be directly integrated with the *Device Drivers* of the HAL level. By using the same *Semi-Hosting* interface, we can bypass the hybrid ISS and directly access these functional calls. This method can simplify traditionally complex device access protocols and avoid most I/O address decoder errors. For high level application developers, this hardware modeling and access method can make the system validation process much easier and speedup the whole system simulation.

5.4 Operating System (OS) APIs realization details

In this part, we show details of POSIX Thread OS APIs implementation with SystemC modules directly. SystemC is not only a language but also contains a simulator which allows to handle high level process operations and scheduling. Following three points take these advantages of SystemC and detailed discussed here:

5.4.1 Thread manipulation

Traditionally, thread manipulation functions are realized with C programs and should well handle internal data structure and resources of each thread. For example, *attribute* should be implemented in each POSIX Thread system to store the information of each thread. In the PThread standard, `pthread_attr_init` and `pthread_attr_destroy` are used to handle the attribute structure for each thread. While `pthread_attr_setschedparam`, `pthread_attr_setstackaddr`, `pthread_attr_setstacksize` and so on can change the thread statuses by modifying attribute structures. With our SystemC realization, the thread manipulation functions become more concise. The attribute structure used traditionally is transformed into a C++ class compiled to host machines directly. As most concurrency and race conditions can be solved by the SystemC framework, this realization can much simply avoid improper operation of thread attributes.

5.4.2 Thread level synchronization

The POSIX Thread APIs mainly includes three synchronization methods which are *Mutex*, *Condition* and *Spin lock*. We briefly introduce them and show how to concisely realize them with SystemC modules.

- **Mutual Exclusion**: which can be abbreviated as *Mutex*. In the POSIX standard, PThread APIs include several ones as `pthread_mutex_init`, `pthread_mutex_lock`, `pthread_mutex_trylock` and `pthread_mutex_unlock`. These APIs are used to avoid simultaneous access of critical sections.

- **Condition Variables**: which allow threads to synchronize based upon the actual value of data. PThread APIs include several related functions which are `cond_init`, `cond_broadcast`, `cond_signal` and `cond_wait`.

- **Spin Lock**: which can synchronize access operations for shared resources. PThread APIs include several such functions as `pthread_spin_init`, `pthread_spin_lock`, `pthread_spin_trylock` and `pthread_spin_unlock`.

For Mutex, condition variable and spin lock, the host can realize them with SystemC modules and use the scheduler module for thread swapping functions if necessary.

5.4.3 Scheduler

The scheduler is the kernel of an operating system. Different scheduling algorithms make the OS suitable for different user environments. It is much easier and more flexible to realize the scheduler with SystemC modules on the host machine directly. For example, the *First-In First-Out* scheduling algorithm can be implemented with `std::list`. The *Priority based multiple queues* scheduling algorithm can also be realized with multiple `std::list`. As there are no race conditions during the scheduling process, these implementations do not require any complex lock mechanism.

5.5 Hardware Abstraction Layer (HAL) APIs realization details

In this part, we give details about how to realize these HAL APIs compatible with the eCos HAL standard [17] by using SystemC modules directly.

5.5.1 Checking and Modifying Processor Statuses

To realize all these HAL APIs from the host platform, we need to grant host HAL SystemC modules the right to access and modify internal processor registers. In our hybrid simulation platform, Instruction Set Simulators should provide this kind of *get* and *set* functions such as `get_all_registers` and `set_multiple_registers` provided by the Xtensa ISS. Beside the processor data, control and status registers access, host HAL SystemC modules should also have the right to read and write the host memory which is specific for the peripheral modeling. In the Xtensa XTSC environment, `peek_physical` and `poke_physical` functions are used to do this work. As both processor status and memories are under control of the host HAL realization, theoretically, the host HAL realization can do the same work as the target assembly HAL realization but with higher speed and more flexible.

5.5.2 Context Switch

A *Context Switch* is the computing process of storing and restoring the state (context) of a processor such that multiple processes can share a single processor resource. This context switch is an essential feature of multitasking operating systems. In our platform, the context switch related HAL APIs are `HAL_SAVE_REGISTERS`, `HAL_CONTEXT_INIT`, `HAL_CONTEXT_SWITCH` and `HAL_CONTEXT_LOAD`. As all these APIs are heavily processor related, the kernel part of this function can only be realized with assembly code. During the configurable processors design space exploration flow, the modification of instruction set and register file may affect the realization of all these context related APIs. As in our IDCT example code, we need to modify the context switch realization to handle the

new address register. As simulators provide the function to access all registers, the realization SystemC code is concise and robust.

5.5.3 Synchronization Mechanism

Synchronization is normally used to make the coordination of events for one operating system. `HAL_SPIN_LOCK` and `HAL_SPIN_UNLOCK` are synchronization related HAL APIs used in our hybrid simulation platform. Generally, a spinlock is a lock where the thread simply waits in a loop repeatedly checking until the lock becomes available. The processor needs to perform a busy waiting operation when the lock is not available. For real execution situation, this solution is necessary and unavoidable. But for simulation, this busy waiting realization wastes much simulation time, cannot provide any useful information, and may lead to deadlock of simulation.

In our hybrid simulation platform, we can use *Host Event* to realize the same spinlock function with much lower cost. `sc_event` is a mechanism used for the synchronization among SystemC modules. When a thread wants to obtain a lock, it should check the lock first. If it is not available immediately, our HAL API realization will wait for a specific event. This event will be triggered when the lock value is modified by another processor. This simple approach is also efficient.

5.5.4 Interrupt Mechanism

Generally an interrupt is an asynchronous event which is triggered by external devices and breaks the normal program execution flow. For a specific architecture, both the interrupt vector table and *Interrupt Service Routines* **(ISR)** are necessary to realize the interrupt mechanism.

With our simulation platform, instead of design ISR for each interrupt, we put all interrupt related functions "under" the ISS. When devices want to interrupt the core, the interrupt SystemC module will handle these requirements directly. For all processes and threads related to this interrupt, the SystemC interrupt module will directly modify the appropriate data in the target memory. Though this mechanism can simplify interrupt related operations, we should also take care of the communication interface between the interrupt module and OS threads.

5.5.5 I/O Devices

In one MPSoC platform, multiple *I/O Devices* may be included as peripherals. Traditionally, as devices and drivers are developed separately, a small modification of the hardware device may cause the failure of the whole MPSoC simulation. In our hybrid simulation platform, instead of realizing and compiling drivers into target processors, we integrate all driver functions with the device realization.

Generally, I/O operations from the target system can be realized in two ways: the non-blocking access for the interrupt mechanism and the blocking access for the repeated status

checking mechanism. Traditionally, for the interrupt mechanism, the driver sets up the device by writing to some status registers and changing the thread into the non-runnable status to wait for the interrupt from this device. While, for the status checking mechanism, the thread need to repeatedly check the device status until the device fulfills the requirement. Both of these two mechanisms can be well modeled with non-blocking functional call and blocking functional call respectively. For the non-blocking functional call of the interrupt mechanism, when the host device gets the request, it returns immediately and launches the SystemC method to execute this command in the background. After some clock cycles pass, the host module can interrupt the processor and return device results. The internal realization of this non-blocking functional call uses the interrupt mechanism. For the blocking method, the functional call returns the control right to the processor when all assigned works are finished.

5.5.6 Reentrancy and atomicity

Reentrancy is used to describe a computer program which can be safely executed concurrently. *Atomicity* refers to a set of operations that can be combined so that they appear to the rest of the system to be a single operation with only success or failure results. Most HAL realization subroutines need to have the reentrancy or atomicity properties such as operating task queues or handling interrupts. As most target realization of HAL APIs needs to take care of reentrancy and atomicity problems, the realization becomes difficult and error-prone. For the atomicity case, several locks are included which should take care of avoiding a dead-lock situation. In the multiprocessor environment, instructions like *Test-and-Set* are necessary to guarantee the atomicity property.

In our hybrid simulation platform, most HAL APIs are implemented locally. As the host simulation platform does not simultaneously execute these HAL functions, problems and bugs related to reentrancy and atomicity are avoided.

5.6 Conclusion

This chapter proposes a possible HW/SW interface modeling method for the automatic generation flow. This method uses the service idea to describe both software and hardware components in one HW/SW interface. Though the performance evaluation is still the bottleneck of this flow, this method provides the possibility to automatic generation for relatively complex HW/SW interfaces.

Besides the automatic generation method, this chapter presents a hybrid simulation platform specifically designed to solve the HdS adaptation problem in the HW/SW interface design. This hybrid platforms shows the I/O device modeling flexibility and the HAL APIs porting advantages. As both flexibility and porting advantages are important properties for system validation at early design stages, our hybrid simulation platform shows its value for the whole MPSoC DSE flow.

Chapter 6

Task Migration Framework for Heterogeneous MPSoC Architectures

Contents

Because of the configurable processors, the MPSoC architecture using several configurable processors with different extended instruction set becomes a nature choice for latest embedded systems [22]. But the performance of traditional static task mapping solutions can not provide enough flexibility and performance due to the workload inbalance. Because of the heterogeneity property, most of existing operating system cannot provide the

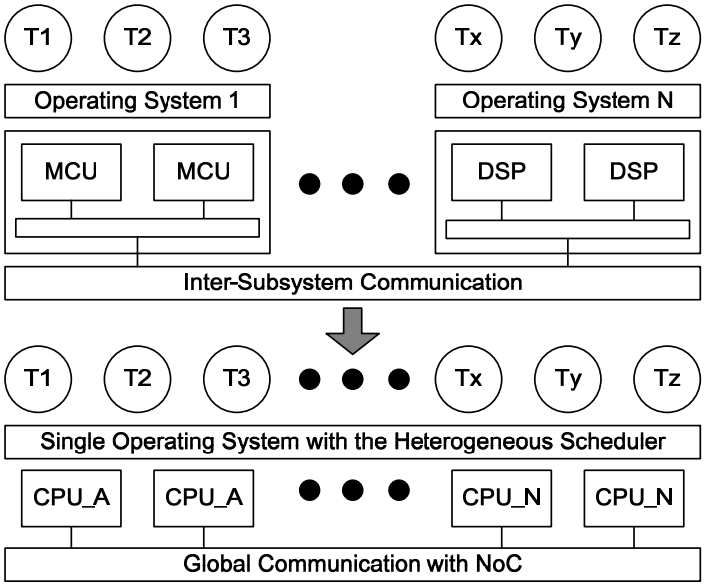

Figure 6.1: Heterogeneous MPSoC Software/Hardware Architectures.
Top: the multi-subsystem based architecture. Bottom: the architecture based on multiple different extended processors.

task migration ability between the processors configured differently. In this chapter, a formal task migration framework is presented to provide a more flexible and higher performance heterogeneous multiprocessor execution environment.

In this chapter, we would like to answer the following question:

- How to make heterogeneous MPSoC more flexible by enabling the task migration function?

6.1 Basic idea

Migrating one task between different processors is regarded as a key function of the multiprocessing OS design for *Symmetric Multi-Processors* **(SMP)** based systems. This ability can balance the workload among different cores to reduce idle time and achieve higher overall performance. Because instruction set formats and register files are totally different in general heterogeneous MPSoC platforms, it is impossible to have the task migration function for them. For example, the top part of Fig.6.1 represents a general heterogeneous MPSoC platform which includes several different kinds of MCUs, DSPs and other types of processors. As instruction sets of MCUs and DSPs are not compatible, tasks compiled and assigned to MCUs cannot be migrated to DSPs even when DSPs are idling and waiting for new tasks. This kind of subsystem based architectures is commonly used for most existing heterogeneous platforms.

As both heterogeneous properties and workload balance ability are essential for modern MPSoC platforms, we propose a framework which can provide both heterogeneous processing units and task migration ability. Because the key constraint of the task migration is

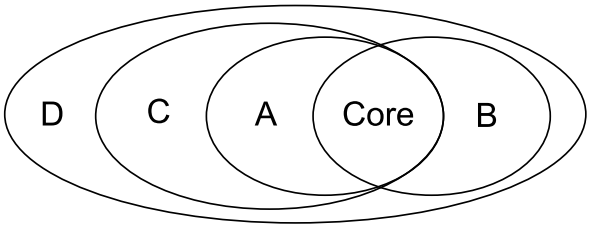

Figure 6.2: Instruction Set Relationship: an example of 4 different instruction sets which share the same core instruction set in the center.

that the system software such as OS and drivers should be able to execute on all underlying processors, we make all processors share the same core instructions and registers set for system software. Besides this, computing related instructions and registers (for the applications usage) can be totally different between each processor to provide special acceleration for different applications. With this framework, both the heterogeneous advantage and the migration advantage can be achieved by our platform.

From the realization point of view, we use heterogeneous MPSoC architecture based on configurable processors. As all extended processors share the same core instruction set which can be used for the OS realization to satisfy all operating system and communication requirements. The lower part of Fig.6.1 describes such a platform that all ASIPs share the same OS image and tasks can be migrated between different kinds of processors. We would like to give the formal definition and more details about this task migration solution.

6.2 Instruction set relationship

To explain the relationship between processor instruction sets and the task migration ability, we use the concise example of Fig.6.2. This diagram represents the instruction set relationship of one configurable heterogeneous MPSoC platform. In this platform, we have 5 different kinds of instruction sets which are **Core**, **A**, **B**, **C** and **D**. We assume that the instruction set relationship is the same as the register file relationship and therefore we have a single relationship. In following parts, we discuss the *Core Instruction Set* and *Extended Instruction Set* separately.

6.2.1 Core Instruction Set

As our work is based on configurable processors, a strong prerequisite is that all processors share the same core instruction set shown in the center of Fig.6.2. Because we need to realize our task migration framework based on this core instruction set, we define the following groups of instructions.

- **Arithmetic Logic Instructions**: which are used for arithmetic and logic operations.
- **Memory Access Instructions**: which are used for transferring data between memories and registers.

- **Program Flow Control Instructions**: which are used for changing the program execution flow based on the processor status.
- **Concurrent Access Control Instructions**: which are used to serialize requests and avoid non-coherent shared memory access cases in multi-processors execution environments.

Besides these classes of instructions, we also need the following 4 groups of core register files for operating system realization.

- **General Purpose Registers**: which are used for the storage of the data and address information.
- **Program Counter Registers**: which are used to indicate the current program address.
- **Program Status Registers**: which are used to store current processor statuses and include the exception status, the interrupt status and so on.
- **Program Stack Register**: which is used for the current stack address and can be realized using one of the general purpose registers.

Due to the generality of operating system software, it is possible and efficient to build a real operating system in which tasks can migrate among multiple heterogeneously extended processors by using only the core instruction set and the core register file. To ensure this stringent requirement, the designers should not remove critical components (such as atomic instruction, interrupt masking and handling) used by this core instruction set and register file during the processor configuration process.

6.2.2 Extended Instruction Set

As we use extended instructions to benefit from the data and instruction level parallelism of the applications, we define extended instruction sets and the set relationship between them. Most extended instructions can be divided into either SIMD or MIMD instructions, both requiring independence between the parallel computations.

Besides extended instructions, it is also often necessary to extend the register files to improve the performance.

- **Very Wide Registers**: to improve SIMD instructions performance, we add very wide register files to store the instruction operands and results.
- **Special Internal Registers**: which are used for some special operations such as accumulator registers for the multiply-accumulate operation.

Normally, these extended instructions can be chosen by the application programmer based on the benchmark profiling results and the existing architecture and area/power constraints. Automatically extracting the candidate extended instruction set from one specific

application is also the focus of researches by both academia and industry for a long time, see for example the book by Leupers[73]. Among others, recent proposals are L. Pozzi et al. [90], F. Sun et al. [95] and Xtensa Processor Extension Synthesis (XPRES) Compiler [3] are such works. The formalism we propose does not make any assumption on how the extended instructions are extracted, and thus the resulting ISA can be classified accordingly.

As both read and write of user extended registers are crucial for the context switch and related detailed migration operations, we should also make sure that the extended instruction sets include specific load and store instructions to access all these extended registers. In the context related functions, the extended register files used or required by the task should be load or stored properly by using this kind of extended instructions. For example, we have the ARM processor with the NEON coprocessor extension [7] for media related applications. The NEON extension integrates thirty-two 64 bit double word registers which can only be load and stored by using NEON extended instructions such as `VLDn` and `VSTn`.

6.2.3 Formal Definition

In one task migration enabled heterogeneous P processors MPSoC platform, we have $N \leqslant P$ different instruction sets that all include the core instruction set I_{core} and the core register file R_{core}. We call $S_{core} = I_{core} \cup R_{core}$ the union of the core instruction set and core register set. Meanwhile, we note the extended instruction set as EI_i and the extended register file as ER_i.

Definition 1: Sets Relationship.

- For each processor in one heterogeneous MPSoC platform, we have the instruction set I_i and register file R_i definition:

$$\forall 1 \leqslant i \leqslant N : I_i = EI_i \cup I_{core}$$

$$\forall 1 \leqslant i \leqslant N : R_i = ER_i \cup R_{core}$$

- For each processor in one heterogeneous MPSoC platform, we have the extended instruction and register set ES_i definition:

$$\forall 1 \leqslant i \leqslant N : ES_i = EI_i \cup ER_i$$

- For each processor in one heterogeneous MPSoC platform, we have the instruction and register set S_i definition:

$$\forall 1 \leqslant i \leqslant N : S_i = I_i \cup R_i = S_{core} \cup ES_i$$

- For all instruction sets in one heterogeneous MPSoC platform, we have the instruction set group $\mathbb{S}$ definition:

$$\mathbb{S} = \{S_1, S_2, ..., S_N\} = \{S_i : 1 \leqslant i \leqslant N\}$$

With these definitions, we can easily express the relationship of Fig.6.2. We note the instruction set group $\mathbb{S} = \{S_{core}, S_A, S_B, S_C, S_D\}$. Because the core instruction set is one part of each processor, we have the requirement that for P processors in one heterogeneous MPSoC platform: $\forall 1 \leqslant i \leqslant N, S_{core} \subseteq S_i$. Thus, for the special case of Fig.6.2, if all of ES_i are not empty, we have following expressions: $S_A \subset S_C$, $S_C \subset S_D$ and $S_B \subset S_D$.

6.3 Task compilation and migration

As extended instructions are only efficient for some specific applications, or even more precisely for some kernels of an application, we compile some application tasks and threads with the basic instruction set while some others with extended ones. The basic threads can be migrated for execution on any available processors while extended application tasks and threads can be only be executed on processors which should realize this instruction set extension. In the example of Fig.6.2, as the instruction set S_A is a subset of S_C and S_D, it means that S_A has less extended instructions than S_C and S_D. On one side, if some tasks compiled to S_C instruction set and use instructions which not belongs to S_A, they are not able to be migrated to processors only support the S_A type instruction set. On the other side, if some tasks just use the S_A instruction set, it is no problem for them to run on S_C and S_D types processors. This execution relationship is presented in Fig.6.3.

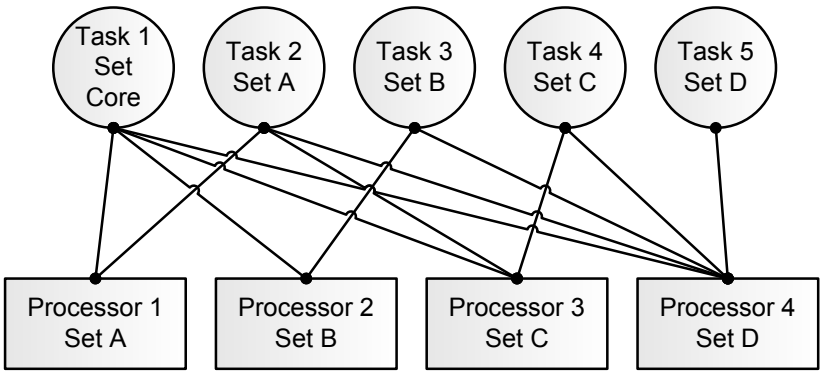

Figure 6.3: Processors and tasks compatibility.
All tasks T_i and processors P_j realize one of S_k from the total instruction set $\mathbb{S}$. The execution relationship is represented with connection between tasks and processors.

The tasks, heterogeneously extended processors and execution relationship can be represented as a the **Compatibility Graph** $\mathcal{G}$ shown in Fig6.3.

Definition 2: Migration possibility of the heterogeneous MPSoC $\mathbb{M}$ is defined as $\mathbb{M} = (\mathbb{S}, \mathbb{T}, \mathbb{P}, \mathbb{G})$.

- $\mathbb{S}$ is the set that includes all instruction sets used in one heterogeneous MPSoC platform (the same as Definition 1).
- $\mathbb{T}$ represents the task set which include N_T tasks for one application system and $N_T \geqslant 1$. We have $\mathbb{T} = \{T_1, T_2, ...T_{NT}\}$. When task T_i is compiled onto one spe-

cific instruction set S_j, we can represent the *is compiled for ISA* relationship with the symbol $\star$. For this case, we have $T_i \star S_j$.

- $\mathbb{P}$ is the set of processors which includes N_P different processors for one MPSoC platform and $N_P \geqslant 1$. When processor P_i realizes one specific instruction set S_j, we can present the *realizes ISA* relationship with the symbol $\triangle$. For this case, we have $P_i \triangle S_j$.

- A **Bipartite Compatibility Graph** $\mathcal{G} = (\mathbb{T}, \mathbb{P}, \mathbb{C})$ represents the compatibility relationship between each $T_i \in \mathbb{T}$ and $P_j \in \mathbb{P}$. An edge $\{T_i, P_j\} = c_k \in \mathbb{C} \subseteq \mathbb{T} \times \mathbb{P}$. This edge c_k means task T_i can be executed by processor P_j.

$$\forall i, j, T_i \in \mathbb{T}, P_j \in \mathbb{P}$$
$$(T_i \star S_k) \wedge (P_j \triangle S_l) : S_k \subseteq S_l \Longleftrightarrow c(T_i, P_j)$$

With both Def.1 and Def.2, we clarify the compatibility relationship among instruction sets, tasks and processors. In Fig.6.3, we have an example of compatibility with 4 processors and 5 tasks. So we have the task set $\mathbb{T} = \{T_1, T_2, T_3, T_4, T_5\}$ and the relationship between tasks and instruction sets $\{T_1 \star S_{core}, T_2 \star S_A, T_3 \star S_B, T_4 \star S_C, T_5 \star S_D\}$. Meanwhile, we have the processor set $\mathbb{P} = \{P_1, P_2, P_3, P_4\}$ and the relationship between processors and instruction sets $\{P_1 \triangle S_A, P_2 \triangle S_B, P_3 \triangle S_C, P_4 \triangle S_D\}$. For the task compatibility of this example, we have the compatibility relationship: $\mathbb{C} = \{c(T_1, P_1), c(T_1, P_2), c(T_1, P_3), c(T_1, P_4), c(T_2, P_1), c(T_2, P_2), c(T_2, P_3), c(T_2, P_4), c(T_3, P_2), c(T_3, P_3), c(T_3, P_4), c(T_4, P_3), c(T_4, P_4), c(T_5, P_4)\}$. All these instruction sets, tasks, processors and compatibility relationships are represented by the compatibility graph $\mathcal{G}$. From this example, we can clearly find tasks compiled with the core instruction set have the best flexibility while the tasks compiled with extended instruction sets can only be executed by specific processors.

6.4 Heterogeneous task scheduling algorithms and Realization

As the compatibility of tasks and processors is readily visible with our task migration framework, we now need to provide a way to choose the right process and thread to migrate or elect. This is the goal of our heterogeneous task scheduling algorithms. Based on the instruction set compatibility rules, we adapt several existing task scheduling algorithms to our heterogeneous MPSoC task migration framework. Because different configured processors can execute different classes of tasks, our task scheduling algorithms try to utilize the extended instruction set advantage and trade-offs between the scheduler efficiency and the execution efficiency. The formal descriptions and realization details of these scheduling algorithms are also given in this section.

With both Def.1 and Def.2, for a specific processor P_i, compatible tasks can be grouped into the set $\mathbb{T}_i$ where $\mathbb{T}_i = \{T_{i1}, T_{i2}, ..., T_{ij}\}$. Based on the *compatibility graph* $\mathcal{G}$, there

should be an edge between each P_i and T_{ij} to guarantee the compatibility. In the following algorithm descriptions, both P_i and $\mathbb{T}_i$ are used to present this compatibility property.

Beside the compatibility property, we also need a value to evaluate the efficiency of processor computation. For a task T_i running on a processor P_j, we have the difference $D(T_i, P_j)$ defined as $\exists S_i, S_j \in \mathbb{S}, (T_i \star S_i) \wedge (P_j \triangle S_j) : D(T_i, P_j) = |S_j - S_i|$. This definition represents the distance between the instructions and registers that the processor P_j provides and the T_i task requires. The bigger number of $D(T_i, P_j)$ means the more unused instructions provided by processor P_j which wastes computation ability and power. Two of the following scheduling algorithms are designed to take account of this efficiency problem. During the scheduling process, $\mathbb{T}_{queue}$ is used to defined all tasks inside the runnable queue structure. Meanwhile, $|\mathbb{T}_{queue}|$ is defined as the size of this task queue.

6.4.1 FMFS algorithm

First Match First Serve **(FMFS)** is one of the simplest algorithms for our heterogeneous MPSoC platform. The basic idea is just add the compatible constraints into the traditional FIFO like scheduling algorithms. When a processor is ready for new tasks execution, it goes through the task queue and picks up the first compatible task to execute. Though this algorithm is simple and efficient for the scheduler realization, it does not fully optimize for extended instruction sets of heterogeneous processors. With this FMFS algorithm, tasks with smaller instruction sets generally have better execution chances which decrease the whole system performance.

Algorithm:
Search the queue in order to select the first task that is compatible.
Performance:
Complexity of this algorithm is $O(1)$.

For implementation, we have following abstract code.

Input:
A compatibility graph $\mathcal{G} = (\mathbb{T}, \mathbb{P}, \mathbb{C})$.
A free processor P_i and a task queue $\mathbb{T}_{queue}$ in FIFO order.
Output:
If exist, select a compatible task for the processor P_i.
Implementation:

```
for j = 1 to |T_queue| in FIFO order
    if c(T_j, P_i) ∈ C // T_j is compatible with P_i
        T_j is the result and finish this scheduling process
    end if
end for
Default idle task is the result // There are no compatible tasks
```

6.4.2 Most compatible algorithm

To fully take the advantage of powerful extended instruction sets, we define the most compatible algorithm. By using this algorithm, when a processor is ready for new tasks execution,

it iterates over the whole task queue and compares the CPU instruction set with each waiting task. After the whole task queue is checked, the compatible task which uses the most instructions is chosen for execution. As this algorithm emphasizes tasks using extended instruction sets, in some cases, it may provide better overall performance. But we should also notice that tasks compiled only with the core instruction set may be in a starving situation if tasks making use of extension are always ready to run.

Algorithm:
Search the queue and execute the most compatible task.
Performance:
Complexity of this algorithm is $O(|\mathbb{T}_{queue}|)$.

For implementation, we have following abstract code.

Input:
A compatibility graph $\mathcal{G} = (\mathbb{T}, \mathbb{P}, \mathbb{C})$.
A free processor P_i and a task queue $\mathbb{T}_{queue}$.
Output:
If exist, select a compatible task for the processor P_i.
Implementation:
A empty candidate task set $\mathbb{T}_{candidate} = \phi$.
forall $T_j \in \mathbb{T}_{queue}$
if $c(T_j, P_i) \in \mathbb{C}$ // T_j is compatible with P_i
$\mathbb{T}_{candidate} = \mathbb{T}_{candidate} \cup T_j$ // Add T_j to the candidate set
end if
end forall
if $\mathbb{T}_{candidate} \neq \phi$
choose the first (or any) task from the set T:
$T = \min(D(T_j \in \mathbb{T}_{candidate}, P_i))$.
else Default idle task is the result // There are no compatible tasks
end if

6.4.3 Priority based most compatible algorithm

To avoid the drawbacks of both the FMFS algorithm and the most compatible algorithm, we combine these two algorithms together and create the priority based most compatible algorithm. In this algorithm, we add a priority level to each task. Instead of having a single task queue, we have several task queues, each corresponding to a priority. This allows to limit the search time and avoids the starving situation, by increasing the priority of long waiting threads. Meanwhile, for each priority level, we still use the most compatible algorithm to find the best candidate for one processor. The priority level for each task can be adjusted depending on some priority calculation algorithms to avoid starving situation and decrease the overall system response time. The drawback of this algorithm is the complex realization and the scheduler performance heavily depends on priority setting and adjustment algorithms.

Algorithm:
Search from the highest priority queue and execute the best compatible waiting task.
Performance:
Complexity of this algorithm is $O(|\mathbb{T}_{queue}|)$.

For implementation, we have following abstract code.

Input:
A compatibility graph $\mathcal{G} = (\mathbb{T}, \mathbb{P}, \mathbb{C})$.
A free processor P_i.
Multiple task queues $\{\mathbb{T}_{queue\_1}, \mathbb{T}_{queue\_2}, ..., \mathbb{T}_{queue\_n}\}$ for n different priorities.
Output:
If exist, select a compatible task for the processor P_i.
Implementation:
A empty candidate task set $\mathbb{T}_{candidate} = \phi$.
for k = 1 to n // n queues with different priorities
forall $T_j \in \mathbb{T}_{queue\_k}$
if $c(T_j, P_i) \in \mathbb{C}$ // T_j is compatible with P_i
$\mathbb{T}_{candidate} = \mathbb{T}_{candidate} \cup T_j$ // Add T_j to the candidate set
end if
end forall
if $\mathbb{T}_{candidate} \neq \phi$
choose the task $T = \min(D(T_j \in \mathbb{T}_{candidate}, P_i))$ and the result is T.
end if
end for
Default idle task is the result // There are no compatible tasks

6.5 Scheduler realization

To realize all discussed task migration algorithms on one heterogeneous MPSoC platform, we should well identify instruction sets provided by processors and used by tasks. Besides this, we also have some task migration realization details which are different from traditional SMP based schedulers.

6.5.1 Instruction Set Identification

As the instruction set representation is important for both processors and tasks, the scheduler of the operating system should have a special mechanism to store this information. We have `ISA_ID` to represent the instruction set information. Then we assign one `CPU_ISA_ID` for each processor and one `TASK_ISA_ID` for each task. The use of specific ID to indicate processor instruction set differences is a method commonly used in industry. As the instruction set relationship is complex in our platform, we would like to have the `ISA_ID` to well represent the instruction set relationship. By using this ID, it is convenient for the scheduler to handle the relationship in run-time environments.

In our framework, we have instruction sets $\mathbb{S} = \{S_1, S_2, ..., S_{NS}\}$ and the relationship $\mathbb{R} = \{S_1 \supseteq S_2, ..., S_{NS-1} \supseteq S_{NS}\}$. We map the instruction set to the binary number set the $\mathbb{N} = \{1, 2, ...\}$ and the relationship to *Bit OR* relationship.

In Fig.6.2, we have 4 different extended instruction sets and the core instruction set. In Table.6.1, we assigned each separated instruction group with a *bit* and each `ISA_ID` with a binary number with bit validation for each small instruction group. By using `ISA_ID`,

Table 6.1: Processors and Supported Tasks ISA Identification Example.

Set	CPU_ISA_ID	Compatible Task	Compatible Task_ISA_ID
Core	0x0000	Core	0x0000
A	0x0001	Core and A	0x0000, 0x0001
B	0x0010	Core and B	0x0000, 0x0010
C	0x0101	Core, A and C	0x0000, 0x0001, 0x0101
D	0x1111	Core, A, B, C and D	0x0000, 0x0001, 0x0010, 0x0101, 0x1111

we replace the complex instruction set relationship with simple bitwise operations. For each processor, the `CPU_ISA_ID` is the same as the `ISA_ID` of the one realized. Meanwhile, for each task, the `Task_ISA_ID` is the instruction set `ISA_ID` used by the compiled binary. The compatibility relationship between `CPU_ISA_ID` and `TASK_ISA_ID` is also illustrated in Table.6.1.

6.5.2 Instruction Set Based Scheduler Realization

With the definition of both `CPU_ISA_ID` and `TASK_ISA_ID`, we use the bitwise *or* operation to handle the instruction set compatibility test. To test compatibility, we need only this operation:

```
(CPU_ISA_ID | Task.TASK_ISA_ID) == CPU_ISA_ID
```

This test only relies on simple bit and compare operations to make the computation efficient for the frequent usage $c(T_j, P_i) \in \mathbb{C}$ in all heterogeneous task scheduling algorithms.

6.5.3 CPU_ISA_ID and TASK_ISA_ID Integration

In our task migration framework, each processor should have a `CPU_ISA_ID` which presents the instruction set and register file it realizes. In realization, we add one specific read-only register to each processor which is hard coded to indicate the corresponding `CPU_ISA_ID`. The task migration framework should access this register during all task operations.

The `TASK_ISA_ID` is assigned to each task during its creation. We have modified the POSIX Thread standard `pthread_attr_t` structure by adding the `TASK_ISA_ID` tag. When a task is created with standard `pthread_create`, this ID is transfered to the OS kernel and used for the heterogeneous task scheduling described before and the context related functions realization in the following discussion.

6.5.4 Context Related Functions Realization

The context related functions for heterogeneous MPSoC platforms are more complex than for homogeneous ones. There are different extended register files $R_i = ER_i \cup R_{core}$ in each processor i, so we need to handle these extra registers ER_i in our context related functions. In contract to loading and storing all extended registers during these operations, we should

only touch the one used by the previous executed task and required by the next executed task. For example, the new context switch function should include following four steps:

- Store all core registers to the stack of the previous executed task.
- Store the necessary extended registers depending on the `TASK_ISA_ID` of the previous executed task.
- Load all core registers from the stack of the next executed task.
- Load all necessary extended register based on the `TASK_ISA_ID` of the next executed task.

As a processor may run a task that makes use of only a subset of the extended register, we save the registers depending on the instruction set used by this task. This save action is feasible because the registers used by the task is a subset of the processor registers and the load action is also feasible for the same reason. The context structure has a shared part that contains the R_{core} registers, and a private part that depends on the extended registers used by the task ER_i.

Though the realization of the context related functions make the structure of each task context different depending on the `TASK_ISA_ID`, it can avoid much unnecessary stack memory occupation and save the time of register operations. We still take the ARM processor example with the NEON coprocessor extension. As the NEON extension integrates thirty-two 64 bit double word registers, it should consume minimal 256 bytes context memory space. Besides the memory cost, loading and storing all these extra registers wastes some time and power which are important for embedded systems. With our task migration framework, we can fix the context size of the task dependent on its `TASK_ISA_ID` when created. If the task does not use the NEON extended instructions and registers, the extra memory and operation time are saved with our task migration framework.

6.6 Conclusion

We describe a task migration framework in this chapter which focuses on heterogeneous architectures which are commonly used for configurable processors based MPSoC design. Though this framework needs a shared core instruction set, it makes possible for tasks be migrated among processors with different extended instruction sets. Beside the compatibility analysis of this task migration framework, We formally define it with 3 proposed scheduling algorithms. Both heterogeneous advantages and the SMP flexibility advantage can be achieved with this task migration framework.

Chapter 7

Experimental Results

Contents

To well verify all modeling techniques and simulation platforms proposed in the previous chapters, we use the *Motion JPEG* decoder case study in this experimental results chapter. We will show the advantages of using multiple abstraction levels, the exploration flow based on these abstraction levels and HW/SW interface for heterogeneous MPSoC architectures based on configurable processors.

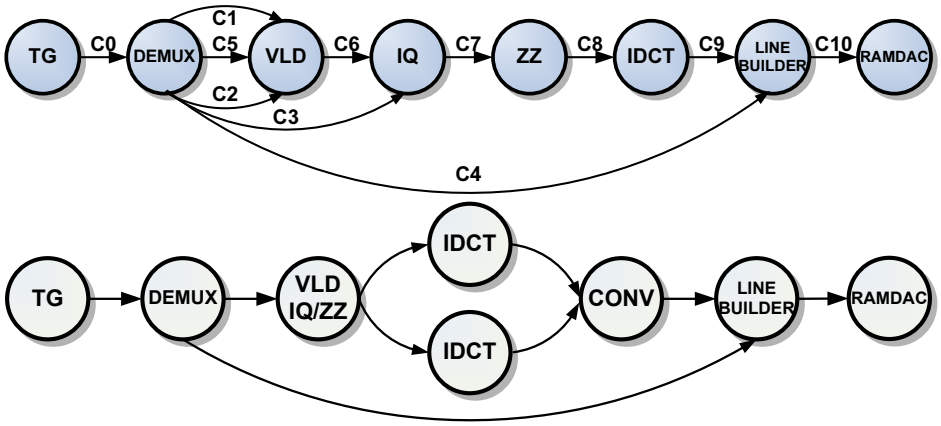

Figure 7.1: Two Functional Models of the Motion-JPEG Case Study

7.1 Motion JPEG Decoder Case Study Introduction

This section presents the software and hardware architecture of the Motion JPEG decoder case study. The following experiments are all based on this architecture, the SystemC simulation platform and the Ubuntu Linux development platform.

7.1.1 Application definition

The Motion-JPEG is a multimedia format in which a video sequence is separately compressed as JPEG images. This format is often used in mobile applications such as digital cameras. In this case study, we have two different realization models for the Motion-JPEG decoder application. They work by reading a stream of JPEG images from the Traffic Generator (TG) task and writing the decoded pixels into the Random Access Memory Digital-to-Analog Convert (RAMDAC). The traditional one has eight initial tasks: TG, DEMUX, VLD, IQ, ZZ, DCT, CONV, LINE BUILDER and RAMDAC in the upside of Fig.7.1. The optimized one shown in the downside of Fig.7.1 merges VLD, IQ and ZZ together to save the communication cost. Because IDCT is the most time consuming task in the whole application, we separate it into 2 or more instances to take the advantage of multiple processors. Both of these two application models are used in the following MJPEG case study.

7.1.2 Operating system definition

Mutek [84] is an open source lightweight operating system mostly for the academic usage. In Fig.7.2, we present the basic architecture of Mutek OS which provides the POSIX Thread API [23] for high level applications. As the POSIX Thread API is one of the OS interface most commonly used for the multi-thread application development, our experiment platform is general and flexible for all application domains. This Mutek OS supports an SMP shared memory multiprocessor architecture and is already ported to several processors which include ARM[6], MIPS[76], SPARC[65], Xtensa[3] and so on.

Because Mutek targets the academic usage, it only supports a few I/O devices and does

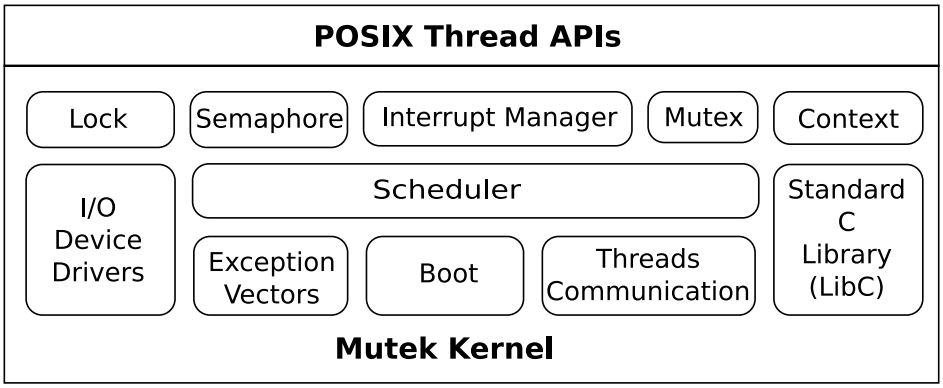

Figure 7.2: Mutek Operating System Architecture [94]

not include the virtual memory support. Also, the scheduling algorithms used in this OS are relatively basic and do not have much optimization. Compared to uClinux and other Linux based open source OS, Mutek is very small and easy to configure. That is the most important reason why we choose Mutek OS in this thesis.

7.1.3 Configurable processors and extended instruction sets

Beside the traditional SMP architecture with multiple symmetric embedded processors, we also use a heterogeneous MPSoC architecture in this thesis. With this architecture, all configured processors use the basic instructions of the Xtensa LX2 processor. The extended instruction set compiler and the software compiler are also provided by Tensilica [3].

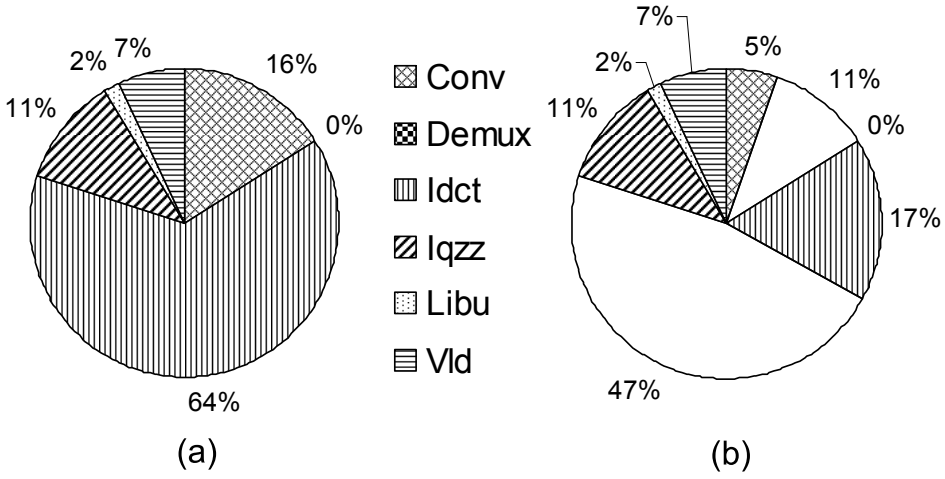

Figure 7.3: Computation Time Difference
With the core instruction set (left) and extended instruction set (right) separately. The acceleration effect of extended instructions is obvious.

After the task profiling of the Motion-JPEG example, we get the left part of Fig.7.3 which indicates the computation time used for each tasks based on the core instruction set. As we focus on the application optimization, this figure only shows 6 software tasks, the operating system and communication cost is excluded to make the comparison clear. In this figure, we

Table 7.1: Both Instruction Set and Register File Extension for IDCT and CONV Tasks with Speed and Cost Information

	IDCT Core	CONV Core
New Instructions	10	14
New Registers	4	28
Extra Gates	70,904	63,869
Total Gates	139,904	132,869
Speedup Effect	380%	309%

find that the IDCT and the YUV to RGB Converter (CONV) are two most time consuming tasks which consume 64% and 16% of CPU time separately. The following instruction set extension work focuses on these two tasks.

Table.7.1 shows user defined extended instruction sets for both IDCT and CONV tasks (these extended instructions are given by designers based on the usage frequency and performance improvement effects). With extended instructions, IDCT and CONV tasks speed up more than 3 times and new computation time of these 6 tasks is shown at the right of Fig.7.3. It is obvious to identify the efficiency of extended instructions for real applications. With this table, we also give the hardware cost of these extended instructions and registers. This information is useful to show the advantage of our migration framework for the Cost/Performance ratio.

7.1.4 Simulation platform and instruction set simulator

In this experimental result section, we have two different simulation systems which are the SoCLib based platform and the Xtensa SystemC (XTSC) based platform.

SoCLib

The *SoCLib* [15] is an open source platform for virtual prototyping of multi-processors system on chip (MPSoC). All this platform is developed based on SystemC [13] with both instruction set simulators and peripheral IP components. All processors and peripherals in the SoCLib platform respect the VCI/OCP communication protocol. In the following experiments, we mainly use the SPARC and MIPS processors from SoCLib. Besides the processors, we have also the peripherals such as tty, memories and locks.

Xtensa SystemC (XTSC)

The *Xtensa SystemC* **(XTSC)** package supports transaction-level modeling of Xtensa cores in a SoC SystemC simulation environment. The XTSC package includes the following items:

- **Xtensa ISS**: It supports two different simulation modes which are the normal cycle accurate simulation mode and the high-speed cycle approximate simulation mode (Turbo Xim mode). The Turbo Xim mode can achieve very high simulation speed with the

assumption of each instruction can be executed in one cycle and no cache effect are modeled.

- **Intermodule Communication**: The intermodule communication component supports the TLM interface and can well connect with other processors and peripherals.
- **Peripherals**: The peripherals library includes a set of configurable SoC example components such as memories, arbiters and routers in the form of SystemC modules.

7.2 Hybrid Simulation Platform with the HAL APIs

In this section, we demonstrate the accuracy of hybrid simulation technology and the simplicity of I/O device modeling with DMA transfer example by using our hybrid simulation platform.

Table 7.2: System Performance and Simulation Speed.

	Hybrid Platform	Traditional ISS Platform	Speedup
Extended Instructions Acceleration Ratio (times)	1.71	1.53	10%
Host Simulator Speed (M Cycles/s)	4.07	0.18	2123%

7.2.1 Hybrid simulation speed and accuracy

We compare both the simulator speed and CMPSoC architecture speedup ratio with extended instructions in Table.7.2. Because speedup effects ($speedup = \frac{Speed\_of\_extended\_instruction\_set}{Speed\_of\_core\_instruction\_set}$) of both traditional and hybrid simulation platform have only 10% difference, we believe that both platforms can well present the acceleration property of extended instructions. Because of the dynamic binary translation technology, the simulation speed of our hybrid simulation platform (presented in Chapter 4.2.2) is almost 21 times faster than traditional ISS based simulation platform. From these results, we are confident that our hybrid simulation platform is interesting for design space exploration. As the absolute performance given by our hybrid simulation platform does not take account of cache effects and communication impacts, to show very accurate performance results is not the target of this hybrid platform. This example also shows that the hybrid simulation platform can well support general multi-thread applications with HAL APIs implemented by SystemC modules on host machines directly.

7.2.2 DMA Transfer Example

DMA is a feature of microprocessors system that allows certain hardware subsystem to access system memory for reading and writing independently of the processors. As DMA

devices are commonly used in most MCPSoC platform, we use it as our example to show how to model peripherals and device drivers in this hybrid simulation platform.

```
int ta_dma::dma_ss_copy(
xtsc_core &core, int from, int to, int size){
  // Copy Original Data to DMA Buffer.
  core.peek_physical(from, size, buff);
  // Consume time for this copy operation
  consume(size/4*time_per_word);
  // Copy DMA Buffer to Target Memory.
  core.poke_physical(to, size, buff);
  return size;
}
```

The `dma_ss_copy` function is the hardware model for the memory to memory data copy function of the DMA device. In this function, we use specific `peek_physical` and `poke_physical` functions introduced in section 4 for target memory operation. Different from traditional complex register read/write operations, this SystemC realization uses only 4 lines of C++ code to realize a DMA copy function and also make a rough estimate of the time used by this operation. Beside the simple realization, this `dma_ss_copy` function can be called directly from the I/O device driver by using semi-hosting functional call. Traditional error-prone register operations are replaced by concise functional calls to facilitate the simulation platform building process. This example shows that the hybrid simulation platform can also simplify the peripherals and device drivers design.

7.3 Design Space Exploration Flow

This section presents how our multiple abstraction levels simulation models can help speed up the design space exploration flow and save the design resources.

7.3.1 Communication optimization at System Level

Because the bus is the most popular and well used interconnection in SoC designs, in this MJPEG case study, we choose bus protocols as our inter-subsystem communication backbone. One of the most interesting properties of bus communication models is the burst mode. Most of the modern bus protocols such as ARM AHB [24] and PI-Bus [25] support the burst mode. Though they may have different burst properties, the transfer size is always the key point. The burst mode is effective only when the transfer size is larger than a specific number.

Fig.7.4 shows the transfer size property of the MJPEG application before optimizations. From this diagram, we find that there are lots of 1 byte transfers. Because of the bus granting latency, these small transfers are not efficient. The optimization method is to merge these small data transfers together into bigger ones and transfers them by using the burst mode provided by bus protocols.

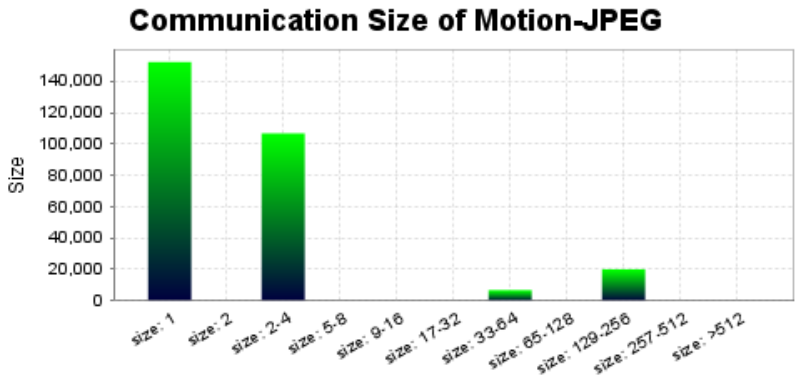

Figure 7.4: Communication Property before Optimization

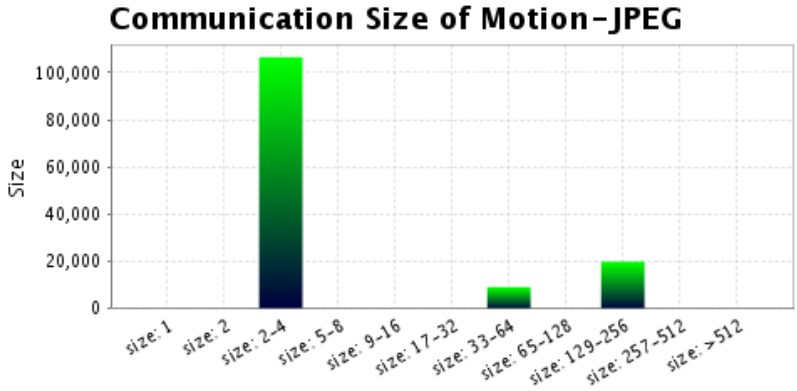

Figure 7.5: Communication Property after Optimization

The transfer size property after optimization is shown in Fig.7.5. We can clearly find that all single byte data transfers are eliminated. To evaluate the effectiveness of this optimization, we take the *PI-Bus* models as the reference protocol. Fig.7.6 shows the percentage of burst mode used under the PI-Bus protocol. From this diagram, we notice the burst ratio is significantly improved. The effectiveness of the system level communication exploration is obvious.

7.3.2 Communication optimization at Virtual Architecture Level

From the VA level communication exploration flow given at Fig.5.1, task mapping is extremely important for the whole system performance. We design a VA architecture which

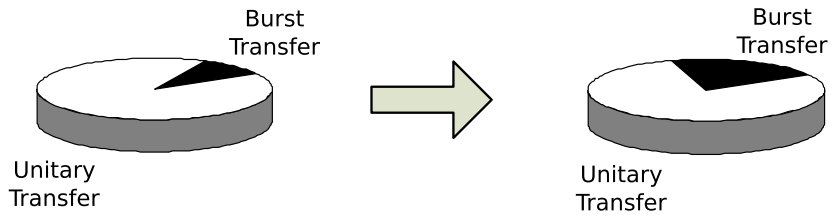

Figure 7.6: Burst Mode Percentage Change with the System Level Optimization

Table 7.3: Communication Size with Different Mapping Solutions (25 Frames MJPEG)

CPUSS1 Tasks	CPUSS2 Tasks	Comm Size
DEMUX,VLD,IQ	ZZ,IDCT,LIBU	2.2MB
DEMUX,LIBU	VLD,IQ,ZZ,IDCT	1.0MB
DEMUX,VLD,ZZ	IQ,IDCT,LIBU	5.6MB

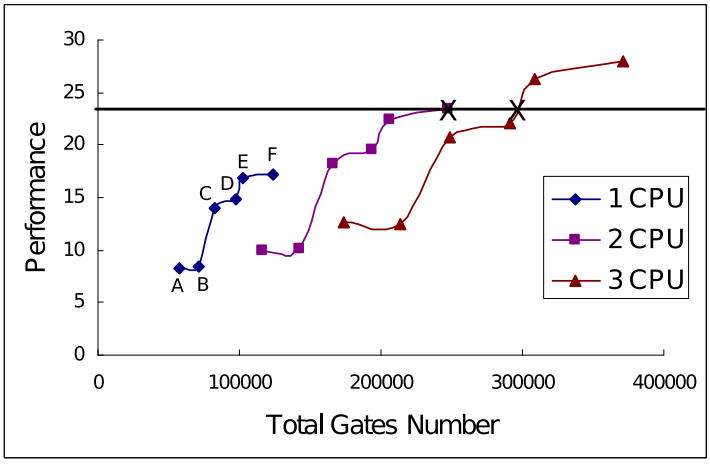

Figure 7.7: Performance VS. Gate Size
A, B, C, D, E, F are 6 performance/Gate Size trade-off points which have different extended instruction sets. All these 6 points are based on the same core number (1) and the core instruction set.

includes two CPU subsystems and two HW subsystems. The two HW subsystems are used by TG and RAMDAC tasks while two CPU subsystems are free for mapping 6 software tasks. In Table.7.3, we show three different mapping solutions and the final inter-subsystem communication size. After 25 frames of Motion-JPEG decoder simulation, the 3^{rd} solution needs 5.6 MB communication size which is more than five times compared with that of the 2^{nd} solution. From this table, we can easily understand advantages of task mapping at the beginning of the whole design process.

7.3.3 Subsystem Exploration at Transaction Accurate Level

Extension VS. Multi-Processors

To speed up the execution of one specific application, extended instruction set and multiple processors are complementary approaches. We would like to show how we explore both methods in our design flow with the Motion-JPEG example. As IDCT is the most computation intensive and time consuming task in our example, we put both IDCT and CONV tasks into one subsystem while DEMUX, VLD/IQ/ZZ and LINE BUILDER into another subsystem. TG and RAMDAC tasks are separately realized as hardware modules. In this example, we try to find which solution is the best choice for IDCT acceleration.

In Fig.7.7, we extend the instruction set of a single processor, double processors and triple processors solutions to find which one provides the lowest space cost to satisfy the

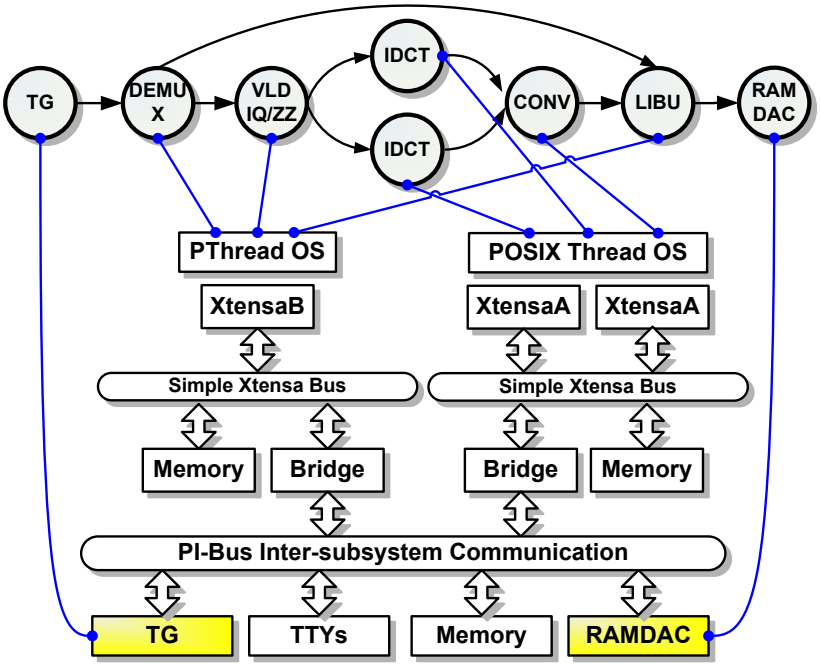

Figure 7.8: Final MPSoC Architecture After Refinement

real-time decoding requirement. The performance number in this figure indicates how many images of QQVGA (160X120) video stream can be decoded during one second, while the space cost number is defined by the total gates number used by processors in this solution. As we find in Fig.7.7, when the performance requirement is low, just extending the instruction set of single processor is a good choice. With more stringent requirements, multi-processor solutions can provide enough performance while the single processor one cannot continue to improve. In this example, to meet the 24 frames per second real-time requirement, we have two extended processor XtensaA as the best solution. To achieve higher exploration speed, all the performance data given in this example is measured at TA level with the annotation data from configured Xtensa processors.

Parameters Refinement

Each parameter at TA level is a statistic data which should be supported by several detailed parameters at CA level. Now we would like to show several examples to confirm the interest of this method.

$$T_{all} = T_{ICache} + T_{DCache} + T_{Execution} \tag{7.1}$$

Execution time for one basic block T_{all} is given by equation 7.1. The T_{ICache} and T_{DCache} is the time spent on caches and determined by cache miss rates, memory access counts and average memory access time. The $T_{Execution}$ is the instruction execution time and depends on the instruction extension set and *Cycles Per Instruction* **(CPI)**. For a specific application, the total time budget at TA level can be decomposed into detailed ones. In the Motion-JPEG example, the cache miss rate can be refined into `Cache Size`, `Line Size` and `Associativity`. If we want the instruction cache miss rate to be lower than 2%, exper-

imental results show that instruction cache should have 2K `Cache Size`, 16 bytes `Line Size` and 2 ways `Associativity`. After all the design space exploration process, the final MPSoC architecture is shown in Fig.7.8.

7.3.4 Simulation Speed at Different Abstraction Levels

The speed is the key advantage of multiple abstraction levels simulation models. We build the simulation model at system level, VA level, TA level and CA level. Table.7.4 gives out the simulation speed at these four different levels with Pentium M processor at 1.86GHz.

Table 7.4: Simulation Time for one frame at Different Abstraction Levels

	Sim Time(s)	Speed Improvement
System Level	0.096	4848.05x
Virtual Architecture Level	0.800	582.21x
Transaction Accurate Level	2.607	178.82x
Cycle Approximate Level	466.158	1

From this table, we clearly find that the higher abstraction level requires less simulation time. Compared with the most detailed CA level, the system level simulation model can finish one frames MJPEG decoding with 3 orders of magnitude faster. The simulation speed at the VA level is also much higher than that of the CA level, so this level can provide the possibility to test many different mapping solutions to meet all those constraints. Because the TA level provides more information than the VA level, the simulation speed is not as high as that of the VA level. From Table.7.4, the simulation speed at the TA level is also much higher than that of the CA level. For sure, abstraction has a cost of accuracy. However, we think that the proposed approach allows to rank solution in a correct way. When systems become complex and subsystem number increases, the speedup effect is a must for design space exploration.

7.4 Hardware/Software Interface for Configurable Processors

7.4.1 SDG case study

To verify the efficiency of the service based method, we choose the motion-JPEG video decoder as our example. In our platform, we provide three kinds of FIFOs for the task communication: pure software FIFOs, SW/HW FIFOs and pure hardware FIFOs. We show the architecture in Fig.7.9. The light color (yellow) blocks belong to the HW/SW interface and the 8 function blocks work based on them.

All the light color (yellow) blocks shown in Fig.7.9 are modeled in this SDG. Because there are many functions and APIs used in the HW/SW interface, we cannot show all the detailed service in one diagram. The services in Fig.5.2 are grouped into several large groups

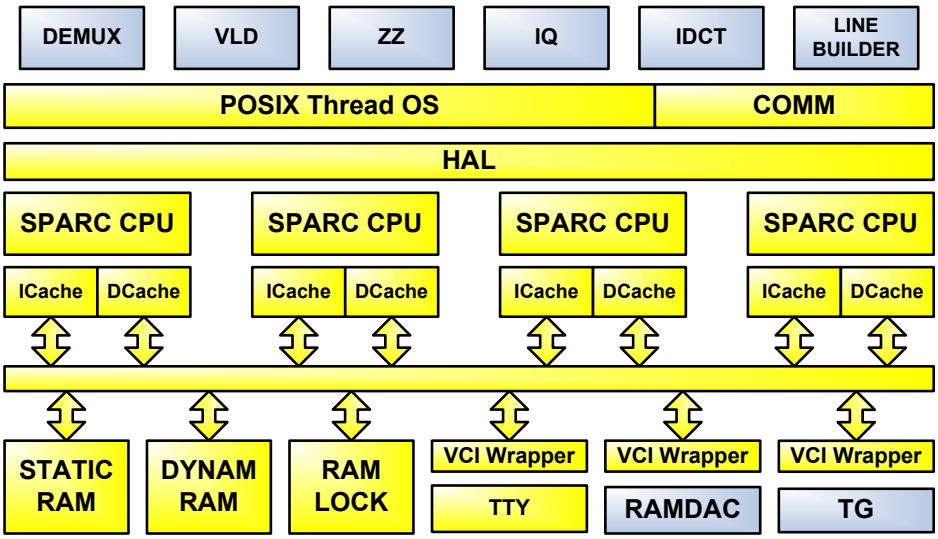

Figure 7.9: HW/SW Interface for SDG Case Study.

Table 7.5: Number of Services Used in Case Study

Service Group	Total number of services	Number of service used	Usage percentage
PThreads OS+HAL	174	110	63%
COMM	44	15	34%
SMP CPUs	2	2	100%
Interconn +Mem	3	3	100%
Total	223	130	58%

and each group includes some detailed services. For example, the POSIX thread operating system group includes 98 different detailed services (shown in Fig.5.2).

As we mentioned, all services and implementation components have to be built into service libraries for the future automatic generation process. This process is realized by selecting both the suitable service components and implementation components from the library. Table 7.5 shows the number of service components in the library and the number of used ones in the detailed level simulation SDG of our case study.

Similarly to Fig.5.2, Table 7.5 is organized into 4 service groups: PThread OS+HAL, communication, SMP CPUs and Interconnection+Memory. From the experiment results, we can clearly find that more than half of the services are chosen to be used for our example. The other services may be useful for other application requirements and other architectures.

Multiple abstraction levels with SDG

With the growth of design complexity, architects often increase the abstraction levels to avoid processing all the detailed information together. One obvious problem with a project that uses different abstraction levels is that the different simulation platforms corresponding to the abstraction levels are mainly developed by hand. Besides the time spent on the development, the platform coherency is also a huge challenge for this method. Our solution is using the SDG-based design method to instantiate the codes at all abstraction levels from the service

Table 7.6: Different Code lines at Each Abstraction Levels

Group	Code lines number in RTL/Binary	Code lines number in TLM
SW modules	1,142	974
HW modules	218	93
PThreads+HAL	4,632	0
Communication	1,939	281
SMP CPUs	5,575	0
Interconn+Mem	2,890	0
Total	16,396	1,348

libraries and constraint files. Through this generation flow, the work requested by these abstraction levels could be minimized.

The SDG has the important advantage of representing the structure of the embedded system independently of its abstraction levels. This ability comes from the underlining interface/implementation structure. For every interface components in SDG, the implementation components can be realized in different ways and at several abstraction levels. So without changing the exported APIs, the implementation code can be refined from Transaction Level to RTL/Binary Level easily. Because the APIs used during SW development and the ports used during HW development can generally be kept stable, there is no gap between the TLM and RTL/Binary for application developers. As for the interface and implementation components that are internal to the SDG, they can be changed between different abstraction levels to support different simulation requirements and allow architecture exploration. The modification of internal components should be transparent to the application developers.

Because the different abstraction levels models are very useful for system simulation, it is important to emphasize the simulation ability of an SDG. For the software and hardware developers, the two most interesting points of the simulation are to validate the design code and optimize the system performance. The SDG in different abstraction levels can generate different simulation models. The generated transaction level simulation model can be used for HW/SW co-validation with high simulation speed, while the generated RTL/Binary simulation model can be used for HW/SW performance evaluation with accurate performance data. Because the generation process may be automated, the developer can use both of these models at different stages of the design process.

With the idea of comparing abstraction levels, we manually build the simulation model of our case study at the TLM and RTL/Binary levels separately and compare them together. Table 7.6 calculates the pure code lines by using kloc tools. From the table, we observe an important difference in code lines between transaction level and RTL/Binary level. The results show that the HW/SW interface at transaction level is very concise and abstract compared to RTL/Binary level design. By using almost the same APIs for software modules and ports for HW modules, the realization code of these 10 modules is relatively similar. The difference in the POSIX APIs could be removed by the work of H. Posadas et al. [88]. The last line of the table shows that total code size of TLM is a mere 8.2% of that of RTL/Binary level.

HW/SW Attributes of SDG and parameters exploration

The separate structure of interface and implementation offers an easier control over the HW/SW labeling. Normally, in every SDG, there are three types of implementation components: *pure software components*, *pure hardware components* and *hybrid components*. The idea of pure software components and pure hardware components is clear. The idea of hybrid components is slightly less intuitive. Some functions such as a FIFO function and a Mutex function can be implemented in pure software, pure hardware or one part in software, the other part in hardware.

As illustrated in Fig.5.2, the pure software part includes the POSIX thread OS services and the local FIFOs. The hybrid part contains two services: the CPU service, which provides software execution using a hardware realization, and the global FIFO, which transfers data from the software module to the hardware module. The pure hardware part includes the memory service and the interconnection service.

The HW/SW property of Fig.5.2 is relatively clear except for that of the global FIFO. We should assign the HW/SW property for all hybrid services before the final stage of the design process. From the research works of [60] and [54], we use three kinds of implementation components for global FIFO in our library. All of these FIFO implementations realize the same read and write FIFO interfaces. This translates, in the SDG, into three different implementation components for one interface. Because the first solution uses the hardware FIFO buffer, the performance will be higher when the FIFO depth is large enough. The obvious drawback is that the hardware FIFO needs more chip area. On the other hand, the other two solutions, which use the memory as the FIFO data buffer, may provoke a memory bottleneck for high speed data communication between a SW module and a HW module. With the SDG, this kind of architecture exploration work could be done easily.

The architecture exploration is not limited to HW/SW partitioning only, the parameters configuration is also important to specify an application. In our case, we explore the parameters of the FIFO communication. As we know, two main parameters affect the performance: the *Depth of the FIFO* and the *Width of the FIFO*. The different parameters are only related to the implementation details, and the interfaces provided to the HW/SW programmers should be the same.

In the Motion-JPEG experiment, we focus on the relationship between the FIFO depth and the cycles of the system execution. From this experiment, we can understand how FIFO communication impacts the performance of the whole system. In Fig.7.10, we set up the FIFO depth from 2 slots to 64 slots, and the simulation clock cycles for 5 frames MJPEG decoder are shown in the graph with different CPU numbers.

With this figure, we clearly find that the system performance is related with the FIFO depth. The context switch and scheduling happen when the software FIFOs and the SW/HW FIFOs are full or empty. So with the larger FIFO depth, the system can normally achieve higher performance. The relationship between the system performance and the CPU number is interesting. We find that the highest performance system is that with 2 CPUs. When the CPU number is more than the number of active threads, the bus and memory are partly occupied by traffic due to scheduler activity and inefficient task migration. So the system

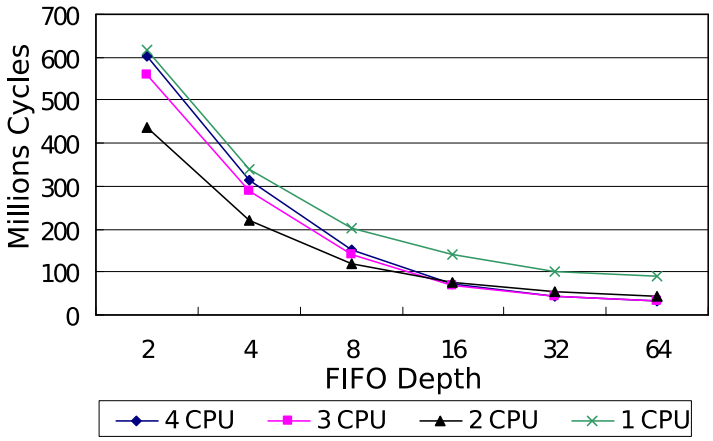

Figure 7.10: Cycles of simulation with the different FIFO depth and different CPU numbers

performance decreases when the CPU number increases.

At the end, we should notice that these different configurations only depend on the implementation of the service. So the SDG method can well cover the architecture and reduce design loops.

7.4.2 Heterogeneous migration

In this section, we use the Motion-JPEG example to show performance and cost advantages of our task migration framework based on heterogeneous MPSoC platforms.

Heterogeneous MPSoC architecture

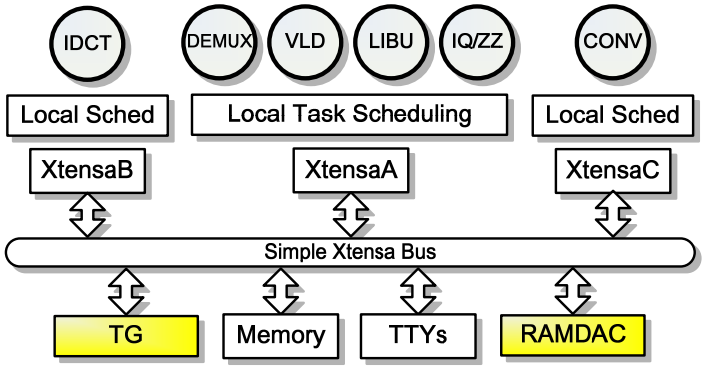

Figure 7.11: Heterogeneous Architecture for the Motion-JPEG Case Study. This system includes three heterogeneous processors and all of them based on the same Xtensa LX2 core instruction set. Processor A is just the basic processor which does not include any extended instructions. Processor B and C include extended instructions for IDCT and CONV separately.

With extended Xtensa processors, our heterogeneous MPSoC architecture used in the following experiments is shown in Fig.7.11.

In the following experiments, we use the *Turbo Xim* simulation mode to profit from the higher simulation speed. The task migration framework has the assumption that all processors should have the coherent data caches to avoid inconsistent cases of communication and task migration. As the Turbo Xim mode does not have the cache effect, it can avoid this problem naturally.

Based on this architecture, we have two different task assignment methods. This traditional one is to porting OS separately for each processor and there are no task migration between different processors. The fixed mapping solution is shown in Fig.7.11. Our migration framework is to let all processors share the same OS image and make task migration possible among different extended processors. Following experimental results are presented to show advantages of our heterogeneous MPSoC task migration framework.

Performance and cost advantage

Task migration can take advantage of the CPU idle time. We use the Motion-JPEG to show, with the same heterogeneous MPSoC architecture, the execution efficiency difference between fixed task mapping and dynamic task migration.

Table 7.7: Comparison of Idle Time for Different Scheduling Frameworks Based on the Same Heterogeneous Architecture

	Fixed Task Assignment	FMFS Algorithm	Most Comp Algorithm	SMP Task Scheduling
Frame/100s	144	288	270	288
Gates	341,773	341,773	341,773	611,295
Perf/Cost	0.50	1.00	0.94	0.56

The results of Table.7.7 compare the performance of four different scheduling solutions. Three of them use the same architecture presented in Fig.7.11 but with different scheduling algorithms. Because of flexibility, performance of both heterogeneous scheduling algorithms overcomes that of the traditional fixed task assignment framework. From this table, we can easily find the performance advantage of the two task scheduling and migration algorithms that we propose (almost 100% higher performance).

We also show that different scheduling algorithms have different performance results. The FMFS algorithm has better performance than the most compatible algorithm because the FMFS one has the simple and efficient realization.

In contrast to these heterogeneous architectures, in the last column, we show the SMP architecture in which each processor includes all extended instructions and registers. Compared with the heterogeneous architecture, the SMP architecture is flexible and high performance. But we should also notice that the hardware cost of this SMP architecture is much higher than the heterogeneous one. For the performance/cost ratio (we normalize the data in Table.7.7), the heterogeneous architecture with our task migration framework is much better than the SMP one.

7.5 Conclusion

The first example presents a hybrid simulation platform specifically designed for configurable processors and MPSoC design. With the comparison of speed and accuracy with our hybrid simulation platform, we confirm that this platform does not sacrifice much accuracy. We show the I/O device modeling flexibility and the HAL APIs porting advantages by using the DMA transfer example. Both flexibility and porting advantages are important properties for system validation at early design stages specifically for configurable processors. Our hybrid simulation platform shows its value for the whole MCPSoC DSE flow.

The next example demonstrates the design space exploration flow based on proposed 4 abstraction levels. From the System level to the Cycle Approximate level, all parameters of the multiple configurable processors SoC architecture are fixed step by step. The simulation speed shows the advantage of using multiple abstraction levels to save both the computation resource and the engineer resource.

SDG is a technology which is created for the modeling of HW/SW interfaces. With the example, we use SDG to model a Mutek based HW/SW interface and help explore the communication architecture. With the complexity growth of the HW/SW interfaces, the SDG may become too large to handle. This limitation makes the SDG modeling technology still far from real usage.

The last experiment presents a task migration framework based on the configurable heterogeneous MPSoC architecture. We show the performance/cost advantage of our framework over existing SMP architectures and fixed task mapping framework. Meanwhile we should also notice that though the heterogeneousness property can help accelerate overall system performance, a large percentage of application tasks only rely on the core instruction set and can be migrated among all execution units. To improve system performance, we should well balance the heterogeneity and homogeneity of one system to well profit from acceleration of extended instruction sets and migration flexibility of the core instruction set.

Chapter 8

Conclusion and Future Works

Compared to off-the-shelf standard processors, the configurable processors provide more flexibilities for optimization of the dedicated applications. Because this configuration property leads to a larger design space, this thesis presents a design methodology and simulation platform specifically designed for this kind of processors. In Chapter 2, we have 6 questions and we give our answers here:

- **Question:** How many abstraction levels should we have and how to integrate software programming interfaces, software simulation technologies and internal modeling together for one abstraction level?

  Answer: In this thesis, we propose 4 different abstraction levels which are *System Level*, *Virtual Architecture Level*, *Transaction Accurate Level* and *Cycle Approximate Level*. At System level, we have no architecture information and all tasks are connected with FIFO channels. The subsystems and inter-subsystem communication is modeled at Virtual Architecture level. The internal components of each subsystem are modeled at Transaction Accurate level. At Cycle Approximate level, all parameters of one MPSoC architecture are fixed. For the software modeling, software are executed directly at both System level and Virtual Architecture level. At Transaction Accurate level, we can execute the software directly with annoted timing information or use the dynamic binary translation technology. Traditional instruction set simulators are used for software simulation at Cycle Approximate level to achieve the highest simulation accuracy. The experiment demonstrates the modeling abilities and simulation speed difference of these 4 abstraction levels.

- **Question:** How to build high level simulation models with high level programming APIs to avoid HdS porting works?

  Answer: In this thesis, we believe that the HdS porting work is the bottleneck of the multiple configurable processors SoC design process. At the instruction set simulator based virtual platforms, we can realize most HAL APIs and device drivers with SystemC modules which are implemented on host machines to avoid much assembly code implementaton. With the semi-hosting functional call provided by many instruction

set simulators, these SystemC modules are invoked when the semi-hosting function is trigged by cross-compiled target binaries. The experiment shows that our method can achieve higher flexibility and avoid much porting works for the modifications of both MPSoC architectures and processor configuration.

- **Questions:** How to define constraints of each abstraction level and refine all high level constraints into lower ones? How to integrate all these abstraction levels together into a single design space exploration flow?

  Answer: In our design space exploration flow, we defined constraints at different granurities. By using the proposed 4 abstraction levels, we can refine the communication between tasks at System level. At Virtual Architecture level, we define the subsystems and map the tasks onto them. The subsystem internal exploration is handled at Transaction Accurate level and all detailed parameters are fixed at Cycle Approximate level. As different abstraction levels rely on different simulation technologies, the design space exploration flow can have different result accuracy and simulation speeds at each step to save the simulation time and the engineer resource.

- **Question:** How to model the HW/SW interface to facilitate the automatic generation and support complex architectures such as the configurable processor based heterogeneous MPSoC platform?

  Answer: In this thesis, we propose to use the Service Dependent Graph for HW/SW interface modeling. By representing the inputs and outputs of both software functions and hardware modules with service providing and service requirement, we can build the relationship of the HW/SW interface components. This modeling method can facilitate the HW/SW interface automatic generation by allowing the system automatically choosing the suitable components from the service dependent graph.

- **Question:** How to make heterogeneous MPSoC more flexible by enabling the task migration function?

  Answer: We find the performance drawback of the fixed task mapping solution for the heterogeneous MPSoC platforms. To improve performance by avoiding the idle time, we choose a special kind of heterogeneous MPSoC platform in which each processor is based on the same core instruction set but with different extended instructions. Following the instruction set compatibility rules, a task can be migrated from one processor to another one with different extended instruction set. The experiments show that our migration framework can efficiently shorten the idle time on the heterogeneous MPSoC platform with only OS modifications.

In the near future, we would like to port more complex applications such as an H.264 encoder and test benches to verify and improve the proposed design methodology. With the advance of simulation technologies, we may also modify the abstraction level definition and the design space exploration flow to profit from the underlying technology evolution. One such direction can be the use of dynamic techniques to benefit from migration between the heterogeneous processors with a more restricted instruction set.

Chapter 9

Résumé en Français

Contents

9.1 Introduction

Avec l'avancée de la micro-électronique, en particulier de l'électronique destinée à la consommation de masse, les concepteurs de systèmes embarqués sont tenus de fournir des solutions répondant aux quatre exigences suivantes:

Performance: un bon produit doit répondre à des critères de performance pour toutes les applications qui s'y exécutent. Avec l'évolution des dernières technologies de compression / décompression vidéo, tels que les normes MPEG 4 et H.264, les protocoles sans fil haute-vitesse ou encore les jeux videos 3D, les exigences en termes de performance de certains systèmes embarqués peuvent même dépasser les capacités de la plupart des ordinateurs de bureau. Afin de répondre à cette exigence de performance, qui croît avec la loi de Moore, l'écart entre les besoins et la performance peuvent être résolus en augmentant le nombre de processeurs et le nombre d'Elements de Calcul spécifiques à l'application (PEs pour Processing Elements).

Consommation d'énergie: les systèmes intégrés doivent remplir leur fonction en consomant le moins de courant d'alimentation possible. Jusqu'à présent, presque tous les appareils portables tels que les téléphones portables à puce, les lecteurs MP3 ou les caméras vidéo utilisent des batteries d'alimentation. Parceque la technologie des batteries croît beaucoup plus lentement que la loi de Moore dans le domaine de la micro-électronique, la consommation d'énergie est l'un des principaux obstacles au développement des appareils portables du futur. Même pour les autres systèmes embarqués tels que les "box" pour télévision et les consoles de jeu vidéo 3D qui sont alimentés par du courant alternatif, la réduction de la consommation d'énergie peut aider à diminuer les nuisances sonores émises par les ventilateurs et améliorer la fiabilité. Les rapports de l'ITRS fournissent plusieurs solutions pour les SoCs destinés aux appareils portables à usage personnel. Ils incluent des optimisations d'architecture à des phases de conception de haut niveau, basées sur des analyses de consommation de puissance, ainsi que des réalisations de MPSoCs avec des PEs spécifiques.

Coût: les concepteurs doivent fournir des produits compétitifs et rentables pour le marché des systèmes embarqués. D'après les prévisions de l'ITRS, le coût du masque sera multiplié par à peu près 20 en dix ans (2018). Dans le même temps, les coûts de développement de logiciels croîssent encore plus rapidement que ceux du matériel de développement. Pour continuer à faire des profits malgré cette tendance et pour réutiliser les mêmes masques pour différentes applications, la méthodologie de conception basée sur des applications de circuits intégrés spécifiques (ASIC), est progressivement remplacée par celle des System-on-Chip (SoC). En utilisant la méthodologie de conception des SoCs, des architectures basées sur plusieurs unités de traitement spécifiques peuvent fournir des solutions plus souples pour un groupe d'applications similaires, permettant ainsi la réutilisation des architectures existantes.

Temps de mise sur le marché: les concepteurs doivent réduire autant que possible le temps de développement pour assurer le succès de la mise sur le marché du produit. Il existe d'énormes écarts entre la loi de Moore pour les circuits intégrés et le développement du matériel/logiciel. Ces écarts de développement font que le produit final ne peut pas suivre la loi de Moore et les contraintes dictées par le marché, ce même en augmentant les

ressources pour le développement. Les plateformes de conception réutilisables, les architectures matérielles souples et la méthodologie de conception au niveau système peuvent être des solutions pour diminuer à la fois le temps de conception et le temps de vérification pour respecter le temps de mise sur le marché.

Ces 4 critères sont apparus il y a plus de 10 ans et ont contribué au développement des méthodologies de conception sur SoC. Les SoC intègrent des éléments de technologie et de conception issus d'autres classes de systèmes parmi une large gamme de produits à base de semi-conducteurs de haute complexité et d'une grande richesse. Dans la conception de SoC, le but est de maximiser le taux de réutilisation des blocs existants, matures et déjà validés, afin de diminuer la phase de conception et de garantir la qualité des nouveaux produits. Les Systèmes sur Puce Multi-Processeurs (MPSoCs), en tant que SoCs, étendent cette idée et plusieurs processeurs génériques (GPPs pour General Purpose Processors) et des unités de calcul spéciales comme les DSP (Digital Signal Processors) pour accroître la performance de calcul de manière générique et flexible. Pour répondre aux contraintes de performance, de consommation, de coût et de temps de mise sur le marché, ces plateformes à base de MPSoCs pourraient être la meilleure solution et sont déjà bien acceptés par les marchés du calcul embarqué et de calcul au sens plus géneral.

L'idée de processeur configurable est de modifier le jeu d'instruction d'un processeur pour accélérer des parties spécifiques à un ensemble d'applications. Etant donné que les systèmes embarqués n'ont besoin de supporter qu'un nombre très réduit d'applications, ils peuvent tirer profit d'un jeu d'instructions spécifique pour atteindre de meilleurs performances sans trop de pertes en termes de coût et de performance. Dans cette thèse, nous aborderons différents niveaux d'abstraction et un flot d'exploration conçu spécialement pour les MPSoCs hétérogènes basés sur des processeurs configurables.

9.2 Problématique

Nous avons introduit dans le chapitre 1 les tendances actuelles des systèmes embarqués, nous aimerions introduire ici l'une des solutions évoquées, les plateformes MPSoC hétérogènes basées sur des processeurs configurables. Cette thèse étudie certains problèmes liés aux processeurs configurables et à l'interface logiciel/matériel de ces plateformes MPSoC hétérogènes.

9.2.1 Processeurs configurables et Systèmes sur Puce Multi-Processeurs (MPSoCs) hétérogènes

Pour évaluer une unité de calcul, il y a deux paramètres importants qui sont le *rapport performance/portes logiques* et la *flexibilité*. Le rapport performance/portes logiques exprime l'efficacité de chaque porte logique pour un système matériel, tandis que la flexibilité exprime la capacité d'un système matériel à s'adapter à différentes applications.

Pour des raisons de généricité, les Processeurs Généralistes **(GPP)** ont normalement le rapport performance/portes logiques le plus faible et la plus grande flexibilité. Au contraire, les systèmes à base d'ASICs (cirrcuits intégrés spécifiques à une application) peuvent utiliser

pleinement les ressources matérielles mais deviennent durs à adapter à d'autres applications. Les processeurs configurables font partie du groupe des processeurs à jeu d'instruction spécifique à une application et présentent un bon compromis entre le rapport performance/portes logiques et flexibilité.

Utiliser de multiples processeurs configurables peut aider le système embarqué à obtenir à la fois du TLP (parallélisme de tâches) et ILP (parallélisme d'instructions). Nous aimerions nous focaliser maintenant sur l'utilisation de l'hétérogénéité de ces systèmes pour améliorer leur performance globale et réduire leur coûts de fabrication.

9.2.2 Motivation

Les processeurs configurables et l'hétérogénéité du système présentant de bonnes propriétés de performance, de consommation d'énergie, de coût et de flexibilité, ils sont de bons candidats pour la prochaine génération de systèmes embarqués. Cependant, les processeurs configurables sont un concept encore relativement nouveau pour la conception de circuits intégrés en industrie. Une architecture générique de MPSoC hétérogènes à partir de laquelle on pourrait dériver tous types de MPSoCs hétérogènes, ainsi qu'un flot de conception associé sont encore l'objet de recherches. Comparée à une approche basée sur un système sur puce monocoeur avec un processeur issu du marché, notre approche présente une plus grande flexibilité pour le concepteur en termes de configuration, d'hétérogénéité et sur des aspects multi-coeurs. Jusqu'à maintenant, il n' y a pas de flot de conception accepté universellement pour bien maîtriser la flexibilité offerte par ce genre de systèmes. Ce travail tente de construire un flot de conception pour des plateformes MPSoC hétérogènes basées sur des processeurs configurables.

En raison du grand nombre de solutions d'implémentation offert par les processeurs configurables et les MPSoCs hétérogènes, une méthode de modélisation et une plateforme de simulation correspondantes sont nécessaires. Dans cette thèse la méthode de modélisation ainsi que les plateformes de simulation sont basées sur un concept de niveau d'abstraction. Avec les niveaux d'abstraction, un système complexe peut être modélisé plus facilement à des niveaux d'abstractions plus élevés et les détails peuvent être représentés à des niveaux d'abstraction plus bas. A un niveau d'abstraction plus haut, la plateforme de simulation est beaucoup plus rapide qu'une plateforme à un niveau plus bas, au prix d'une perte de précision.

Basé sur ces modèles et plateformes de simulation, un flot d'exploration des solutions d'implémentations peut être proposé pour aider les concepteurs à trouver la configuration du processeur et l'architecture MPSoC hétérogène optimales. Dans ce flot d'exploration des solutions d'implémentations, une technologie de modélisation à base de différents niveaux d'abstraction peut aider à régler des détails à plus ou moins haut niveau, et les plateformes de simulation peuvent aider à vérifier à la fois le fonctionnement et la performance à chaque phase d'exploration. Ce flot à différents niveaux d'abstraction peut aussi aider à combler le fossé qui sépare aujourd'hui les besoins et les détails de réalisation en ajoutant plusieurs modèles intermédiaires et en divisant de grandes boucles de conception en de plus petites.

Dans ce flot d'exploration des solutions d'implémentations, nous montrons qu'il reste deux problèmes non résolus. Le problème de *l'interface Matériel/Logiciel* devient extrêmement complexe en raison de la grande complexité et flexibilité des plateformes MPSoC hétérogènes basées sur des processeurs configurables. Le développement d'une interface Matériel/Logiciel devient une partie du flot de conception nécessitant beaucoup de temps et source d'erreurs. Bien que lié au flot d'exploration proposé ci-dessus, le portage de cette interface Matériel/Logiciel est plus liée à des aspects logiciels et a un grand impact sur les performances et la flexibilité du système embarqué complet.

Un autre problème consiste à avoir un système de gestion de tâches qui peut tirer pleinement avantage de ce genre de plateforme MPSoC hétérogènes. Un système de gestion de tâches bien pensé doit pouvoir équilibrer la charge de travail entre les différents processeurs et augmenter le rapport performance/coût pour les systèmes embarqués. Ces trois sections sont nécessaires pour faire des MPSoC hétérogènes à base de processeurs configurables une solution mature et viable pour les applications embarquées dans le futur.

9.2.3 Flot d'exploration des solutions d'implémentation à base de niveaux d'abstractions multiples

L'**Abstraction** est le processus ou résultat d'une généralisation en réduisant l'information contenue dans un concept ou un phénomène observable, typiquement pour retenir uniquement l'information qui est utile pour un but spécifique. Dans un flot d'exploration des solutions d'implémentation, l'abstraction est utilisé pour aider à trouver un compromis entre vitesse de simulation et précision du résultat en termes de performance ou même de fonctionnalité.

Cette propriété a besoin d'être particulièrement creusée pour les processeurs configurables basés sur des processeurs configurables. En nous basant sur les niveaux d'abstraction, nous aimerions bâtir un flot d'exploration des solutions d'implémentations qui puisse profiter pleinement du concept d'abstraction.

Différents cas d'utilisation

Il y a déjà eu plusieurs définitions de niveaux d'abstraction auparavant. Pour tous ces travaux, les différents niveaux d'abstraction ciblent des utilisateurs différents. Par exemple, les développeurs de l'application logicielle peuvent avoir besoin de savoir si les interruptions sont déclenchables et quels traitants d'interruption sont disponibles. En revanche, les développeurs du *logiciel dépendant du matériel* **(HdS)** ont besoin d'implémenter ces traitants en fonction des fonctionnalités offertes par les configurations du matériel. Enfin, les développeurs du matériel implémentent ces fonctionnalités en groupant et combinant des circuits logiques. Dans cette thèse, trois groupes d'utilisateurs sont visés:

- **Les développeurs de l'application logicielle**: Ce groupe de développeurs utilise les modèles de simulation pour aider à concevoir l'application logicielle. Ils ne nécessitent

pas d'informations précises, ils ont juste besoin d'une vue grossière de l'architecture matérielle et utilisent des systèmes d'exploitation, pilotes et bibliothèques existants.

- **Développeurs du logiciel dépendant du matériel**: Ce groupe de développeurs utilise des modèles de simulation pour accélérer et vérifier le travail de portage du logiciel dépendant du matériel. Pour ces développeurs, l'architecture détaillée du processeur et le comportement des composants d'E/S sont très importants pour leur travail. Mis à part le travail de portage des interfaces utilisateur, la plupart du travail de portage du logiciel dépendant du matériel n'a pas besoin d'une très grande vitesse de simulation.

- **Développeurs de matériel**: Ce groupe de développeurs utilise des modèles de simulation pour le développement du matériel et sa vérification. Ces développeurs utilisent des modèles de simulation pour développer des vecteurs de test et prennent les plateformes de simulation comme référence pour la comparaison des résultats de test et la vérification du système matériel. Compte tenu de ces exigeances, le modèle de simulation devrait être très proche de la vraie plateforme matérielle et avec une précision au niveau cycle/bit. La vitesse de simulation n'est pas importante pour l'exécution et la comparaison des vecteurs de test.

Pour ces trois groupes de développeurs, il est clair qu'une unique plateforme de simulation ne peut pas satisfaire ces différents cas d'utilisation. Les niveaux d'abstraction sont la solution pour modéliser la même plateforme matérielle avec des interfaces de programmation et des vitesses et résultats de simulation totalement différents.

Différentes interfaces

Pour ces trois groupes d'utilisateurs, différentes APIs *Application Programmation Interface* devraient être prises en compte. Par exemple, les développeurs d'application logicielle ont besoin d'une interface à un niveau beaucoup plus haut que les développeurs de matériel.

Cette thèse aimerait répondre à la question suivante: *Quelles interfaces doivent être fournies aux différentes catégories d'utilisateurs par les plateformes de simulation à chaque niveau d'abstraction?*

Technologies de simulation différentes

Les différents groupes d'utilisateurs ayant des besoins en vitesse et en précision différents, pour satisfaire le besoin de chacun d'eux, chaque plateforme de simulation doit avoir des technologies de simulation de logiciel différentes pour satisfaire les besoins définis par les utilisateurs

Cette thèse vise à répondre à cette question: *Quelles technologies de simulations doivent être fournies aux différentes catégories d'utilisateurs par une plateforme à un niveau d'abstraction donné?*

Modélisations internes différentes

La partie simulant le logiciel mise à part, la modélisation du matériel est également importante pour une plateforme de simulation. Etant donnés les besoins différents des utilisateurs, différents niveaux de détail devraient être proposés pour chaque niveau d'abstraction.

Cette thèse vise à répondre à cette question: *Quels détails de modélisation interne doivent être fournis aux différentes catégories d'utilisateurs par plateforme de simulation à chaque niveau d'abstraction?*

Flot avec les niveaux d'abstraction

En utilisant plusieurs flots d'exploration des solutions d'implémentation. Jusqu'à maintenant, les méthodes les plus répandues sont celles basées sur des simulations éprouvées et flexibles. La vitesse et le niveau de détail sont les deux propriétés les plus importantes d'une plateforme de simulation, les niveaux d'abstraction sont utilisés pour équilibrer ces deux propriétés. Le flot d'exploration des solutions d'implémentation devrait utiliser correctement tous ces niveaux d'abstraction et diminuer le coût des boucles de conception, élevé dans les flots traditionnels.

Cette thèse se concentre sur l'hétérogénéité et la configuration des processeurs Ces deux propriétés exigent de définir comment bien intégrer ces propriétés dans les différents niveaux d'abstraction. Comme ces propriétés augmentent énormément le champ de solutions d'implémentations par rapport à des architectures MPSoC traditionnelles, nous voulons également savoir comment contrôler ces propriétés à des niveaux d'abstraction bas pour maîtriser la taille du champ d'implémentations, et comment utiliser ces deux propriétés pour éviter des boucles de conception durant ce processus.

Cette thèse vise à répondre à la question suivante: *Comment intégrer tous ces niveaux d'abstraction dans un flot d'exploration des solutions d'implémentation?*

9.2.4 Modélisation d'interface Matériel/Logiciel et plateforme de simulation de MPSoC hybrides

(L'interface Matériel/Logiciel) est utilisée pour décrire la connection entre les modules de l'application logicielle et les modules fonctionnels du matériel dans les systèmes sur puce embarqués.

Comme cette thèse se concentre sur le flot d'exploration des solutions d'implémentation basé sur des processeurs configurables et des architectures MPSoC hétérogènes, la difficulté de définir une interface Matériel/Logiciel en raison de la configuration et de l'hétérogénéité devrait être résolue pour simplifier le flot d'exploration complet.

Modélisation de l'interface Matériel/Logiciel

Vu la complexité des processeurs configurables et des architectures MPSoC hétérogènes, *l'interface Matériel/Logiciel* devient critique dans le processurs complet de développement

du système. Durant le processus de flot d'exploration des solutions d'implémentation, les processeurs de l'interface Matériel/Logiciel ont besoin d'être configurés pour répondre aux exigences du logiciel applicatif. Ceci est encore plus difficile lorsque les architectures et processeurs configurables sont modifiés durant le processus, car la partie du HdS sous-jacente doit être modifiée fréquemment et mène à des problèmes de maintenance et de portabilité.

Une solution possible pour ce problème complexe est de générer l'interface Matériel/-Logiciel pour une plateforme spécifique à une application automatiquement. Dans le domaine des systèmes embarqués, la génération automatique du HdS et de la plateforme matérielle est appelée processus de *Synthèse au Niveau Système*. Malheureusement, ce problème n'est pas bien formalisé est n'est pas facile à formaliser du tout. Bien que la plupart des idées de synthèse sont des extensions de la *Synthèse au Niveau registre (RTL)*, la synthèse au niveau système ne bénificie pas des théories d'algèbre booléenne et de logique binaire. Sans contexte ou relations mathématiques forts, la synthèse au niveau système demeure une tâche de conception.

Cette thèse voudrait répondre à cette question:

Comment modéliser l'interface Matériel/Logiciel pour faciliter la génération automatique et le support d'architectures complexes comme des plateformes MPSoC hétérogènes basées sur des processeurs configurables

Interface de programmations (APIs) de haut niveau

La génération automatique des interfaces Matériel/Logiciel n'est pas l'unique solution, nous pouvons envisager d'autres méthodes pour faciliter l'intégration du matériel et du logiciel. Durant les premières phases du processus d'exploration, ces modifications nécessitent de nombreuses mises à jour de l'implémentation du HdS pour valider à la fois les applications et la plateforme MPSoC hétérogène. Par exemple, la réalisation du code assembleur demande beaucoup d'efforts et est source d'erreurs lors des premières phases de conception. Dans le même temps, nous observons également que le HdS n'occupe normalement qu'une petite partie des ressources de calcul. Rendre la réalisation du HdS compatible pour l'exploration des solutions d'implémentation dans les premières phases devient une question ouverte pour les concepteurs de plateforme à un haut niveau d'abstraction. En marge de ces questions, comment intégrer correctement les interfaces logicielles de haut niveau avec différentes technologies de simulations est aussi une question à laquelle cette thèse essaie de répondre.

9.2.5 Le flot de migration des tâches pour les architectures MPSoC hétérogènes

En plus de la modélisation complexe du HdS et du problème de génération, il est également difficile de concevoir un HdS qui permette une migration des tâches flexible avec une haute performance pour des MPSoC hétérogènes.

Plateforme de MPSoC hétérogène rigide

Malgré que les MPSoC hétérogènes aient beaucoup de désavantages que nous avons précedemment expliqués, ils ne sont pas aussi flexibles que les plateformes MPSoC basées sur une architecture SMP (Symmetric Multi-Processing). Habituellement, sur des plateformes MPSoC hétérogènes, chaque tâche est mise en mémoire sur un processeur spécifique au lieu d'un groupe de processeurs comme c'est le cas sur les plateformes SMP. Cette nécessité de mise en mémoire rigide fait que le partitionnement de la tâche est très complexe ou alors que les processus sont mal répartis sur les processeurs.

Représentation de la répartition des élements logiciels sur les élements matériels

Il y a plusieurs modèles de calculs pour des systèmes embarqués multiprocesseurs tels que le modèle de *Flot de données synchrone* **(SDF)** [72] et le modèle de *réseaux de Kahn* **(KPN)** [62]. Comme ces modèles ne sont pas dépendant de la sous-couche plateforme MPSoC, la mise en mémoire de ces tâches/processus et de chaque processeur n'est pas définie. Pour une plateforme MPSoC hétérogène basée sur des processeurs configurables, parceque les processeurs sont différents entre eux, les solutions de mise en mémoire existantes varient peu pour tous les modèles de calculs.

Dans cette thèse, nous posons six questions sur les flots d'exploration des solutions d'implémentation basés sur des processeurs configurables, auxquels nous devons répondre.

- Combien de niveaux d'abstraction devrions nous avoir et comment intégrer les interfaces de programmation logicielle, les technologies de simulation logicielle et la modélisation interne ensemble pour un niveau d'abstraction?

- Comment définir les contraintes de chaque niveau d'abstraction et raffiner toutes les contraintes de haut niveau vers un niveau plus bas.

- Comment intégrer tous ces niveaux d'abstraction en un seul flot d'exploration des solutions d'implémentation?

- Comment modéliser l'interface matérielle/logicielle pour faciliter la génération automatique et permettre des architectures complexes tels que les plateformes MPSoC hétérogènes constituées de processeurs configurables?

- Comment construire des modèles de simulation de haut niveau avec des API de haut niveau de programmation afin d'éviter le travail de portabilité du HdS?

- Comment rendre les MPSoC hétérogènes plus flexible en permettant la migration des tâches?

9.3 Flot d'exploration des solutions d'implémentation avec de multiples niveaux d'abstraction

Beaucoup de chercheurs se sont penchés sur les niveaux d'abstraction et proposent différentes définitions à partir de différents points de vue. En particulier, les architectures MPSoC hétérogènes configurables sont plus complexes que les architectures classiques. Pour rendre la définition des niveaux d'abstraction plus facile à définir, dans cette section, nous définissons trois axes qui nous permettent de classifier les niveaux d'abstraction des MPSoC. Avec ces trois axes, cette section va présenter quatre différents modèles de niveaux d'abstraction pour les MPSoC. Ces quatre niveaux d'abstraction sont ciblés pour différents groupes d'utilisateurs avec différentes interfaces de programmation logicielle. En plus de ces interfaces, des plateformes de simulation utilisent différentes technologies pour chaque niveaux qui peut proposer un bon compromis entre la vitesse de la simulation et la précision des résultats. Les méthodes de modélisation matérielle sont de plus détaillées dans cette section pour différents niveaux d'abstraction.

Dans cette section, nous voulons également fournir quelques solutions pour choisir une architecture de MPSoC hétérogène configurable pour déterminer une architecture logiciel/-matériel qui respecte les contraintes de conception du système. Avec la flexibilité des processeurs configurables et la propriété d'hétérogénéité, il y a une infinité de moyens pour affiner une architecture de MPSoC. Toutes ces possibilités mènent à une exploration de beaucoup de solutions d'implémentations. Pour être capable de trouver une solution à portée de main à un problème de conception de systèmes, nous présentons une politique basée sur l'idée de budget, qui transforme des contraintes spécifiques et rafinent tous les budgets au travers de quatre niveaux d'abstraction.

9.3.1 Quatre niveaux d'abstraction pour les MPSoC hétérogèes configurables

Niveau Système

Dans cette thèse, le *Niveau Système* **(SL)** est le niveau d'abstraction le plus élevé utilisé dans l'étape de conception. Parceque la clé dans la conception des MPSoC est le parallélisme de l'application, ce niveau d'abstraction est conçu pour fournir un environnement d'exécution parallèle. Cette application est décrite dans un ensemble de processus qui s'échangent des données exclusivement en utilisant les canaux de type *First In First Out* **(FIFO)**, qui sont bloquants et directs. Ces processus combinés aux canaux de cette façon sont connus en tant que *KPN*. En plus de ce modèle de calculs, il existe un flot de données statique. Ces deux modèles ainsi que d'autres peuvent être adaptés à ce Niveau Système pour valider le parallélisme au niveau logiciel.

Au Niveau Système, il n'y a pas de différence entre les processus logiciels et les modules matériels. Tous ceux-ci sont généralement des fils d'exécution codés à haut niveau qui communiquent entre eux avec des APIs de communication. Plusieurs APIs de communications

les plus couramment utilisées sont `channelInit`, `channelRead` et `channelWrite`. L'API est appelée par le processus principal pour créer un canal de communication inter-processus. `ChannelRead` et `channelWrite` sont utilisés par les processus pour lire et écrire des données.

Niveau d'Architecture Virtuel

Le Niveau d'Architecture Virtuel est un niveau d'abstraction qui est plus détaillé que le Niveau Système. La principale différence est que le Niveau d'Architecture Virtuel inclue des informations sur l'architecture. A ce niveau, l'architecture est représentée à un niveau de granularité grossier et abstrait le système réel en *Sous-systèmes* et *Communications Inter-sous-systèmes*. Le système est composé de plusieurs sous-systèmes qui peuvent être divisés en sous-systèmes avec processeur **(CPUSS)** et sous-systèmes matériels **(HWSS)**. Tous ces sous-systèmes sont connectés entre eux en utilisant le module de communication inter-sous-systèmes.

L'objectif de ce Niveau Système est de fournir un modèle d'exécution qui peut donner des informations plus précises sur les communications inter-sous-systèmes avec une haute rapidité de simulation.

Les sous-systèmes sont des modèles qui ne prennent pas en compte le temps et qui sont compilés sur et pour des machines hôtes afin d'augmenter significativement le temps des simulations sytèmes. Mais la communication inter-sous-systèmes peut être précise au cycle ou au temps donné par le modèle **TLM** (*Transaction Level Modeling*) afin de fournir des informations détaillées sur le temps pour l'exploration de la communication.

Niveau "Transaction Accurate"

Ce niveau est créé pour la modélisation interne de sous-systèmes avec des informations relatives au temps. Un sous-système logiciel inclue généralement une application disposant de plusieurs fils d'exécution, d'un système d'exploitation, de processeurs, mémoires, communications inter-systèmes, ... etc. Le niveau a pour but de modéliser tous ces composants internes avec le TLM afin d'accélérer la simulation, au prix de la précision. La perte de précision doit être quantifiée par des expériences.

Le Niveau "Transaction Accurate" doit gérer plusieurs interfaces: une du côté logiciel qui utilise des API au niveau de la couche d'abstraction matérielle (HAL) ou des fils d'exécution POSIX et l'autre du côté matériel qui utilise des protocoles matériels. Au-dessus de la couche d'abstraction matérielle, le système d'exploitation, les librairies de communications entrantes et sortantes (E/S) et les applications logicielles disposant de plusieurs fils d'exécution sont ajoutées. Avec les mêmes APIs au niveau du HAL, l'effort pour réécrire les programmes au bas niveau d'assemblage en explorant les jeux d'instructions des différents processeurs est réduit. Parceque le système d'exploitation et l'application logicielle sont annotées temporellement et exécutées directement avec le Niveau "Transaction Accurate", la vitesse de simulation du MPSoC est plus élevée qu'avec les plateformes virtuelles traditionnelles basées sur les simulateurs disposant de jeux d'instructions.

Au Niveau "Transaction Accurate", les composants logiciels et matériels d'un MPSoC sont faciles à intégrer pour une plateforme exécutable avec une rapidité de simulation élevée et une précision estimée relativement bonne [45]. Toutes ces propriétés rendent le Niveau Précis de Transaction adapté pour l'exploration des MPSoC hétérogènes configurables.

Niveau de Cycle Approximé (CA)

Le niveau de *Cycle Approximé* **CA** est le niveau d'abstraction le plus détaillé dans notre flot d'exploration des solutions d'implémentation. Le but de ce niveau est de fournir des résultats très précis de la simulation au prix d'une vitesse plus faible. Cela signifie que toutes les informations détaillées concernant l'architecture devraient apparaître dans ce modèle de simulation.

Du côté matériel, nous avons une approximation des cycles des jeux d'instructions du simulateur pour l'exécution logicielle. Avec ces jeux d'instruction, nous avons le modèle cache avec sa taille ainsi que celle de sa ligne, et les paramètres d'associativité. Etant donné que l'interface de programmation logicielle du Niveau "Cycle Approximate" est dans un format binaire, tout le HdS et les applications devraient être compilées directement en binaire cible. Pour connecter chaque processeur, nous proposons deux méthodes de connexion. L'une est un modèles de connexions en SystemC, précises au bit près et l'autre un modèle du niveau de transaction. Les composants matériels doivent être implémentés avec des modules en SystemC pour fournir des informations précises quant à la durée d'exécution et ce pendant le déroulement de la simulation.

Comparaison des niveaux d'abstraction

Le Tableau.9.1 résume ce chapitre et compare les quatre niveaux d'abstraction. Ce tableau est également utile pour le flot d'exploration des solutions d'implémentation des multiples niveaux d'abstraction présentés dans le chapitre suivant.

9.3.2 Flot d'exploration basé sur le budget

En règle générale, le *budget* se réfère à une liste de tous les revenus et dépenses. Dans le domaine de la synthèse logique, le budget de la durée d'exécution est utilisé pour choisir les composants logiques adaptés pour des exigences au niveau temporel [46]. Dans le flot d'exploration des solutions d'implémentation, le budget désigne les ressources que chaque module ou sous-système peut utiliser pour construire le système global. Par exemple la consommation de courant est un paramètre important pour la plupart des architectures MPSoC. Dans le flot d'exploration des solutions d'implémentation, le budget "courant" pour toute la puce peut être divisée en plusieurs budgets plus petits pour chacun des sous-systèmes et modules. Avec ce procédé de raffinage, les détails de ces derniers peut être fixé à la fin de ce flot. Bien que la notion de budget soit supposée fonctionner avec un ajout de contraintes, cette propriété ne peut malheureusement pas toujours être vraie. Par exemple, ajouter une

Table 9.1: Comparison of Different Abstraction Levels

	Niveau Système	Niveau d'architecture virtuelle	Niveau "Transaction accurate"	Niveau "Cycle accurate"
Tâches logicielles	Code C/C++	Code C/C++	Code C/C++	Code C/C++
Interface de programmation logicielle	APIs de communication	APIs de communication	APIs de fils d'exécution POSIX ou APIs HAL	Format binaire
Technologie de simulation logic.	Exécution native	Exécution native	Traduction des binaires dynamique	Interprétation des instruction pendant l'exécution
Composants matériels	Fils d'exécution natifs	SC_Thread	Modules SystemC	Modules SystemC
Interface de communication matérielle	APIs de communication	APIs de communication	TLM ou VCI	TLM ou VCI
Architecture MPSoC	Pas encore défini	Inter-subsystem communication	Subsystem details	Jeux d'instructions fixés
Précision de la simulation	Pas d'information sur le temps d'exécution	Communication Inter-sous-systèmes	Cycle Approximate	Cycle Approximate

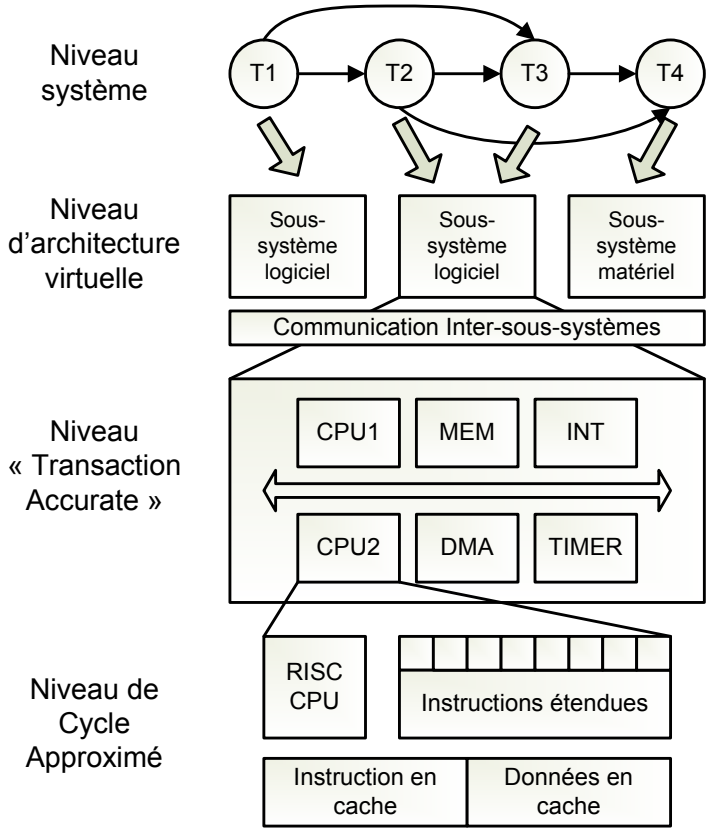

Figure 9.1: Flot d'exploration des solutions d'implémentation pour les différents niveaux d'abstraction

exigence de bande passante conduira vers une croissance exponentielle avec des retards pour certains types d'architecture de communication.

Notre flot d'exploration des solutions d'implémentation pour les différents niveaux d'abstraction suit l'approche de partionnement des problèmes basés sur le budget. La figure 9.1 présente tout le flot depuis le niveau d'abstraction le plus élevé (SL) au niveau le plus détaillé (CA). Au niveau système, les programmeurs optimisent les modèles de calculs tels que ceux de KPN ou SDF pour les propriétés liées à la communication. En utilisant les environnements exécutables fournis par les modèles du niveau SL, les programmeurs peuvent minimiser la taille de la communication inter-tâches et optimiser les propriétés de communication afin d'améliorer les performances du système. Au niveau VA, les programmeurs créent une architecture multi sous-systèmes basique et relient tous les processus de ce niveau à chaque sous-système. Après avoir exploré la communication inter-sous-systèmes avec une vitesse de simulation très rapide, les programmeurs peuvent obtenir la taille de la communication et transférer un motif de cette communication. En utilisant ces statistiques, les programmeurs peuvent donner un type de communication inter-sous-systèmes à choisir entre un bus, une maille ou un réseau sur une puce et obtenir le taux d'utilisation de ce module de communication.

Après que l'optimisation de la communication entre les sous-systèmes soit bien optimisée au niveau VA, nous continuons d'explorer chaque sous-système au niveau TA. Parce

que chaque sous-système logiciel général inclue des CPUs, mémoires, bus et autres modules, son architecture est plus complexe que celle des sous-systèmes matériels. Au niveau TA, tous les modules complexes dans un sous-système logiciel sont modélisés et construits ensembles. Les données de performance du logiciel sont basées sur les annotations du niveau CA avec quelques statistiques. Les résultats de la simulation fournis à ce niveau aident les programmeurs à concevoir le générateur et modifier les périphériques matériels d'un sous-système. Après que les composants aient été fixés pour chaque sous-système, nous pouvons automatiquement générer les processeurs configurables et modifier la taille du cache pour respecter les contraintes du niveau TA. Etant donné que le niveau CA est le niveau d'abstraction le plus détaillé, tous les paramètres matériels de cette solution pour les MPSoC hétérogènes configurables sont fixés et les APIs HAL sont soigneusement réalisés en accord avec les jeux d'instruction des processeurs obtenus. Tout le flot d'exploration des solutions d'implémentation est fini quand un modèle CA obtenu respecte toutes les contraintes budgétaires.

Pour résumer, notre flot d'exploration des solutions d'implémentation basé sur la notion de budget peut convenir au procédé de raffinage avec quatre niveaux d'abstraction MPSoC.

- Raffinage de la communication au *niveau système* **(SL)**.
- Raffinage des sous-systèmes au *niveau d'Architecture Virtuelle* **(VA)**.
- Raffinage des sous-systèmes internes au *niveau "Transaction Accurate"* **(TA)**.
- Extension des instructions du CPU et raffinage du cache au *niveau de "Cycle Approximate"* **(CA)**.

9.3.3 Conclusion

En intégrant les interfaces de programmation logicielle ainsi que les technologies de modélisation matérielle et logicielle, nous pouvons construire un flot d'exploration des solutions d'implémentation à plusieurs niveaux d'abstraction efficace et rapide. Avec la méthodologie de raffinage basée sur la notion de budget, beaucoup de grandes boucles deviennent plus petites en reliant le niveau d'abstraction actuel à celui de plus haut niveau le plus proche. Bien que l'implémentation détaillée de cette méthodologie est toujours dépendente des applications réelles, l'utilisation de ces multiples niveaux d'abstraction et un moyen d'aider les programmeurs à atteindre leur objectifs plus facilement.

9.4 Modélisation de l'interface logiciel/matériel et plateforme de simulation MPSoC hybride

Pour simplifier le flot d'exploration des solutions d'implémentation, en particulier pour séparer le processus d'adaptation du système d'exploitations, sur l'architecture et l'optimisation de l'application, nous créons une nouvelle plateforme de simulation hybride qui peut fournir

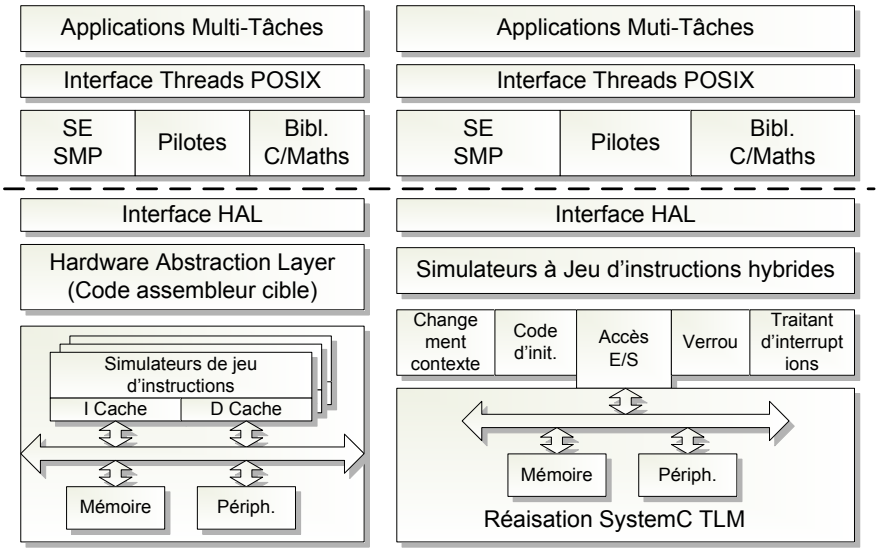

Figure 9.2: Plateforme hybride de simulation de MPSoC
A gauche: la plateforme de simulation classique à base dun simulateur de jeu d'instruction.
A droite: la simulation hybride avec l'exécution native du HAL.

une vitesse de simulation beaucoup plus rapide et des résultats de simulations assez précis sans avoir à rentrer dans trop de détails d'implémentations non nécessaires.

Dans la figure 9.2, nous avons les plateformes de simulation traditionnelles à gauche. A droite de la figure 9.2, nous montrons la plateforme de simulation hybride avec l'interface de programmation du HAL. Dans cette plateforme, nous utilisons le même système d'exploitation, la même bibliothèque de communications et applications que dans les simulateurs classiques. Dans cette figure, la différence la plus importante est que la plateforme de simulation hybride remplace l'implémentation en code assembleur du HAL sur le simulateur de jeu d'instruction par des modules SystemC sur la machine hôte.

D'un point de vue technique, nous avons besoin d'une méthode pour laisser les interfaces outrepasser le simulateur de jeu d'instruction et atteindre les modules SystemC sous-jacents. Dans notre plateforme de simulation hybride, nous proposons l'usage d'une *Interface "Semi Hosting"* [3] pour les simulateurs de jeux d'instructions. L'interface semi-hosting peut être réalisée avec une instruction spécifique qui peut être utilisée uniquement pendant la simulation. Quand le simulateur de jeux d'instruction récupère cette instruction, au lieu de la décoder et de l'exécuter directement, il appelle une fonction de "hooking" de l'hôte pour l'exécuter. Le "hooking" est une technique qui utilise ce que l'on appellera des "hooks" pour effectuer une série de procédures pour traiter un évènement. Après que l'évènement traité se produit, le flot de contrôle suit la série dans un ordre spécifique. En général, la plupart des simulateurs utilisent ce moyen pour implémenter certains appels de fonctions qui ne sont pas réalisés dans les environnements embarqués cibles comme dans le simulateur MIPS SPIM [21] et autres. Dans notre plateforme, cette fonction hook invoque le module hôte correspondant qui peut modifier le statut du processeur et des données mémoire. Après le retour du module hôte, l'application embarquée peut présenter le même flot d'exécution que la réalisation classique.

Cette thèse présente une plateforme de simulation hybride spécifiquement conçue pour résoudre le problème d'adaptation du HdS dans la conception de l'interface Matériel/Logiciel. Cette plateforme hybride montre la flexibilité de modélisation des périphériques d'E/S et les avantages en termes du portages de l'interface du HAL. Comme la flexibilité et la facilité de portage sont deux propriétés importantes pour la validation du système lors des premières phases de conception, notre plateforme de simulation hybride montre ses qualités pour le flot d'exploration des solutions d'implémentation entier.

9.5 Flot de migration des tâches pour les architectures MPSoC hétérogènes

Migrer une tâche d'un processeur à un autre est considéré comme une fonctionalité clé d'un SE supportant un système basé sur des *Multi-processeurs Symétriques* (SMP). Cette capacité peut équilibrer la charge de travail parmis différents coeurs pour réduire le temps d'inactivité et atteindre une performance globale plus élevée.

Comme les formats des jeux d'instructions et les fichiers de registres sont totalement différents dans les plateformes MPSoC hétérogènes classiques, il est impossible d'avoir de la migration de tâche dans celles-ci. Par exemple, le haut de la Fig. 6.1 représente une plateforme MPSoC hétérogène générale qui inclue différentes sortes de MCUs (microcontrôleurs), des DSPs et d'autres types de processeurs. Etant donné que les jeux d'instruction des MCUs et DSPs ne sont pas compatibles, les tâches compilées et assignées à des MCUs ne peuvent pas être migrées sur les DSPs même si les DSPs sont inactifs et en attente de nouvelles tâches. Ce genre d'architectures basées sur des sous-systèmes est généralement utilisé pour la plupart des plateformes hétérogènees existantes.

Les propriétés d'hétérogénéité et l'équilibrage de la charge de travail étant essentielles pour les plateformes MPSoC modernes, nous proposons un cadre de travail qui peut fournir à la fois des unités de calcul hétérogènes et une capacité de migration de tâches. La contrainte clé de la migration de tâches réside dans le fait que le logiciel système comme le SE et les pilotes devraient être capables de s'exécuter sur tous les processeurs due l'architecture, nous faisons en sorte que tous les processeurs partage un même coeur d'instructions et de registres pour le logiciel système. En plus de cela, les instructions de calcul et les registres (pour l'usage de l'application) peuvent être totalement différents entre chaque processeur pour fournir une accélération spécifique pour différentes applications. Avec ce cadre de travail, l'avantage que présentent l'hétérogénéité et la mgration de tâches peuvent être tous les deux exploités par notre plateforme.

Du point de vue de la réalisation, nous utilisons une architecture MPSoC hétérogène basée sur des processeurs configurables. Comme tous les processeurs étendus partagent le même coeur d'instructions qui peuvent être utilisées pour l'implémentation du système d'exploitation satisfaisant toutes les contraintes imposées au SE et aux communications.

Le bas de la Fig.9.3 décrit une telle plateforme où tous les processeurs configurables exécutent le même système d'exploitation et les tâches peuvent être migrées entre différents

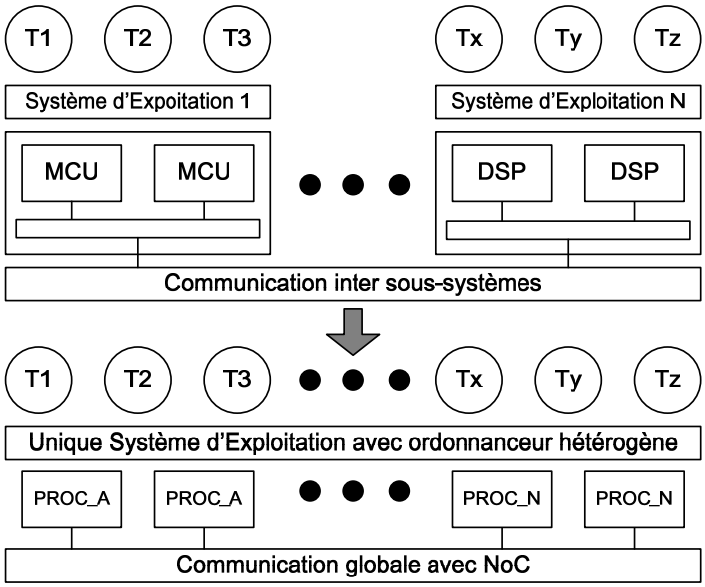

Figure 9.3: Architectures Matériel/Logiciel des MPSoC hétérogènes.
En haut: architecture basée sur de multiples sous-systèmes En bas: architecture basée sur plusieurs processeurs étendus différents

types de processeurs. Cette thèse aimerait donner une définition formelle et plus de détails sur la solution de migration de tâches adoptée. Nous y décrivons un flot de migration de tâches qui se concentre sur les architectures qui sont généralement utilisées pour les systèmes MPSoC à base de processeurs configurables. Les avantages que présentent l'hétérogénéité et la flexibilité de type SMP peut être atteinte avec ce flot de migration de tâches.

9.6 Conclusion et travaux futurs

Comparés aux processeurs standards du marché, les processeurs configurables procurent une plus grande flexibilité pour l'optimisation des applications dédiées. Etant donné que cette propriété de configuration mène à un plus grand espace d'implémentations, cette thèse présente une méthodologie de conception et de plateformes de simulation spécifiquement conçues pour ce type de processeurs. Dans le chapitre 2, nous posons 6 questions dont nous donnons la réponse ici:

- **Question:** Combien de niveaux d'abstraction devrions nous avoir et comment intégrer ensemble les interfaces de programmation, les technologies de simulation logicielles et la modélisation interne pour un niveau d'abstraction donné?

  Réponse: Dans cette thèse, nous proposons 4 différents niveaux d'abstraction qui sont le *Niveau Système*, *Niveau architecture virtuelle*, *Niveau "Transaction accurate"* et *Niveau "Cycle approximate"* Au niveau système, nous n'avons aucune information d'architecture et toutes les tâches sont connectées par des canaux FIFOs. Les sous-systèmes et communications inter-sous-système sont modélisés au niveau Architecture Virtuelle. Les composants internes de chaque sous-système sont modélisés au

niveau "Transaction Accurate". Au niveau "Cycle Approximate", tous les paramètres sont fixés pour une architecture MPSoC. Pour la modélisation logicielle, le logiciel est exécuté directement au niveau Système et Architecture virtuelle. Au niveau "Transaction accurate", nous pouvons exécuter le logiciel directement avec des informations de temps ou utiliser des méthodes de traduction du binaire dynamiques. Les simulateurs de jeux d'instruction classiques sont utilisés pour la simulation logicielle au niveau "Cycle approximate", pour atteindre la plus grande précision de simulation. L'expérience montre les capacités de modélisation et la difference de vitesse de simulation de ces quatre niveaux d'abstraction.

- **Question:** Comment construire des modèles de simulation de haut niveau avec des interfaces de programmation de haut niveau pour éviter le travail de portage du HdS?

  Réponse: Dans cette thèse, nous soutenons que le travail de portage du HdS est le goulot d'étranglement du processus de conception de SoC basé sur de multiples processeurs configurables. Pour les plateformes virtuelles basées sur des simulateurs de jeu d'instruction, nous pouvons implémenter la plupart des interfaces du HAL et les pilotes de périphérique avec des modules SystemC qui sont implémentées sur des machines hôtes pour éviter trop d'implémentations en langage assembleur. Avec les appels de fonction à exécution partielle sur la machine hôte fournie par de nombreux simulateurs de jeux d'instructions, ces modules SystemC sont appelés quand l'exécution partielle d'une fonction sur la machine hôte est déclenchée par le binaire cross-compilé cible. L'expérience montre que notre méthode peut atteindre une plus grande flexibilité et éviter beaucoup de travail de portage pour modifier à la fois les architectures MPSoC et la configuration du processeur.

- **Questions:** Comment définir les contraintes de chaque niveau d'abstraction et raffiner toutes les contraintes de haut niveau en contraintes de plus bas niveau? Comment intégrer tous ces niveaux d'abstraction dans un unique flot d'exploration des solutions d'implémentation?

  Réponse: Dans notre flot d'exploration des solutions d'implémentation, nous définissons des contraintes à différents niveaux de granularité. En utilisant les 4 niveaux d'abstraction proposés, nous pouvons raffiner la communication inter-tâches au niveau système. Au niveau architecture virtuelle, nous définissons les sous-systèmes et répartissons les tâches dessus. L'exploration des sous-systèmes internes est traité au niveau "Transaction accurate" et tous les paramètres détaillés sont fixés au niveau "Cycle approximate". Etant donné que les différents niveaux d'abstraction reposent sur différentes technologies de simulation, le flot d'exploration des solutions d'implémentation peut fournir différents résultats et vitesses de simulation à chaque étape pour économiser du temps de simulation et des heures ingénieur.

- **Question:** Comment modéliser l'interface matériel/logiciel pour faciliter sa génération automatique et supporter les architectures complexes comme les platformes MPSoC hétérogènes basées sur des processeurs configurables?

Réponse: Dans cette thèse, nous proposons d'utiliser un graphe de dépendance de services (GDS) pour la modélisation matériel/logiciel. En représentant les entrées et sorties des fonctions logicielles et des modules matériels, avec une offre de service d'une part et un besoin de service d'autre part. Cette méthode de modélisation peut faciliter la génération automatique de l'interface matériel/logiciel en permettant au système de choisir automatiquement les bons composants à partir du SDG.

- **Question:** Comment rendre les MPSoCs hétérogènes plus flexibles en activant une fonctionnalité de migration de tâches?

  Réponse: La solution d'une répartition statique des tâches n'est pas optimale pour les plateformes MPSoC hétérogènes. Pour améliorer la performance en évitant l'inactivité des processeurs, nous choisissons un type spécial de plateformes hétérogènes dans lequel chaque processeur est basé sur un même coeur de jeu de simulation mais avec différentes instructions étendues. En respectant les règles de compatibilité des jeux d'instruction, une tâche peut être migrée d'un processeur sur l'autre avec différentes extensions du jeu d'instruction. L'expérience montre que notre flot de migration des tâches peut réduire efficacement le temps d'inactivité sur la plateforme MPSoC hétérogène avec uniquements des modifications dans le système d'exploitation.

Dans un futur proche, nous aimerions porter des applications plus complexes comme l'encodeur H264 et des applications de test pour vérifier et améliorer la méthodologie de conception proposée. Avec les avancées des technologies de simulation, nous pouvons aussi modifier la définition des niveaux d'abstraction et du flot d'exploration du champ de solution d'implémentations pour bénéficier de l'évolution des technolgies sous-jacentes. Une direction d'investigation possible peut être l'utilisation de techniques d'exécution dynamiques pour bénéficier de la migration entre les processeurs hétérogènes avec un jeu d'instruction plus réduit.

Bibliography

[1] ITRS System Drivers technonical report 2007. [Online]. Available: http://www.itrs.net/Links/2007ITRS/Home2007.htm. (document), 1, 1.1, 1.2, 1.4

[2] ITRS Design technonical report 2007. [Online]. Available: http://www.itrs.net/Links/2007ITRS/Home2007.htm. (document), 1, 1.3

[3] Tensilica Inc., Xtensa Microprocessor, 2008. [Online]. Available: http://www.tensilica.com. (document), 2.3, 2.1.1, 2.4, 3.1.3, 4.3.4, 5.3, 5.3, 6.2.2, 7.1.2, 7.1.3, 9.4

[4] H.264: Advanced video coding for generic audiovisual services. [Online]. Available: http://www.itu.int/rec/T-REC-H.264/en. 1

[5] Arc International, ARCtangent Processor, 2008. [Online]. Available: http://www.arc.com. 2.1.1

[6] ARM Corporate, Cortex(TM)-A8 processor, 2008. [Online]. Available: http://www.arm.com. 2.1.1, 7.1.2

[7] ARM Corporate, NEON(TM) technology, 2008. [Online]. Available: http://www.arm.com. 2.1.1, 6.2.2

[8] ARM Corporate, Jazelle(R) technology, 2008. [Online]. Available: http://www.arm.com. 2.1.1

[9] ROMA: Reconfigurable Operators for Multimedia Applications. Participants : Emmanuel Casseau, Shafqat Khan, Daniel Ménard, François Charot, Christophe Wolinski, Erwan Raffin, Olivier Sentieys. [Online]. Available: https://roma.irisa.fr/. 2.1.1

[10] Texas Instruments Incorporated, OMAP Platform for Wireless Handset Solutions, 2008. [Online]. Available: http://www.ti.com. 2.1.2, 3.3.1

[11] Texas Instruments Incorporated, Davinci Platform for Digital Video and Audio Solutions, 2008. [Online]. Available: http://www.ti.com/corp/docs/landing/davinci/index.html. 2.1.2

[12] STMicroelectronics, Nomadik Platform, 2008. [Online]. Available: http://www.st.com. 2.1.2, 3.3.1

[13] SystemC 2.2.0, Open SystemC Initiative (OSCI) 2008. [Online]. Available: http://www.systemc.org. 3.1.2, 7.1.4

[14] TLM 2.0, Open SystemC Initiative (OSCI) 2008. [Online]. Available: http://www.systemc.org/downloads/standards/tlm20/. 3.1.2, 5.3.1

[15] SoCLib, "A modeling and simulation platform for system on chip", 2008. [Online]. Available: http://www.soclib.fr/Home.html. 3.1.3, 3.2.2, 7.1.4

[16] CoWare Inc., CoWare Virtual Platform, 2008. [Online]. Available: http://www.coware.com. 3.1.3, 3.2.1, 3.2.2

[17] eCos Operating System, 2008. [Online]. Available: http://ecos.sourceware.org. 3.2.1, 3.2.2, 4.1.1, 5.5

[18] Virtutech Inc., Virtutech Simics Platform, 2008. [Online]. Available: http://www.virtutech.com. 3.2.1

[19] ARM Corporate, RealView System Generator, 2008. [Online]. Available: http://www.arm.com/products/DevTools/SystemGenerator.html. 3.2.1

[20] Intel Inc., X86 Series Processor, 2008. [Online]. Available: http://www.intel.com. 4.1.1

[21] SPIM: A MIPS32 Simulator, James Larus. [Online]. Available: http://pages.cs.wisc.edu/ larus/spim.html. 5.3, 9.4

[22] MPSoC with 6 Heterogeneous Configurable Processors for the Printer and Scanner Solution. [Online]. Available: http://www.tensilica.com/markets/customers/printers_scanners.htm. 6

[23] POSIX Thread, IEEE Standard 1003.1 (The IEEE and The Open Group) 2008. [Online]. Available: http://www.opengroup.org/onlinepubs/009695399/basedefs/pthread.h.html. 7.1.2

[24] ARM Corporate., AHB 3 Specification, 2008. [Online]. Available: http://www.arm.com/products/solutions/AMBA3AXI.html. 7.3.1

[25] *Draft Standard omi 324: PI-Bus, rev. 0.3d, open microprocessor systems initiative. Siemens AC. Munich.*, 1994. [Online]. Available: http://www.cordis.lu/esprit/src/omihome.htm. 7.3.1

[26] Avaro, O., A. Eleftheriadis, C. Herpel, G. Rajan, and L. Ward: *Mpeg-4 systems: Overview*. Signal Processing: Image Communication, Elsevier, 15:281–298, Feb. 2000. 1

[27] Balakrishnan, S., R. Rajwar, M. Upton, and K.K. Lai: *The impact of performance asymmetry in emerging multicore architectures*. In *ISCA*, pp. 506–517. IEEE Computer Society, 2005. 3.3.1

[28] Balarin, F., Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A.L. Sangiovanni-Vincentelli: *Metropolis: An integrated electronic system design environment.* IEEE Computer, 36(4):45–52, 2003. 3.1.3

[29] Becchi, M. and P. Crowley: *Dynamic thread assignment on heterogeneous multiprocessor architectures*. In *Conf. Computing Frontiers*, pp. 29–40. ACM, 2006, ISBN 1-59593-302-6. 3.3.1

[30] Bellard, F.: *Qemu, a fast and portable dynamic translator*. In *USENIX Annual Technical Conference, FREENIX Track*, pp. 41–46. USENIX, 2005. 3.1.1, 4.3.4

[31] Bergamaschi, R.A. and W.R. Lee: *Designing system-on-chip using cores*. In *Proceeding of DAC 2000*. Los Angeles, California, June 5-9 2000. 3.2.1

[32] Bershad, B.N., C. Chambers, S.J. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E.G. Sirer: *Spin - an extensible microkernel for application-specific operating system services*. Operating Systems Review, 29(1):74–77, 1995. 3.2.2

[33] Birnbaum, M. and H. Sachs: *How vsia answers the soc dilemma*. Computer, 32(6):42–50, Jun 1999, ISSN 0018-9162. 4.2.2

[34] Brucker, P.: *Scheduling Algorithms*. Springer-Verlag, 2007, ISBN 9783540695158. 3.3.1

[35] Cai, L. and D. Gajski: *Transaction level modeling: an overview*. First IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, 2003., pp. 19–24, Oct. 2003. 3.1.3, 3.2.2

[36] Cai, L., A. Gerstlauer, and D. Gajski: *Retargetable profiling for rapid, early system-level design space exploration*. Design Automation Conference, 0:281–286, 2004. 3.1.1

[37] Calandrino, J.M., D. Baumberger, T. Li, S. Hahn, and J.H. Anderson: *Soft real-time scheduling on performance asymmetric multicore platforms*. In *RTAS '07: Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium*, pp. 101–112, Washington, DC, USA, 2007. IEEE Computer Society, ISBN 0-7695-2800-7. 3.3.1

[38] Cassidy, A.S., J.M. Paul, and D.E. Thomas: *Layered, multi-threaded, high-level performance design*. In *DATE*, pp. 10954–10959. IEEE Computer Society, 2003, ISBN 0-7695-1870-2. 3.1.3, 4.3

[39] Cesário, W.O., G. Nicolescu, L. Gauthier, D. Lyonnard, and A.A. Jerraya: *Colif: A design representation for application-specific multiprocessor socs*. IEEE Design & Test of Computers, 18(5):8–20, 2001. 3.2.1

[40] Cmelik, R.F. and D. Keppel: *Shade: A fast instruction-set simulator for execution profiling*. In *SIGMETRICS*, pp. 128–137, 1994. 3.1.1

[41] Cong, J., Y. Fan, G. Han, and Z. Zhang: *Application-specific instruction generation for configurable processor architectures*. In *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pp. 183–189, New York, NY, USA, 2004. ACM, ISBN 1-58113-829-6. 2.1.1

[42] Ebcioğlu, K. and E.R. Altman: *Daisy: dynamic compilation for 100% architectural compatibility*. In *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*, pp. 26–37, New York, NY, USA, 1997. ACM, ISBN 0-89791-901-7. 3.1.1

[43] Etiemble, D., S. Bouaziz, and L. Lacassagne: *Customizing 16-bit fp instructions on a nios ii processor for fpga image and media processing*. In Miranda, M. and S. Ha (eds.): *ESTImedia*, pp. 61–66. IEEE Computer Society, 2005, ISBN 0-7803-9347-3. 2.1.1

[44] Gauthier, L., S. Yoo, and A.A. Jerraya: *Automatic generation and targeting of application-specific operating systems and embedded systems software*. IEEE Trans. on CAD of Integrated Circuits and Systems, 20(11):1293–1301, 2001. 3.2.2

[45] Gerin, P., X. Guérin, and F. Pétrot: *Efficient implementation of native software simulation for mpsoc*. Design, Automation and Test in Europe Conference and Exhibition, 2008. DATE '08, pp. 676–681, 10-14 Mar. 2008. 4.2.3, 4.3.4, 9.3.1

[46] Geurts, W., F. Catthoor, and H.D. Man: *Time constrained allocation and assignment techniques for high throughput signal processing*. In *DAC*, pp. 124–127, 1992. 4.3, 9.3.2

[47] Ghiasi, S., T.W. Keller, and F.L.R. III: *Scheduling for heterogeneous processors in server systems*. In Bagherzadeh, N., M. Valero, and A. Ramírez (eds.): *Conf. Computing Frontiers*, pp. 199–210. ACM, 2005, ISBN 1-59593-018-3. 3.3.1

[48] Goodwin, D. and D. Petkov: *Automatic generation of application specific processors*. In Moreno, J.H., P.K. Murthy, T.M. Conte, and P. Faraboschi (eds.): *CASES*, pp. 137–147. ACM, 2003, ISBN 1-58113-676-5. 2.1.1

[49] Goodwin, D., C. Rowen, and G. Martin: *Configurable multi-processor platforms for next generation embedded systems*. In *ASP-DAC*, pp. 744–746. IEEE, 2007. 2

[50] Govil, K., E. Chan, and H. Wasserman: *Comparing algorithm for dynamic speed-setting of a low-power cpu*. In *MOBICOM*, pp. 13–25, 1995. 2.1.2

[51] Gschwind, M.: *Instruction set selection for asip design*. Proceedings of the Seventh International Workshop on Hardware/Software Codesign, 1999. (CODES '99), pp. 7–11, 1999. 2.1.1

[52] Hennessy, J.L. and D.A. Patterson: *Computer Architecture : A Quantitative Approach*. Morgan Kaufmann Publishers, fourth ed., 2007. chapter 2 and chapter 4. 1, 2.1.1, 3.3.1

[53] Hofstee, H.P.: *Power efficient processor architecture and the cell processor*. In *Proceeding of the* 11^{th} *Int'l Symposium on High-Performance Computer Architecture (HPCA-11 2005)*. San Francisco, February 12-16 2005. 1, 3.3.1

[54] Hommais, D.: *Une méthode d'évaluation et de synthèse des communications dans les systèmes intégrés matériel-logiciel*. PhD thesis, Université Paris VI, 2001. 3.1.1, 7.4.1

[55] Hwang, Y., S. Abdi, and D. Gajski: *Cycle-approximate retargetable performance estimation at the transaction level*. Design, Automation and Test in Europe, 2008. DATE '08, pp. 3–8, March 2008. 3.1.1, 3.2.2

[56] Iseli, C. and E. Sanchez: *Spyder: a reconfigurable vliw processor using fpgas*. FPGAs for Custom Computing Machines, 1993. Proceedings. IEEE Workshop on, pp. 17–24, Apr 1993. 2.1.1

[57] Ivanek, F.: *Convergence and competition on the way toward 4g [from the editor's desk]*. Microwave Magazine, IEEE, 9(4):6–14, Aug. 2008, ISSN 1527-3342. 1

[58] Jain, M., M. Balakrishnan, and A. Kumar: *Asip design methodologies: survey and issues*. VLSI Design, 2001. Fourteenth International Conference on, pp. 76–81, 2001. (document), 2.2

[59] Jerraya, A., A. Bouchhima, and F. Pétrot: *Programming models and hw-sw interfaces abstraction for multi-processor soc*. Design Automation Conference, 2006 43rd ACM/IEEE, pp. 280–285, 0-0 2006, ISSN 0738-100X. 3.1.3, 4.2

[60] Jerraya, A., F. Rousseau, A. Bouchhima, M.W. Youssef, A. Grasset, W. Cesario, L. Kriaa, and A. Sarmento: *Service dependency graph: an efficient model for hardware/software interfaces modeling and generation for soc design*. Hardware/Software Codesign and System Synthesis, 2005. CODES+ISSS '05. Third IEEE/ACM/IFIP International Conference on, pp. 261–266, Sept. 2005. 7.4.1

[61] Jerraya, A. and W. Wolf: *Hardware/software interface codesign for embedded systems*. Computer, 38(2):63–69, Feb. 2005, ISSN 0018-9162. (document), 2.6

[62] Kahn, G.: *The semantics of simple language for parallel programming*. In *IFIP Congress*, pp. 471–475, 1974. 2.5.2, 3.1.3, 4.1.1, 9.2.5

[63] Kangas, T., P. Kukkala, H. Orsila, E. Salminen, M. Hännikäinen, T.D. Hämäläinen, J. Riihimäki, and K. Kuusilinna: *Uml-based multiprocessor soc design framework*. ACM Trans. Embedded Comput. Syst., 5(2):281–320, 2006. 3.1.3

[64] Keutzer, K., S. Malik, and A.R. Newton: *From asic to asip: The next design discontinuity*. In *ICCD*, pp. 84–90. IEEE Computer Society, 2002, ISBN 0-7695-1700-5. 2.1.1

[65] Kongetira, P., K. Aingaran, and K. Olukotun: *Niagara: A 32-way multithreaded sparc processor*. IEEE Micro, 25(2):21–29, 2005, ISSN 0272-1732. 4.1.1, 7.1.2

[66] Kraemer, S., L. Gao, J. Weinstock, R. Leupers, G. Ascheid, and H. Meyr: *Hysim: a fast simulation framework for embedded software development*. In Ha, S., K. Choi, N.D. Dutt, and J. Teich (eds.): *CODES+ISSS*, pp. 75–80. ACM, 2007, ISBN 978-1-59593-824-4. 3.2.2

[67] Kukkala, P., J. Riihimäki, M. Hännikäinen, T.D. Hämäläinen, and K. Kronlöf: *Uml 2.0 profile for embedded system design*. In *Proceedings of 8th Design, Automation and Test in Europe (DATE 2005)*. Munich, Germany, March 7-11 2005. 3.2.1

[68] Kumar, R., D.M. Tullsen, P. Ranganathan, N.P. Jouppi, and K.I. Farkas: *Single-isa heterogeneous multi-core architectures for multithreaded workload performance*. In *ISCA*, pp. 64–75. IEEE Computer Society, 2004, ISBN 0-7695-2143-6. 3.3.1

[69] La Rosa, A., L. Lavagno, and C. Passerone: *Hardware/software design space exploration for a reconfigurable processor*. Design, Automation and Test in Europe Conference and Exhibition, 2003, pp. 570–575, 2003, ISSN 1530-1591. 2.1.1

[70] Lajolo, M., M. Lazarescu, and A. Sangiovanni-Vincentelli: *A compilation-based software estimation scheme for hardware/software co-simulation*. Hardware/Software Codesign, 1999. (CODES '99) Proceedings of the Seventh International Workshop on, pp. 85–89, 1999. 3.2.2

[71] Le Moullec, Y., J.P. Diguet, and J.L. Philippe: *Design-trotter: a multimedia embedded systems design space exploration tool*. Multimedia Signal Processing, 2002 IEEE Workshop on, pp. 448–451, Dec. 2002. 4.3

[72] Lee, E. and D. Messerschmitt: *Synchronous data flow*. Proceedings of the IEEE, 75(9):1235–1245, Sept. 1987, ISSN 0018-9219. 2.5.2, 4.1.1, 9.2.5

[73] Leupers, R.: *Retargetable Code Generation for Digital Signal Processors*. Kluwer Academic Publishers, Norwell, MA, USA, 1997, ISBN 0792399587. 6.2.2

[74] Martin, G.: *Overview of the mpsoc design challenge*. In Sentovich, E. (ed.): *DAC*, pp. 274–279. ACM, 2006, ISBN 1-59593-381-6. 3.3.1

[75] Massas, P.G. de, P. Amblard, and F. Pétrot: *On sparc leon-2 isa extensions experiments for mpeg encoding acceleration*. VLSI Des., 2007(1):1–1, 2007, ISSN 1065-514X. 2.1.1

[76] Mirapuri, S., M. Woodacre, and N. Vasseghi: *The mips r4000 processor*. Micro, IEEE, 12(2):10–22, Apr 1992, ISSN 0272-1732. 4.1.1, 7.1.2

[77] Mohanty, S., V.K. Prasanna, S. Neema, and J.R. Davis: *Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation*. In *LCTES-SCOPES*, pp. 18–27. ACM, 2002, ISBN 1-58113-527-0. 3.1.3

[78] Moigne, R.L., O. Pasquier, and J.P. Calvez: *A generic rtos model for real-time systems simulation with systemc*. In *DATE*, pp. 82–87. IEEE Computer Society, 2004, ISBN 0-7695-2085-5. 3.1.1

[79] Moore, G.: *Cramming more components onto integrated circuits*. Proceedings of the IEEE, 86(1):82–85, Jan 1998, ISSN 0018-9219. 1

[80] Muttreja, A., A. Raghunathan, S. Ravi, and N.K. Jha: *Hybrid simulation for energy estimation of embedded software*. IEEE Trans. on CAD of Integrated Circuits and Systems, 26(10):1843–1854, 2007. 3.2.2

[81] Nakamura, Y., K. Hosokawa, I. Kuroda, K. Yoshikawa, and T. Yoshimura: *A fast hardware/software co-verification method for system-on-a-chip by using a c/c++ simulator and fpga emulator with shared register communication*. In Malik, S., L. Fix, and A.B. Kahng (eds.): *DAC*, pp. 299–304. ACM, 2004, ISBN 1-58113-828-8. 3.2.2

[82] Paolucci, P.S., A.A. Jerraya, R. Leupers, L. Thiele, and P. Vicini: *Shapes: : a tiled scalable software hardware architecture platform for embedded systems*. In Bergamaschi, R.A. and K. Choi (eds.): *CODES+ISSS*, pp. 167–172. ACM, 2006, ISBN 1-59593-370-0. 2.1.2, 3.3.1

[83] Paulin, P.G., C. Pilkington, and E. Bensoudane: *Stepnp: A system-level exploration platform for network processors.* IEEE Design & Test of Computers, 19(6):17–26, 2002. 3.1.3, 3.2.2

[84] Petrot, F. and P. Gomez: *Lightweight implementation of the posix threads api for an on-chip mips multiprocessor with vci interconnect*. Design, Automation and Test in Europe Conference and Exhibition, 2003, pp. 51–56 suppl., 2003, ISSN 1530-1591. 7.1.2

[85] Pham, D., T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P. Harvey, H. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D. Stasiak, M. Suzuoki, O. Takahashi, J. Warnock, S. Weitzel, D. Wendel, and K. Yazawa: *Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor*. Solid-State Circuits, IEEE Journal of, 41(1):179–196, Jan. 2006, ISSN 0018-9200. 3.3.1

[86] Pimentel, A.D., C. Erbas, and S. Polstra: *A systematic approach to exploring embedded system architectures at multiple abstraction levels*. IEEE Trans. Computers, 55(2):99–112, 2006. 3.1.3

[87] Pimentel, A.D., L.O. Hertzberger, P. Lieverse, P. van der Wolf, and E.F. Deprettere: *Exploring embedded-systems architectures with artemis*. Computer, 34(11):57–63, 2001, ISSN 0018-9162. 4.3.4, 5.1

[88] Posadas, H., J. Ádamez, P. Sánchez, E. Villar, and F. Blasco: *Posix modeling in systemc*. In *proceedings of ASP-DAC'06*. Pacifico Yokohama, Yokohama, Japan, Jan. 24-27 2006. 7.4.1

[89] Posadas, H., F. Herrera, P. Sanchez, E. Villar, and F. Blasco: *System-level performance analysis in systemc*. Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings, 1:378–383 Vol.1, Feb. 2004, ISSN 1530-1591. 3.1.1

[90] Pozzi, L., K. Atasu, and P. Ienne: *Exact and approximate algorithms for the extension of embedded processor instruction sets*. IEEE Trans. on CAD of Integrated Circuits and Systems, 25(7):1209–1229, 2006. 2.1.1, 3.1.3, 6.2.2

[91] Reyes, V., T. Bautista, G. Marrero, P.P. Carballo, and W. Kruijtzer: *Casse: A system-level modeling and design-space exploration tool for multiprocessor systems-on-chip*. In *DSD*, pp. 476–483. IEEE Computer Society, 2004, ISBN 0-7695-2203-3. 3.1.3

[92] Rosenblum, M., E. Bugnion, S. Devine, and S.A. Herrod: *Using the simos machine simulator to study complex computer systems*. ACM Trans. Model. Comput. Simul., 7(1):78–103, 1997. 3.1.1

[93] Sato, J., M. Imai, T. Hakata, A.Y. Alomary, and N. Hikichi: *An integrated design environment for application specific integrated processor*. In *ICCD*, pp. 414–417. IEEE Computer Society, 1991, ISBN 0-8186-2270-9. 2.1.1

[94] Senouci, B., A. Bouchhima, F. Rousseau, F. Petrot, and A. Jerraya: *Fast prototyping of posix based applications on a multiprocessor soc architecture: "hardware-dependent software oriented approach"*. Rapid System Prototyping, 2006. Seventeenth IEEE International Workshop on, pp. 69–75, June 2006, ISSN 1074-6005. (document), 7.2

[95] Sun, F., S. Ravi, A. Raghunathan, and N.K. Jha: *A scalable synthesis methodology for application-specific processors*. IEEE Trans. VLSI Syst., 14(11):1175–1188, 2006. 2.1.1, 3.1.3, 6.2.2

[96] Szymanek, R., F. Catthoor, and K. Kuchcinski: *Time-energy design space exploration for multi-layer memory architectures*. Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings, 1:318–323 Vol.1, Feb. 2004, ISSN 1530-1591. 4.3

[97] Van Praet, J., G. Goossens, D. Lanneer, and H. De Man: *Instruction set definition and instruction selection for asips*. High-Level Synthesis, 1994., Proceedings of the Seventh International Symposium on, pp. 11–16, May 1994. 2.1.1

[98] Wagner, F.R., W.O. Cesário, and A.A. Jerraya: *Hardware/software ip integration using the roses design environment*. ACM Trans. Embedded Comput. Syst., 6(3), 2007. 3.2.1

[99] Wirthlin, M., B. Hutchings, and K. Gilson: *The nano processor: a low resource reconfigurable processor*. FPGAs for Custom Computing Machines, 1994. Proceedings. IEEE Workshop on, pp. 23–30, Apr 1994. 2.1.1

[100] Yen, T.Y. and W. Wolf: *Performance estimation for real-time distributed embedded systems*. IEEE Trans. Parallel Distrib. Syst., 9(11):1125–1136, 1998. 3.1.1

[101] Yoo, S., G. Nicolescu, L. Gauthier, and A.A. Jerraya: *Automatic generation of fast timed simulation models for operating systems in soc design*. In *DATE*, pp. 620–627. IEEE Computer Society, 2002, ISBN 0-7695-1471-5. 3.1.1

[102] Yu, H., A. Gerstlauer, and D. Gajski: *Rtos scheduling in transaction level models*. First IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, 2003., pp. 31–36, Oct. 2003. 3.1.1

[103] Zimmermann, G.: *The mimola design system a computer aided digital processor design method*. In *DAC '79: Proceedings of the 16th Conference on Design automation*, pp. 53–58, Piscataway, NJ, USA, 1979. IEEE Press. 2.1.1, 2.1.1

Glossaire

A

API Application Programming Interface.
An application programming interface is a set of declarations of the functions that an operating system, library or service provides to support requests made by computer programs.

ASIC Application-Specific Integrated Circuit.
An application-specific integrated circuit is an integrated circuit customized for a particular use, rather than intended for general-purpose use.

C

CA Cycle Approximate.
CA level is the most detailed abstraction level in our multiple abstraction levels design space exploration flow.

CPU Central Processing Unit.
A central processing unit is a description of a class of logic machines that can execute computer programs.

D

DSP Digital Signal Processor.
A digital signal processor is a specialized microprocessor designed specifically for digital signal processing, generally in real-time computing.

F

FIFO First In, First Out.

First in first out is an abstraction in ways of organizing and manipulation of data relative to time and prioritization.

G

GPP General Purpose Processor.
A general purpose processor is a processor that is not tied to, or integrated with, a particular language or piece of software.

H

HAL Hardware abstraction layer.
A hardware abstraction layer is an abstraction layer, implemented in software, between the physical hardware of a computer and the software that runs on that computer. Its function is to hide differences in hardware from most of the operating system kernel, so that most of the kernel-mode code does not need to be changed to run on systems with different hardware.

HdS Hardware-dependent Software.
Hardware-dependent software is the part of an operating system which varies across microprocessor boards and is comprised notably of device drivers and boot code which performs hardware initialization.

I

IDCT Inverse Discrete Cosine Transform.
A inverse discrete cosine transform expresses the opposite process of transforming a sequence of finitely many data points in terms of a sum of cosine functions oscillating at different frequencies.

ILP Instruction-Level Parallelism.
Instruction-level parallelism is a measure of how many of the operations in a computer program can be performed simultaneously.

ISR Interrupt Service Routine.
An interrupt service routine is a callback subroutine in an operating system or device driver whose execution is triggered by the reception of an interrupt.

ISS Instruction Set Simulator.

An Instruction Set Simulator is a simulation model which mimics the behavior of a mainframe or microprocessor by "reading" instructions and maintaining internal variables which represent the processor’s registers.

ITRS International Technology Roadmap for Semiconductors.
The International Technology Roadmap for Semiconductors, known throughout the world as the ITRS, is the fifteen-year assessment of the semiconductor industry’s future technology requirements. These future needs drive present-day strategies for world-wide research and development among manufacturers’ research facilities, universities, and national labs.

J

JPEG Joint Photographic Experts Group.
JPEG is a commonly used method and standard of compression for photographic images and is created by the joint photographic experts group.

M

MIMD Multiple Instruction stream, Multiple Data stream.
Multiple instruction stream multiple data stream is a technique employed to achieve thread level parallelism.

MJPEG Motion JPEG.
In multimedia, motion JPEG is an informal name for multimedia formats where each video frame or interlaced field of a digital video sequence is separately compressed as a JPEG image.

MPSoC Multiprocessor System-on-chip.
The multiprocessor system-on-chip is a system-on-chip which uses multiple processors, usually targeted for embedded aplications.

O

OS Operating System.
An operating system is the software component of a computer system that is responsible for the management and coordination of activities and the sharing of the resources of the computer.

R

RISC Reduced Instruction Set Computer.
Reduced instruction set computer represents a CPU design strategy emphasizing the insight that simplified instructions which can be executed very quickly to provide higher performance.

RTL Register Transfer Level.
In integrated circuit design, register transfer level description is a way of describing the operation of a synchronous digital circuit. In RTL design, a circuit's behavior is defined in terms of the flow of signals (or transfer of data) between hardware registers, and the logical operations performed on those signals.

S

SIMD Single Instruction, Multiple Data.
Single instruction multiple data is a technique employed to achieve data level parallelism, as in a vector processor.

SMP Symmetric Multiprocessing.
Symmetric multiprocessing involves a multiprocessor computer architecture where two or more identical processors can connect to a single shared main memory.

SoC System-on-Chip.
System-on-chip refers to integrating all components of a computer or other electronic system into a single integrated circuit (chip).

T

TA Transaction Accurate.
TA level is created for subsystem internal modeling with timing information.

TLM Transaction-Level Modeling.
Transaction-level modeling is a high-level approach to modeling digital systems where details of communication among modules are separated from the details of the implementation of functional units or of the communication architecture.

TLP Thread-Level Parallelism.
Thread-level parallelism is a form of parallelization of computer code across multiple processors in parallel computing environments. It focusses on distributing execution processes (threads) across different parallel computing nodes.

V

VA Virtual Architecture.

The VA level is a more detailed abstraction level compared with the system level. The most important difference is that the VA level includes some architecture information.

VCI Virtual Component Interface.

Virtual component interface is a standard defined by the Virtual Socket Interface Alliance (VSIA). The overall objective is to obtain a general interface, such that Intellectual Property (IP), in the shape of Virtual Components (VCs) of any origin, can be connected to Systems-on-Chips of any chip integrator.

Abstract: In the consumer electronics domain, we propose to use the multiprocessor system-on-chip solution with configurable processors and heterogeneous architectures to satisfy all the requirements of performance, power consumption, cost and time-to-market. In this thesis, the heterogeneous architectures are defined as a group of processors which are based on the same core instruction set with different extensions. Because of the configurability and the heterogeneousness, the design space becomes extremely large and includes both software and hardware optimizations. To solve this problem, we build a design space exploration flow using 4 abstraction levels with different details and simulation speed. As we find that the *Hardware-dependent Software* **(HdS)** is the bottleneck in this flow, a hybrid simulation method is introduced to avoid these HdS adaptation efforts. To give realization high performance and flexibility, we propose one task migration framework in which one task can be executed on several compatible processors with different extended instructions. A Motion-JPEG case study is used to validate all these works.

Key words: System-on-Chip, multiple processors, configurable processors, heterogeneousness, abstraction levels, budget, design space exploration flow, hybrid simulation and task migration.

Résumé: Dans le domaine de l'électronique pour la consommation de masse, les concepteurs sont tenus de fournir des systèmes embarqués qui doivent satisfaire des exigences de performance, de consommation, de coût et de temps de mise sur le marché. Pour satisfaire toutes ces exigences, nous nous concentrons sur les systèmes sur puce multi-processeurs (MPSoCs) à base de processeurs configurables. Dans cette thèse, les architectures hétérogènes sont définies comme des architectures multiprocesseur à base de processeurs qui sont basées sur le même jeu d'instructions avec des extensions différentes. Cette thèse tente de résoudre certaines des difficultés causées par l'utilisation de processeurs configurables et les architectures hétérogènes. Dans de telles architectures, le champ des solutions d'implémentation devient extrêmement large et inclut aussi bien des optimisations logicielles que des optimisations matérielles. C'est pourquoi nous présentons 4 niveaux d'abstraction différents avec des niveaux de détail et des vitesses de simulation différentes pour faciliter l'exploration des différentes solutions d'implémentation. Une méthode de simulation hybride est également intégrée à ces niveaux d'abstraction pour éviter les efforts d'adaptation du logiciel dépendant du matériel durant l'exploration. Pour que l'implémentation choisie soit hautement performante et flexible, nous proposons un schéma de migration de tâches dans lequel une tâche peut être exécutée sur plusieurs processeurs compatibles avec différentes extensions d'instructions. Une application décodeur Motion-JPEG a été utilisée pour valider ces travaux.

Mots clés: Système sur Puce, multi-processeurs, processeurs configurables, hétérogénéité, niveaux d'abstraction, budget, flot d'exploration des solutions d'implémentations, simulation hybride et migration de tâches.

ISBN: 978-2-84813-129-30