Pilot – A Platform-Based HW/SW Synthesis System for FPSoC*

Zhong Chen[†], Jason Cong[‡], Yiping Fan[‡], Xun Yang[‡], Zhiru Zhang[‡]

[†]Department of Computer Science & Technology, Peking University, Beijing, 100871, China

[‡]UCLA Computer Science Department, UCLA, Los Angeles, CA 90095, USA

ABSTRACT

This paper presents Pilot, a platform-based HW/SW synthesis system for field programmable System-on-a-Chip (FPSoC). It starts from a system-level design specification and targets at FPSoC platforms. In order to automate this process as much as possible, a System-level Data Model (SDM) is built in Pilot to provide a basis for developing system-level HW/SW synthesis algorithms and an abstraction for accepting different types of design specifications. A preliminary HW/SW co-design flow based on SDM is also proposed. Many key issues such as profiling, HW/SW partitioning, scheduling, interface generation, and code generation are addressed. A JPEG encoder is used as an example to demonstrate the design flow. The experimental results are given for the evaluation of Pilot.

Keywords

Pilot, FPSoC, Platform-Based Design, System-level Data Model, Model of Computation

1. INTRODUCTION

A System-on-a-Chip (SoC) platform is a family of micro-architectures that is essentially "fixed" so that it supports a substantial re-use of software [8]. The basic components of the platform, including a number of predetermined processors, memory blocks, on-chip bus architectures, various IP cores, etc., should be unchanged during the design process. The main benefit of starting with an SoC platform is that the component designs, including CPU, memory and some common peripherals, are already optimized and verified. Designs based on such platforms are able to utilize a stable, core-based architecture that can be rapidly extended and customized for a range of applications. Derivatives can be easily created by adding different software or hardware to this groundwork. The designers using the platform only need to focus on creating their value-added parts of the system. This significantly saves the design time and further ensures first-time design success.

For the purpose of fully exploiting the benefits of the SoC platforms, platform-based design [8] was recently introduced as a solution that would balance production cost with design time. The tenet behind the platform-based design approach is to take advantage of the predefined micro-architectures, thereby avoiding designing a chip from scratch. Several research projects have been propelled along this trend. The "Mescal" project [8] by UC Berkeley and Princeton is working to define a methodology and design environment for application-specific fully programmable platforms. The "Metropolis" project [10] by UC Berkeley is trying to employ a formal design methodology and generic mechanism to model arbitrary communication and computation semantics and model the heterogeneous system. By going from the highest level of abstraction down to the implementation through a series of refinements, a refined system will eventually be mapped to a given platform.

In this paper, we present a platform-based HW/SW synthesis system called Pilot. Instead of facing the problem of platform creation, which is generally difficult, Pilot looks at how to effectively map a system-level design specification onto a given SoC platform, in particular, the FPSoC platform. A FPSoC platform integrates microprocessors, RAMs, programmable blocks, and several peripherals in a programmable device. Such a programmable device is a viable solution for rapid prototyping of a complex system. Due to limited on-chip resources for a selected platform, efficient system level synthesis algorithms such as scheduling, partitioning, communication synthesis, etc., are needed to tradeoff performance for cost of the system. Pilot tries to tackle these issues and provide a framework for optimizing and synthesizing designs at the system level.

The rest of the paper is organized as follows. Section 2 gives an overview of the Pilot framework and presents the design framework. Section 3 introduces the SDM that serves as the internal representation of the whole system. Sections 4 and 5 discuss the Pilot input language and candidate platform, respectively. Section 6 explores the details of our HW/SW implementation methodology. In Section 7, a JPEG encoding system is used as an example to go through our platform-based synthesis flow. Section 8 describes the current status of our Pilot system. Section 9 concludes the paper.

2. OVERVIEW OF THE PILOT SYSTEM

2.1 Objective

Our Pilot synthesis system starts from system-level design specification, and targets at FPSoC platform. To be more precise, Pilot tries to provide synthesis capabilities to map a system-level design specification to a given FPSoC platform. The goal is to automate this design process as much as possible, and explore the solution space of simultaneously optimizing the HW/SW implementation of an application at the system level.

2.2 System Organization

An SDM is developed to provide a basis for system-level HW/SW synthesis algorithms. It also serves as a unified internal

* This research is partially supported by the National Science Foundation under the award CCR-0096383, the MARCO/DARPA Gigascale Silicon Research Center (GSRC) and Altera Corporation.

representation of the system to capture the design throughout the whole synthesis process. The core of SDM is the FunState [5] Model of Computation (MoC). It is able to represent the heterogeneous HW/SW system formally, abstractly and unambiguously. In addition to FunState MoC, we have supplementary information stored in SDM to capture the platform and input language-specific characteristics.

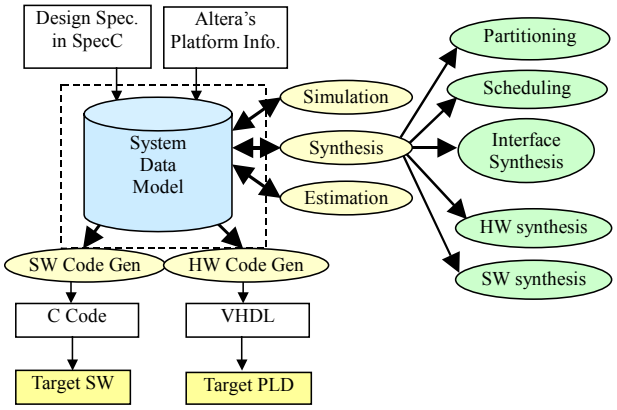

FIGURE 1. PROPOSED PILOT DESIGN FRAMEWORK.

An SDM-centralized design framework of our platform-based HW/SW synthesis system is illustrated in Figure 1. The input of the Pilot is a system level design specification and a platform description, both of which are converted into the internal representation. Building upon SDM, we have several tools to do HW/SW co-simulation, performance estimation and co-synthesis. After the partitioning, scheduling and HW/SW synthesis, the C code and VHDL code will be generated for the software and hardware parts, respectively.

3. SYSTEM-LEVEL DATA MODEL

In this section, we discuss SDM, a unified internal representation of the Pilot system. A unified internal representation can cover the whole life cycle of the flow and support inter-operatability of CAD tools, thus simplifying the design flow. SDM consists of two parts:

- A mathematical model (i.e., MoC) which abstracts the design and captures the essential properties of the system.
- Supplementary information repository which stores the platform information and input language-specific features.

3.1 Requirements for Model of Computation

We believe that taking a systematic, engineering-oriented, tool-based approach instead of an ad hoc one is essential for coping with the increasing complexity of designing current HW/SW systems. One key of a sound system-level design methodology is to adopt an MoC [11], because it provides proper abstraction of a system, and concretely captures the properties to be analyzed and verified.

A good MoC should be able to support hierarchy, abstraction, timing, parallelism and non-determinism. Support of hierarchy is crucial for reducing the complexity because it can simplify the specification process by enabling top-down or bottom-up specification. Representing concurrency is indispensable as modern embedded systems typically have many parts working in parallel. Handling timing constraints is important to the design of an embedded system especially the real-time system, while non-determinism is used to model unspecified or unknown behaviors and to avoid over-restriction on the design implementation.

A good MoC should be executable, synthesizable, verifiable, and unbiased towards any specific implementation. Otherwise, the simulation, synthesis, validation and verification of the design could not proceed.

In past decades, a variety of MoCs have been proposed, such as CFSM [12] (co-design finite state machine), Statecharts [13], SDF [15], Colored Petri Net [16] etc. However, due to the heterogeneous nature of the current embedded system, designers have to employ a mixture of different MoCs and modeling languages to depict the system functions with different characteristics, which makes the design inefficient and error-prone. Integrating different MoCs into a single system is becoming one of the major sources of complexity in a heterogeneous system design environment. The difficulty would be significantly reduced if we could have a versatile MoC, able to represent several other MoCs. Different types of design specification could be transformed into the unified internal representation.

3.2 FunState MoC

After careful evaluation of several candidates, we chose FunState [5] as the core MoC of SDM because it satisfies all the requirements and is capable of representing several well-known computing paradigms including CFSM, SDF, Colored Petri Net, SPI [14], etc.

FunState is a mixed data/control flow MoC. The data flow part is a function network and the control flow part is a finite-state machine. The data flow part consists of a set of function units (functions and embedded components) that perform intensive computation on incoming streams of tokens. A set of storage units (queues and registers) are used to store the tokens. Functions and storages are connected by directed arcs. In the control part, a finite state machine controls the behaviors of the function units dictating when and how they are to be executed.

The precise definitions of the key constituent of FunState are given as follows:

- *Components:* The basic FunState component consists of a network N and a finite state machine M. The network $N = (F, S, E)$ contains a set of storage units $s \in S$, a set of functions $f \in F$, and a set of directed edges $e \in E$, where $E \subseteq (F \times S) \cup (S \times F)$.
- *Storage units:* Queues have FIFO behavior and unbounded length. Notation $q\#$ represents the number of tokens in the queue, and $q\$1, q\$2, \ldots, q\$n$ denotes the values of the tokens. Registers are linear arrays of values of limited length n.
- *Functions:* The function objects $f \in F$ are uniquely named. They operate on tokens when fired. Every function is either in idle or running state.
- *Finite State Machine*: State machine controls the activation of embedded components and functions. A synchronous/reactive model is used. Transitions are labeled with conditions and actions. Conditions are predicates on storage units. The action consists of a set of names of function objects.

FunState can efficiently represent concurrency and hierarchy. Functions can be hierarchical and are called embedded components. States becomes hierarchical when sub-automata are embedded. FunState can also be extended to a timed model to describe the timing properties and constraints of the system. Each of the functions can be associated with a physical latency attribute. A timing constraint of a path in the function network can be specified.

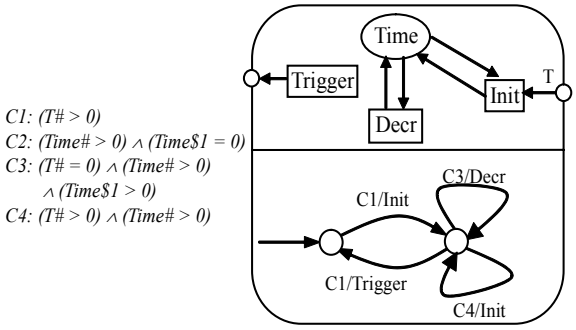

FIGURE 2. A TIMER REPRESENTED IN FUNSTATE

Figure 2 illustrates a timer example represented by a FunState component. The timer periodically decreases by a particular amount of time *T*. Each time it reaches 0, a signal is sent out by a function named *Trigger*. Figure 3 shows the SDF representation and the equivalent FunState representation of an example.

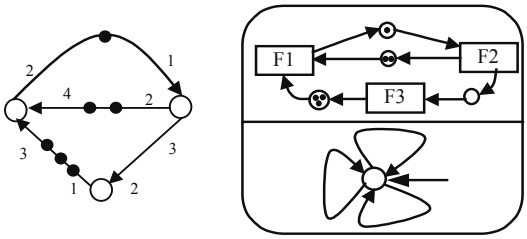

FIGURE 3. SDF REPRESENTATION IN FUNSTATE.

4. INPUT LANGUAGE

We prefer a language rather than some visual diagrams to be the input of our Pilot system. Although both of them can be used to model the embedded system, designers feel more comfortable with programming languages than diagrams. Furthermore, using a language as input easily allows many current existing systems written in textual languages to move toward SoC implementation.

4.1 Requirements for SLDL

A system level design description language (SLDL) is needed to provide a uniform functional specification of the SoC system. The most popular languages, such as C/C++/Java are strong for software development, but lack hardware design concepts. A good SLDL should support the following essential concepts: concurrency, timing, state transition, synchronization, exception handling, software program construct and behavioral hierarchy, etc. Table 1 compares several well-known SLDLs against the above requirements. As the table shows, only SystemC [21] and SpecC [1] satisfy all the requirements. SystemC provides a library of C++ classes to model the hardware. It can used to effectively model and simulate a system. However, it has been envisioned [2] that synthesis tools for SystemC may have difficulties in understanding the information in the libraries, where it is hard to differentiate the code for simulation and the code for specification.

4.2 SpecC

Instead of taking the library-based approach, SpecC extends C language with a set of new constructs that are needed by the SoC design specification. This ensures synthesis tools a easier task of analyzing the specification because all extended constructs are well-defined and the library for simulation is hidden. Since synthesis is our focus, we chose SpecC as the input language of Pilot.

TABLE 1. SLDL SUPPORT FOR ESSENTIAL REQUIREMENTS.

	VHDL	Hard-wareC	State-charts	C/C++/ Java	SystemC	SpecC
Concurrency	●	●	●	○/○/○	●	●
Timing	●	◉	◉	○/○/○	●	●
State Transition	◉	○	●	○/○/○	●	●
Synchronization	●	●	●	○/○/◉	●	●
Exception handling	●	○	○	◉/●/●	●	●
Program Construct	●	○	○	●/●/●	●	●
Behavioral Hierarchy	○	○	●	○/○/○	●	●

● fully supported ◉ partially supported ○ not supported

Syntactically, SpecC is a superset of ANSI-C. Semantically, SpecC explicitly separates communication from computation. The behavior encapsulates the computations and the channel encapsulates the communication. A hierarchical network of behaviors interconnected by hierarchical channels specifies the functionality of a SpecC design.

As mentioned in Section 2, a SpecC to SDM converter is developed to map SpecC input to our internal representation. It intends to transform SpecC behavioral hierarchy to FunState representation and attaches all the language-specific features (e.g., composite data types, variable's scope, etc.) as the supplementary information to SDM. We transform the SpecC into SDM by applying a set of mapping rules. These rules are listed in Table 2.

Note that we do not strictly restrict our input to be SpecC, as the Pilot system has the capability of taking other SLDLs. SDM itself is input-language-independent and the only overhead is that different converters might be needed.

TABLE 2. SPECC TO SDM MAPPING RULES.

Constructs in SpecC	Constructs in SDM
Port	Component Interface
Variable	Register
Channel	Queue
Behavior	Component
Method of Behavior	Sub-Component
Par Statement	Concurrent State Machine
FSM Statement	Conventional Finite State Machine

5. CANDIDATE PLATFORM

Several requirements for a suitable platform are described as follows:

- *Integrity*: The platform should contain a set of dedicated and optimized components, such as microprocessor, bus, memory,

and I/O sub-systems, thereby constituting a total usable system.

- *Usability*: A set of basic tools should be available to support the hardware platform, so that the design can be accelerated and optimized. For example, an optimized compiler for the target microprocessor is very important.
- *Flexibility*: The platform should be extendible and reducible to fit a specific application and satisfy a certain range of performance/cost trade-off.

There exist several well-developed FPSoC platforms in industry. Xilinx' Virtex-II Pro device [19] contains up to four embedded 300 MHz PowerPC processors, serial I/O transceivers and an FPGA fabric, all of which are connected by IBM's CoreConnect on-chip bus. While Virtex-II, the predecessor of Virtex-II Pro, only contains a soft processor. This FPSoC solution comes with Wind River System's embedded design tools.

Quicklogic's QuickMIPS platform [20] contains a 175MHz MIPS4Kc processor, a PCI interface, a memory interface, a 110 Base T Ethernet MAC, an AMBA bus and up to 500,000 system gates of programmable logic. Its system Development Kit comes with drivers for Ethernet, PCI, MMC, Timer and UART etc. It also contains SDK of WindRivers VxWorks®, BlueCat Linux® RTOS and debugger.

Altera's ARM-based Excalibur devices [18] embed an ARM922T hard processor core along with memories and peripherals into a programmable logic device (PLD). The devices offer over 200 MIPS performance and integrated on-board memory, external memory interfaces, standard peripherals and interfaces to the logic portion of the device. SOPC™ builder for system integration and C/C++ compiler/debugger is also provided.

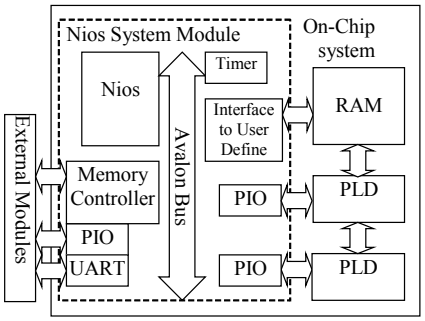

FIGURE 4. NIOS EMBEDDED SYSTEM

Altera's Nios embedded system [18] is based on a flexible and configurable Nios soft processor, as shown in Figure 4. With multi-master Avalon™ bus applied in a Nios-based embedded system, a number of peripherals are available including UART, PIO, DMA, internal memory and external memory interfaces, etc. Furthermore, users can customize their own peripherals. Even the instruction set of the Nios processors is customizable to some extent.

Nios embedded processor occupies only a small part of the programmable device (37% of the APEX20K200, a medium size FPGA). Hence, integrating customized logics and multiple processors on one chip is possible. Altera also provides Nios compiler and SOPC™ tools to accelerate the design cycle.

The design methodology of the Pilot system fits for multiple platforms. We chose Altera's Nios embedded system as the candidate platform to evaluate our approach. A Nios development board is used as our experimental environment. It is populated by an APEX™ 20KE device, a configuration controller, and off-chip SRAMS for storing software programs [18].

6. HARDWARE/SOFTWARE GENERATION

6.1 Hardware Generation

In our Pilot system, hardware generation refines the hardware part of SDM into synthesizable HDL, and then maps the resulting micro-architecture specification into the real platform — Nios embedded system.

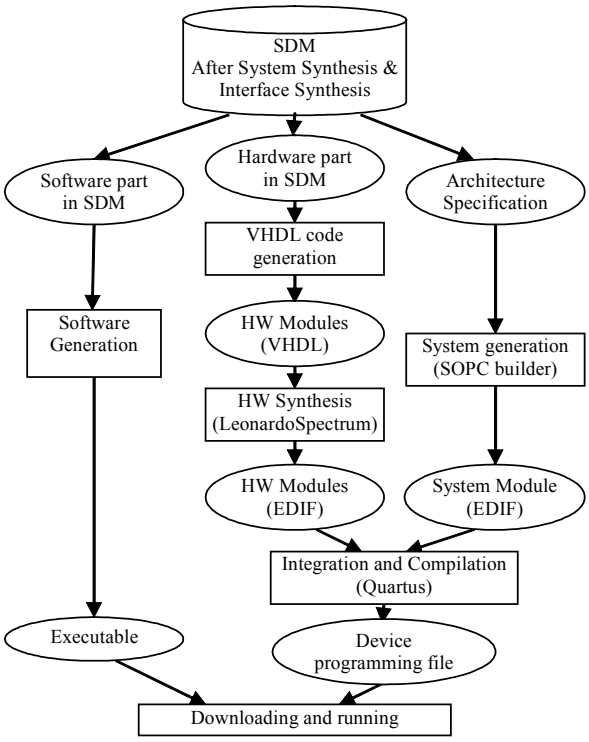

FIGURE 5. SW/HW GENERATION FLOW IN PILOT.

The middle branch of Figure 5 shows our hardware generation flow, which includes VHDL code generation, hardware synthesis, system generation, system integration and compilation. After system-level synthesis, hardware and software parts have been partitioned and specified in SDM. VHDL code is then generated from the hardware part and fed into a synthesis tool, and the netlists are generated in EDIF format for target device. These hardware netlists will be integrated into the whole design as a peripheral module which interacts with the system module through communication modules.

6.2 Software Generation

The left branch of Figure 5 shows the software generation flow. Similar to the SpecC to SDM converter, the C code generator also applies a set of rules that map SDM constructs to C constructs. As a result, every port is mapped into a parameter, every memory becomes a local variable, and every child component instantiation

becomes a statement of function call, etc. These rules are listed in Table 3. The generated C code will be compiled by Nios GCC compiler and executed on Nios processor.

TABLE 3. SDM TO C MAPPING RULES.

Constructs in SDM	Constructs in SDM
Interface	Parameter
Register	Variable
Queue	Composite Structure
Component	Function
Component Instantiation	Function Call
Finite State Machine	Switch…Case Statement

6.3 Interface Generation

The platform supports different schemes of communication and interfacing between software and customized hardware. This flexibility enables designers to explore the design space to obtain an optimized interfacing mechanism and achieve the best performance/cost ratio. In this section, we will compare three possible hardware/software communication mechanisms in the Nios embedded system. They are message passing, buffering, and shared memory, all of which are applied in our case study as discussed in section 7.

6.3.1 Message Passing

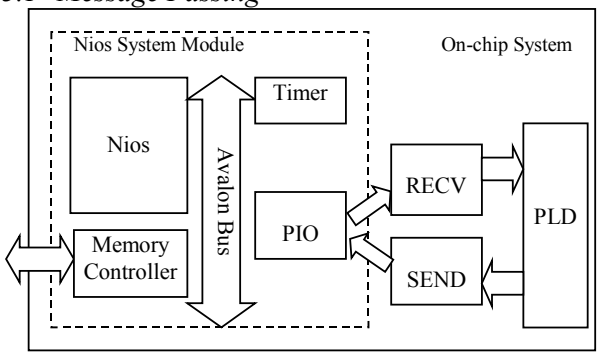

FIGURE 6. MESSAGE PASSING.

Message passing supports a handshaking protocol for communicating peers. In this communication scheme, the data is transferred through messages, and several messages may be combined to form one data package. Handshaking modules should be inserted to guarantee the correctness of package assembling and disassembling. In our current implementation as shown in Figure 6, a RECV module is inserted to assemble data frames into one package for the PLD module input. A SEND module will disassemble a large data package into frames so that PIO can receive them one-by-one. The RECV and SEND modules are FSMs to communicate with PIO in a handshaking manner. The PLD starts running after the RECV has finished, and the SEND starts running after the PLD has given a "DONE" signal.

This communication scheme requires two extra FSM modules in the system, and communication routines should also be inserted into the software part. However, the interface of the PLD does not need to be modified to fit the communication.

6.3.2 Buffering and Shared Memory

Buffering and shared memory are two communication schemes, both of which extend the bus inside the system module to access the on-chip memory. The on-chip memory is also directly accessible by the PLD modules. Figure 7 is the illustration of the architecture for these communication schemes.

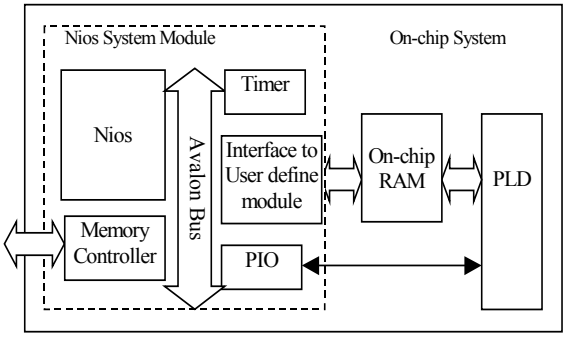

FIGURE 7. BUFFERING AND SHARED MEMORY.

When PLD wants to get data in a buffering communication mechanism, data will be transferred from the software's data memory to this on-chip memory first. PLD then reads the data from the buffer. The communication routine handles the transfer from data memory to buffer, and the software program need not be further modified.

The shared memory mechanism requires the designers to modify the software program and redirect the memory address storing the data to the address of the on-chip memory. The communicating data will be written and read directly into the shared on-chip memory, either by software or by PLD.

Both of these methods require another mechanism to control the exclusive access of the on-chip memory. For example, in order to synchronize the memory access, we can use PIO to send *RESET* and *START* signals to the PLD module, and to receive a *DONE* signal from the PLD module.

Buffering and shared memory simplify the communication controller between the hardware and software parts. In most cases, however, in order to correctly access the shared memory, the interface of the hardware module should be modified.

6.4 System Integration

The right branch of Figure 5 is essentially derived from the interface synthesis, which considers the performance/cost tradeoff of the whole system and selects the most suitable communication mechanism for HW/SW interaction.

The architecture specification describes the mechanism of the hardware/software communication and interfacing method. For example, a shared memory and parallel I/O port support different communication mechanisms and have a great impact in terms of both performance and cost on the resulting system. This will be elaborated in the next subsection.

This interface generation and system integration step first takes architecture specification as input, then generates the system module including CPU, bus, memory controller, and the specified peripheral modules. Interfacing modules selected by interface

synthesis are then generated and inserted between the hardware modules and the system module.

The entire design will be integrated and compiled for the target FPGA device, and the corresponding device programming file will be generated as the result of this step.

In the hardware and interface generation steps, simulation and verification are also applied to ensure correctness of the design.

7. CURRENT STATUS OF PILOT SYSTEM

Pilot is still under-construction and is far from fully completed. However, a number of infrastructural tools have been developed in Pilot, such as:

- *Co-simulation*: A simulator based on FunState MoC is developed for validation and verification of the design. It supports different levels of design; therefore, early functional verification can be performed.
- *HW/SW code generation*: A SDM to C source code generator has been developed for the final implementation of the software components. A hardware code generator is developed to generate VHDL code for the identified HW components.
- *Partitioning and scheduling*: Currently, partitioning and scheduling are done manually. However, since SDM has been already built up, corresponding system level synthesis algorithms are expected to be released in the near future.

8. CASE STUDY – JPEG ENCODER

8.1 Introduction of JPEG Encoder

JPEG is a standard for image compression [6], either for full-color images or gray-scale images. We take the discrete cosine transform (DCT) based JPEG standard, which is the simplest and most commonly used JPEG mode, as our example. We use the existing SpecC specification [3][4] provided by Prof. D. Gajski's research group at UCI. It is purely a software specification without any unnecessary hardware implementation details.

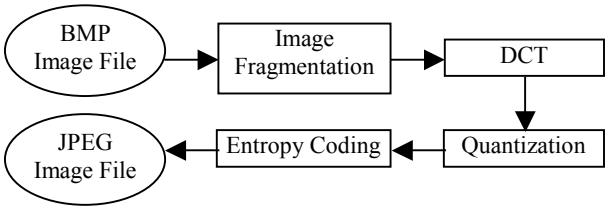

FIGURE 8. JPEG SYSTEM FLOW.

Based on the sequential DCT-based mode, the JPEG encoder is divided into four functional blocks: the image fragmentation block, the DCT block, the quantization block and the entropy coding block, as shown in Figure 8.

In the image fragmentation, a BMP image is divided into the non-overlapping data blocks, each containing an 8x8 matrix of pixels. DCT transforms each data block into a frequency representation. There are two commonly-used DCT algorithms for this translation process: the standard DCT and the ChenDCT [7]. ChenDCT algorithm is used in our design. The quantization function quantizes the DCT output coefficient. For DCT output, the entropy coding module encodes AC coefficients using a predictive coder and encodes DC coefficients using a run-length coder. Finally, the Huffman coding algorithm is applied to generate a JPEG image.

8.2 JPEG Internal Representation in SDM

The dataflow model and non-deterministic FSM of FunState support the specification of non-determinism. Synthesis algorithm has the potential and flexibility to refine the specification and generate a fixed control flow solution. Here we use a fixed control flow specification and directly translate it into SDM.

The fixed control flow specification representation in FunState is introduced below. In a fixed control flow specification, the relative execution order of different components of the system is fixed. Every behavior in the specification is represented as a component in FunState. The control flow is represented in the control part of FunState. The original hierarchy of the specification is maintained. Channels are represented as queues and variables are represented as memories. The ports of each behavior are represented as ports of a component. Figure 9 illustrates the JPEG example represented in SDM.

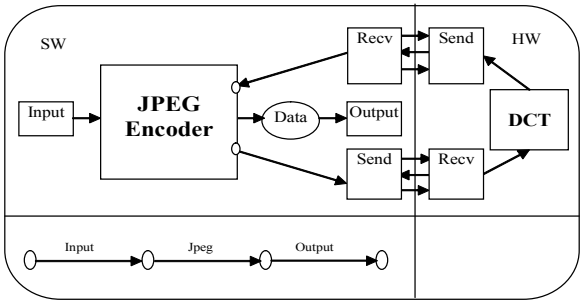

FIGURE 9. JPEG REPRESENTATION IN SDM.

The upper part of the model in Figure 9 represents data flow. The lower part represents FSM. The rectangle in the upper part represents function, the oval between functions represents register, and the circle between functions represents queue. The FSM has an entry state and an end state. The arc between these states indicates that under certain conditions, the function in the upper part is activated.

8.3 Experimental Results

In this section, we will discuss the experimental results of our Pilot flow to implement JPEG to the Nios embedded system. Results from profiling, synthesis, simulation and system integration are given. In addition, possible interesting issues from the result analysis are discussed.

8.3.1 Profiling Results

In profiling, we run the JPEG program both in a PC with Pentium III 650MHz CPU and in Nios embedded system. It can be seen from both of the results presented in Table 4 that the DCT module consumes most of the JPEG running time. In our Pilot flow, the profiling results are used to guide the HW/SW partitioning.

The latency of the Nios embedded system is much longer, since it has a much lower frequency (33 MHz vs. 650 MHz of PII). However, embedded systems have the advantages on low power, small volume, and the capabilities of ubiquitous computation [17].

TABLE 4. PROFILING RESULTS.

Module Name	PC(PIII 650MHz)		NIOS(SW) (33MHz)	
	Time (10^{-6}s)	Ratio (%)	Time (10^{-6}s)	Ratio (%)
Handle Data	2.56	1.72%	50.31	1.22%
DCT	115.48	77.47%	3160.56	76.46%
Quantization	7.22	4.84%	176.42	4.27%
Huffman Encode	23.80	15.97%	746.29	18.05%
Total	149.06	100.00%	4133.57	100.00%

8.3.2 Hardware Cost and Performance

According to the profiling results, we chose the DCT module to be implemented in hardware implementation. Table 5 lists the results from synthesis and simulation tools. SEND and RECV are used as the communication module in the message passing scheme. Half-DCT is a part of the DCT module to loop through the columns of the input matrix (the other parts are a loop through the rows and post-processing). Since the entire DCT in a message passing scheme costs a large number of logic resources, we extract half of it to fit a real device.

The resource cost numbers (IOs and LCs) and frequency are from LeonardoSpectrum for Altera Version v2002a.14. The target device: APEX20KE. Cycle numbers are from the simulation by ModelSim5.5. The buffering and shared memory schemes have the same hardware implementation, and cost much less resources than the message passing scheme. While the latency is longer.

TABLE 5. SYNTHESIS AND SIMULATION RESULTS OF THE GENERATED VHDL CODE.

	SEND	RECV	Half-DCT (1)	DCT (1)	DCT (2)
IOs	916	591	1476	1476	64
Logic Cells	621	686	3461	6253	2100
Frequency (MHz)	98.6	179.8	42.6	45.6	38.9
Cycle	128	128	571	1809	3988
Latency (10^{-6}s)	1.30	0.712	13.4	39.7	102

(1) Message passing; (2) Buffering and Shared memory

Table 6 is a resource cost comparison among three JPEG encoder implementations. The first design only loads half of the DCT into the device and applies message passing between Half-DCT and the Nios system module. The second one applies a buffer between DCT and Nios. The third one applies a shared memory communication scheme. These results are from Quartus II v2.0. For comparison, the resource cost of the Nios system module and the capacity of the target device are also given.

TABLE 6. COST COMPARISON OF THREE IMPLEMENTATIONS.

	Half-DCT Message Passing	DCT Buffering	DCT Shared Memory	Nios System Module	Device EP20K200 EFC484-2X
Total logic elements	7439 (89%)	4555 (54%)	4726 (56%)	2766 (33%)	8320
Total ESB bits	26496 (24%)	27840 (26%)	27840 (26%)	26496 (24%)	106496

Since DCT with a buffering and shared memory mechanism only needs several ports to access memory, the port size is much smaller than that of the message-passing DCT. This reduces the synthesis complexity dramatically and the synthesis result of buffering and shared memory DCT is much better than the message passing one (Table 5). Although in the entire design, buffering or shared memory will cost a few more embedded system block (ESB) resources (Table 6), the extra cost is trivial. Buffering and shared memory schemes enforce memory access, which introduces extra cycles compared with the message-passing scheme.

8.3.3 JPEG Designs' Performance

Table 7 lists the run time results of every module in the resulting designs. These numbers are measured by Nios SDK 2.0 and in the unit of times/second. The execution cycle of every module begins when the data from the previous module is ready, and ends when the resulting data is produced.

Although the DCT function runs in the fewest cycles in the message passing scheme, the communication cost overwhelms the gain realized by the fewer running cycles. The dominating communication cost is the run time of software communication routines handling the receiving and sending operations for message passing. Moreover, the other half of the DCT is executed by software, thus the entire DCT does not show great improvement in performance. In a buffering scheme, the communication overhead is much lighter since all of the communication operations are data transfers between the buffer and data memory of the CPU.

Shared memory views the data as the global variable accessible by the modules connected with it. The programs in the software portion (in our scene, they are *HandleData* and *Quantization*) are not required to transfer data. As long as the previous software program ends, the output can be directly accessed from the HW module. The output of the HW module is not transferred either, thus avoiding the communication cost. This results in the highest throughput of the DCT module.

As we observe the overall performance of the JPEG Encoder design, the highest one is the shared memory scheme with speedup of 3.2. The optimized speedup is 3.33 by eliminating the run time of the DCT.

TABLE 7. RUN TIME OF EVERY MODULE OF JPEG DESIGN IN DIFFERENT IMPLEMENTATIONS (TIMES/SEC).

	Pure SW	Half-DCT Message Passing	Buffering	Shared memory
HandleData	21422.59	21422.59	21422.59	21569.24
DCT	194.82	*293.51	2117.89	8156.11
Quantization	3229.26	3255.96	3477.7	3267.38
HuffmanEncode	1006.88	1006.88	1093.8	1006.88
Overall	0.75	0.99	2.09	2.40
Overall Speedup	1.00	1.32	2.79	3.20

*The half hardware DCT runs 2115.26 times/second.

9. CONCLUSIONS

The paper has presented a new platform-based synthesis framework. A unified SDM is developed to capture the entire design and is the basis to build other system level synthesis tools. A JPEG encoder is used as an example to evaluate and validate the design flow. Experimental results show the impact on the area and performance of the final designs by applying HW/SW partitioning and different interfacing schemes. Future works will focus on developing synthesis algorithms that aim to maximize the overall system performance under the platform resource constraints.

ACKNOWLEDGMENT

The authors wish to thank Prof. D. Gajski's research group at UCI for providing us with the SpecC compiler source code and related documents. We are also grateful to Altera Corporation for providing us with the experimental devices.

REFERENCES

[1] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao, *"SpecC: Specification Language and Methodology,"* Kluwer Academic Publishers, Boston, March 2000.

[2] J. Zhu and D. Gajski, *"Compiling SpecC for simulation,"* Proceedings of the conference on Asia South Pacific Design Automation Conference, Yokohama, Japan, 2001.

[3] H. Yin, H. Du, T. C. Lee and D. Gajski, *"Design of a JPEG Encoder using SpecC Methodology,"* Technical Report ICS-00-23, Department of Information and Computer Science, University of California, Irvine, July 2000.

[4] L. Cai, J. Peng, C. Chang, A. Gerstlauer, H. Li, A. Selka, C. Siska, L. Sun, S. Zhao and D. Gajski, *"Design of a JPEG Encoding System,"* Technical Report ICS-99-54, Department of Information and Computer Science, University of California, Irvine, Nov. 1999.

[5] K. Strehl, L. Thiele, M. Gries, D. Ziegenbein, R. Ernst, and J. Teich, *"FunState – An Internal Design Representation for Codesign,"* IEEE Trans. VLSI Systems, vol. 9, pp. 524 – 544.

[6] V. Bhasharan and K. Konstantinides, *"Image and Video Compression Standards,"* Second Editioin, Kluwer Academic Pubishers, 1997.

[7] W. H. Chen, C. Smith and S. Fralick, *"A Fast Computational Algorithm for the Discrete Cosine Transform,"* IEEE Trans. Communications, vol. 25(9), pp. 1004-1009, Sept. 1977.

[8] K. Keutzer, S. Malik, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli, *"System Level Design: Orthogonalization of Concerns and Platform-Based Design,"* IEEE Transactions on Computer-Aided Design, Vol. 19(12), December 2000.

[9] Mescal Web Site, http://www.gigascale.org/mescal.

[10] Metropolis Web Site, http://www.gigascale.org/metropolis.

[11] M. Sgroi, L. Lavagno and A. Sangiovanni-Vincentelli, *"Formal Models for Embedded System Design,"* IEEE Design & Test of Computers, vol. 17(2), pp. 14-27, June 2000.

[12] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki and B. Tabbara, "*Hardware-Software Co-Design of Embedded Systems: The POLIS Approach,*" Kluwer Academic Publishers, 1997.

[13] D. Harel, *"Statecharts: A visual formalism for complex systems,"* Science of Computer Programming, vol. 8(3), pp. 231-274, 1987.

[14] D. Ziegenbein, K. Richter, R. Ernst, L. Thiele and J. Teich, *"SPI - A System Model for Heterogeneously Specified Embedded Systems,"* Technical Report TR-SPI-00-04, Institut für Datentechnik und Kommunikationsnetze, TU Braunschweig, 2000.

[15] E.A. Lee and D.G. Messerschmitt, *"Synchronous dataflow,"* Proceedings of the IEEE, vol. 75(9), pp. 1235-1245, September, 1987.

[16] K. Jensen, *"Colored Petri nets: A high level language for system design and analysis,"* Advances in Petri nets 1990, G. Rozenberg(ed), Lecture Notes in Computer Science, Springer, LNCS 483, 1990.

[17] D. Tennenhouse, *"Proactive computing,"* Communications of the ACM, vol.43, (no.5), pp. 43-50, May, 2000.

[18] Altera Web Site, http://www.altera.com/.

[19] Xilinx Web Site, http://www.xilinx.com/.

[20] Quicklogic Web Site, http://www.quicklogic.com/.

[21] SystemC Web Site, http://www.systemc.org/.