Debugging HW/SW Interface for MPSoC: Video Encoder System Design Case Study

Mohamed-Wassim Youssef, Sungjoo Yoo, Arif Sasongko, Yanick Paviot, Ahmed A. Jerraya
System Level Synthesis Group, TIMA Laboratory, Grenoble, France
{wassim.youssef, sungjoo.yoo, arif.sasongko, yanick.paviot, ahmed.jerraya}@imag.fr

ABSTRACT

This paper reports a case study of multiprocessor SoC (MPSoC) design of a complex video encoder, namely OpenDivX. OpenDivX is a popular version of MPEG4. It requires massive computation resources and deals with complex data structures to represent video streams. In this study, the initial specification is given in sequential C code that had to be parallelized to be executed on four different processors. High level programming model, namely Message Passing Interface (MPI) was used to enable inter-task communication among parallelized C code. A four processor hardware prototyping platform was used to debug the parallelized software before final SoC hardware is ready. The targeting of abstract parallel code using MPI to the multiprocessor architecture required the design of an additional hardware-dependent software layer to refine the abstract programming model. The design was made by a team work of three types of designer: application software, hardware-dependent software and hardware platform designers. The collaboration was necessary to master the whole flow from the specification to the platform.

The study showed that HW/SW interface debug was the most time-consuming step. This is identified as a potential killer for application-specific MPSoC design. To further investigate the ways to accelerate the HW/SW interface debug, we analyzed bugs found in the case study and the available debug environments. Finally, we address a debug strategy that exploits efficiently existing debug environments to reduce the time for HW/SW interface debug.

Categories and Subject Descriptors

B.1 Control structures and microprogramming, D.1.3 Concurrent programming, D.2.5 Testing and debugging

General Terms

Design, experimentation, verification

Keywords

Multiprocessor system-on-chip, Hardware-software interface, Hardware-dependant software, Debug

1. INTRODUCTION

For many people main difficulties when designing multiprocessor system on chip (MPSoC) are the parallelization of sequential code and mapping of functions on the multiprocessor architecture. This is partially true and explains recent studies such as parallel programming models [1], the exploration of multiprocessor SoC architectures including network-on-chips [2], mapping parallel application software on the architecture [3][4], and refinement of parallel programming models [5][6].

The truth is that MPSoC design also includes an integration step where distributed software (SW) needs to be adapted to hardware (HW) [7][8]. This includes the development of hardware-dependent software layer to adapt application SW to the HW architecture. During the hardware-dependent software design, the debug of HW/SW interface may be extremely difficult and time consuming. This restricts design space exploration (in terms of HW architecture and mapping) and may even induce missing the time-to-market constraint.

There have been few practical reports on the difficulties of debugging the HW/SW interface and on how to accelerate the HW/SW interface debug. In this paper, we study and report the problems of HW/SW interface debug. This study is a first step to understand the debug problems that SoC designers are facing and to identify necessary design methods to overcome the problems.

To investigate the debug problem, we performed a case study of complex video encoder, namely OpenDivX [9], which is a popular version of MPEG4, on a multiprocessor architecture with four processors. In this study, the initial specification is given in sequential C code that had to be parallelized into concurrent tasks to be executed on the four different processors. High level programming model, namely Message Passing Interface (MPI) [10] was used for the communication between concurrent tasks. The targeting of abstract parallel code using MPI to the HW architecture required the design of an additional hardware-dependent software layer to refine the abstract programming model. A four processor prototyping platform [11] was used to debug the refinement of the abstract programming model, i.e. HW/SW interface.

The rest of the paper presents the case study. Section 2 explains the MPSoC design flow that we used in this case study. Section 3 presents the OpenDiVX application. Section 4 gives our experience in designing HW/SW interfaces. Section 5 presents lessons learned from this case study. Section 6 concludes this paper.

2. A Generic MPSoC Design Flow

Figure 1 shows a generic MPSoC design flow. In the figure, ovals represent design steps and rectangles represent the input and output of design step. The design flow consists of parallelization of initial sequential code, application mapping on the multiprocessor HW architecture, HW design, and hardware-dependent software (HdS) design to refine the parallel programming model. Before final HW is ready, a HW prototype is generally built to allow for SW development and the debug of HW/SW interfaces. When both HW and SW are designed, a final HW/SW integration step allows to debug the entire system design.

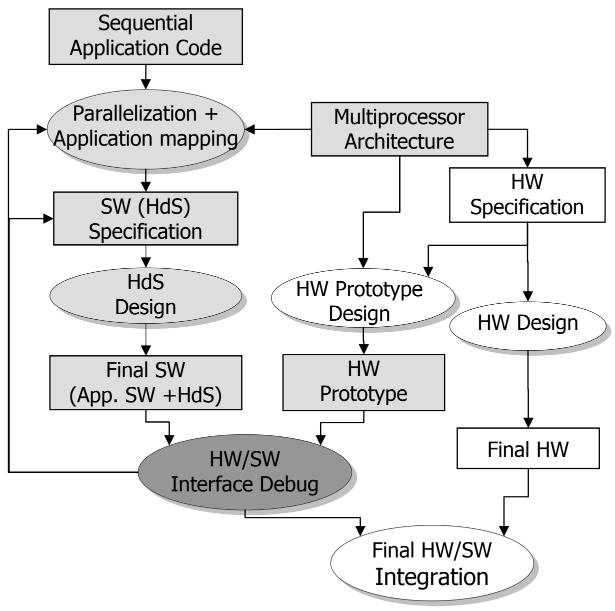


Figure 1. A generic MPSoC design flow.

2.1 Parallelization

To target the sequential application SW code on a multiprocessor architecture, the code needs to be distributed, i.e. parallelized using a parallel programming model on the architecture. In this case study, the parallelization meant task-level parallelization. It is to decompose the sequential code into concurrent tasks communicating with each other via parallel programming primitives.

Parallelization step includes identifying a set of code sections that need to be transformed to concurrent tasks, transformation of identified code sections into concurrent tasks, and establishing inter-task communication between the identified tasks using parallel programming model primitives (e.g. send/recv).

As parallel programming models suited to multiprocessor architectures, there are two types: shared memory model (e.g. OpenMP [12]) and message passing model (e.g. MPI [10]). In our case study, we used MPI as a parallel programming model. However, the above parallelization step is common to both shared memory and message passing models.

The correctness of the parallelized code is proved by comparing both results of execution of sequential and parallelized code using the same testbenches. In the design flow, we used two simulation environments to validate the correctness of parallelized code. One is the existing MPI environment called MPICH [18]. The other is a simulation model of MPI primitives in SystemC [13] (called MPI/SC) that we developed.

2.2 Application Mapping

The designer maps the parallelized code on the given multiprocessor architecture. The mapping includes mapping tasks on processors and mapping inter-task communication on the interconnection network of the multiprocessor architecture. The design space of mapping is huge since the size of design space is exponential with respect to the number of application SW tasks and that of processors in the multiprocessor architecture.

The evaluation of application mapping can be done by high-level estimation techniques, e.g. trace-based simulation [4] or in low-level validation using HW/SW cosimulation or prototyping.

2.3 Refinement of Parallel Programming Model

Targeting the parallelized code on the multiprocessor architecture requires the refinement of parallel programming model on the architecture. The refinement is to design hardware-dependent software that implements the API of parallel programming model. The hardware-dependent software contains also μ-kernel(s) and boot code to enable MPSoC initialisation, task scheduling, interruption and I/O.

The hardware-dependent software is usually designed manually. Commercial system-level design tools such as System Studio [14], ConvergenSC [15], Platform Express [16], etc. help designers' manual design of hardware-dependent software. There are a few research tools that aim to provide automatic methods of hardware-dependent software design [5][6].

Whether the hardware-dependent software is designed manually or automatically, the designer needs to fix all the implementation parameters such as address map, interrupt priorities, stack sizes, etc. as the specification of hardware-dependent software.

2.4 HW/SW Interface Debug

After the hardware-dependent software design, designers need to debug the HW/SW interface. We define the term, *HW/SW interface debug* to be 'both debugging hardware-dependent software itself and debugging the interaction between hardware-dependent software and the HW architecture'.

Several HW prototyping platforms for MPSoC are available [11][17]. In this work, HW/SW interface debug is done in a cycle accurate way using a commercial prototyping platform, ARM AP Integrator [11], which has four ARM processors (two ARM7s and two ARM9s) connected via AMBA. For the debug, first, the platform is configured by setting clock frequencies of processors and AMBA, and debug options. Then, the SW code including the parallelized code and hardware-dependent software are compiled and downloaded on each of processors in the platform. Then, the entire prototype is ready to perform the application, OpenDivX. The prototyping platform supports breakpoints and source level debugging.

The grey area in Figure 1 represents the scope of this case study in the design flow. The key focus of this study is the debug of hardware-dependent software on the HW prototype.

3. Video Encoder Application: OpenDivX

Figure 2 shows a block diagram of OpenDivX encoder [9]. Basically, the encoding is based on removing spatial and temporal redundancy from input video frames. This encoder produces two types of video frame: I and P frame. An I frame represents an *intra* frame which is the frame that contains self sufficient information to be decoded. A P frame stands for a *predicted* frame. The number of P frames per one I frame, which is an important factor for compression ratio and result quality, is an implementation parameter.

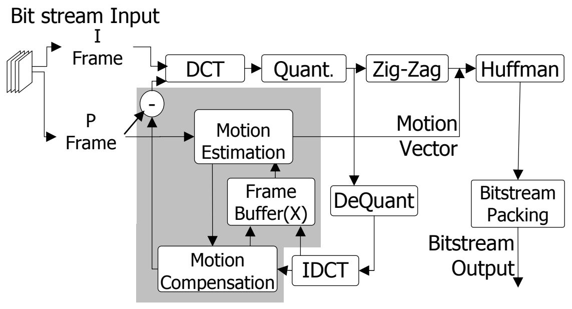


Figure 2. Block Diagram of OpenDivX Application

To produce an I frame, the encoder applies DCT and quantization to the input video frame. After this step, the result goes through zig-zag, Huffman compression, and bit stream packing to produce the output. To prepare the coding of P frame, dequantization and inverse DCT are applied to the result of DCT and quantization to produce the image frame that will be obtained by an OpenDivX decoder. The decoded image frame (X frame) is stored in the frame buffer. It will be used as a reference frame for coding a P frame. Then, the difference between the new input video frame and motion-compensated frame is calculated. The difference is applied to the chain of DCT, quantization, and zig-zag. The result of this operation chain and the motion vector go through Huffman coding and bit stream packing to produce a P frame.

The OpenDivX encoder application was initially written in sequential code as most of existing code of multimedia applications. The entire code size is about 10K lines with 43 files. It needs 1 GHz P3 processor to encode 20 QCIF (176x144 pixels) frames per second. Designing this application in an embedded SoC is a challenge since it requires very high computation power that a single embedded processor such as ARM7 or ARM9 cannot support. When executing the application on a 28 MHz ARM9, though the code size is relatively small, 104KB, it takes 227 seconds to process 20 input frames. Thus, to achieve the real-time performance with the given application SW code, we need optimisations of application SW code, HW acceleration, and possibly massively parallel processor architectures.

4. Design Case Study

We designed the OpenDivX encoder system through the design flow explained in Section 2. This section gives the details of design process. Figure 3 summarizes the key steps:

- Parallelizing the sequential code and validating the parallelized code (Figure 3 (a)—(b))
- Hardware-dependent software design (Figure 3 (c)—(d))
- HW/SW interface debug (Figure 3 (e))

The design was a team work of three types of designer: application software, hardware-dependent software and platform designers. The collaboration was necessary to master the whole flow from the specification to the platform.

4.1 Parallelization

To parallelize the initial sequential code of OpenDivX on the multiprocessor architecture with four processors, we used a profiling of code execution to detect computational bottlenecks. This resulted in the identification of the motion estimation and compensation function (MEC), the shaded area in Figure 2 as a good candidate for parallelization since it consumes about 50% of total processor computation cycle. To exploit four processors in the architecture, we made three tasks (called *slaves*) of MEC function by applying each of three slave tasks to a third of image frame. Then, we made another task called *master* using the other part of OpenDivX application. Figure 3 (a) shows four tasks (one master denoted with 'M' and three slaves 'S1', 'S2', and 'S3').

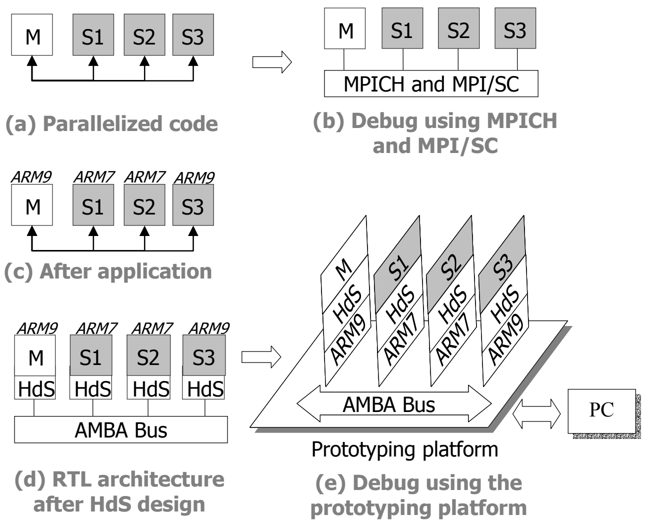


Figure 3. OpenDivX Design and Debug Flow

Table 1 gives the details of data exchanged between master and slave tasks: data structure name, data size and type. In terms of exchanged data size, master sends 149 KB and receives 25 KB per image frame to/from each slave.

Table 1. Data exchanged between master and slave tasks

Structure	Size	Type
Master to Slave		
Curr_comp_y	176x144/3	SInt
Curr_comp_u,v	176x144/4x3*2	SInt
Curr	176x144/3	SInt
Prev	208x176	SInt
Prev_u, v	208x176/4*2	SInt
Tab_int	14	SInt
Tab_float	2	Float
Slave to Master		
Curr_comp_y	176x144/3	SInt
Curr_comp_u,v	176x144/4x3*2	SInt
Mv16_W,H	11x9x4/3*2	Float
Mv8_W,H	11x9x4/3*2	Float
Mode_16	11x9/3	SInt

The parallelized code is then validated using MPICH [18] and MPI/SC (as shown in Figure 3 (b)). The parallelized code is valid, if its execution result is identical to that of sequential code. The parallelization process took a designer 3 man-weeks. In terms of modification of code, we modified and created 1029 lines of code in 10 different source files.

4.2 Application Mapping

We mapped each of the four tasks on one of the four processors (Figure 3 (c)). The master task is mapped on an ARM9 processor and the other three tasks are mapped on the remaining two ARM7 processors and the other ARM9.

4.3 Refinement of Parallel Programming Model

Figure 4 illustrates the refinement flow. Figure 4 (a) exemplifies a part of system specification of OpenDivX in SystemC. In the figure, master task code calls an MPI primitive, MPI_Send() to send data to a slave task. Slave task code calls the corresponding primitive, MPI_Recv() to receive the data. The refinement of parallel programming model is to implement these primitives on the multiprocessor architecture.

In the case study, we used only the MPI primitives for point-to-point communication (MPI_Send/Recv). For the point-to-point MPI communication, two (control and data) FIFOs are managed. When MPI_Send is called, it puts the data to be sent into the data FIFO, updates a control information (e.g. data FIFO count) in the control FIFO and sends a message (request-to-send) to the receiver. The access to the control FIFO is serialized via a semaphore. At the receiver side, when MPI_Recv is called, first, the control FIFO at the sender side is checked to see if there are available data. If so, the data is fetched. If not, the receiver waits on the semaphore (later waken up via interrupt).

To implement the MPI primitives, the designer describes the specification of hardware-dependent software by fixing the parameters of implementation details. Figure 4 (b) shows examples of such parameters. As shown in the figure, the parameters describe the implementation details of hardware-dependent software such as the addresses and sizes of FIFO's and the details of μ-kernel such as interrupt level, address of interrupt control register, etc.

In our case study, we used a free software tool called ROSES [5] to design hardware-dependent software. It takes as the input a SystemC specification (shown in Figure 4 (a)) including the implementation parameters. It generates hardware-dependent software on multiprocessor architectures. The generation is based on reusing a SW code library of hardware-dependent software.

The grey area in Figure 4 (c) corresponds to the hardware-dependent software generated by the tool. As shown in the figure, the hardware-dependent software represents three types of SW code: MPI code, μ-kernel, and multiprocessor boot.

Table 2 shows the code size of generated hardware-dependent software in terms of executable code size (KB) and source code size (lines of code, LOC).

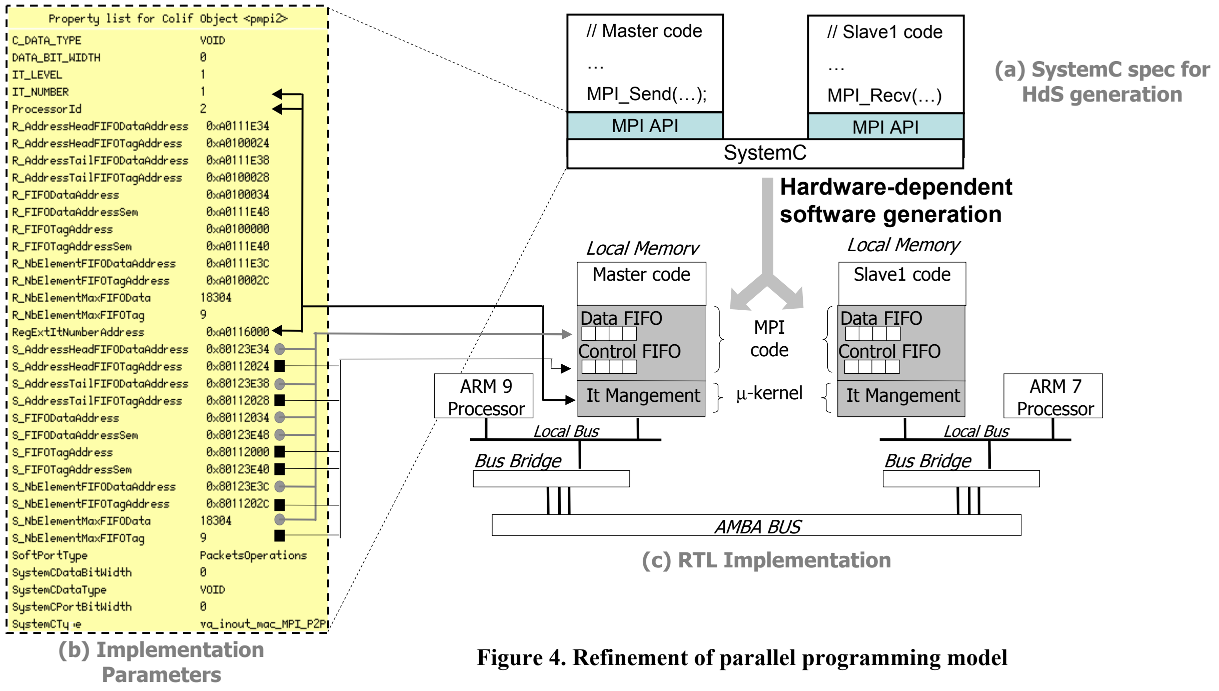
To write the input specification (in SystemC) of the design tool, we wrote 30 files (729 lines) and 553 implementation parameters. The major difficulty of this step consists in the huge number of parameters required to be fixed. Such many parameters issue an erroneous interpretation of some parameters, which needs to be debugged in the HW/SW interface debug.

Table 2. Code size of hardware-dependent software

Code	Master	Slave
MPI	3.4 KB, 923 LOC	2.9 KB, 892 LOC
μ-kernel	2.3 KB, 1069 LOC	2.2 KB, 1065 LOC
MP boot	0.7 KB, 379 LOC	0.7 KB, 379 LOC
Total	6.4 KB, 2371 LOC	5.8 KB, 2336 LOC

4.4 HW/SW Interface Debug

The generated code of hardware-dependent software is compiled and downloaded on the prototyping platform (ARM AP Integrator). We found and fixed many bugs before obtaining the correct code. To investigate the bug sources and to find an efficient strategy to perform HW/SW interface debug, we classified bugs found on the prototyping platform.



Property list for Colif Object <pmpi2>

C_DATA_TYPE	VOID
DATA_BIT_WIDTH	0
IT_LEVEL	1
IT_NUMBER	1
ProcessorId	2
R_AddressHeadFIFODataAddress	0xA0111E34
R_AddressHeadFIFOTagAddress	0xA0100024
R_AddressTailFIFODataAddress	0xA0111E38
R_AddressTailFIFOTagAddress	0xA0100028
R_FIFODataAddress	0xA0100034
R_FIFODataAddressSem	0xA0111E48
R_FIFOTagAddress	0xA0100000
R_FIFOTagAddressSem	0xA0111E40
R_NbElementFIFODataAddress	0xA0111E3C
R_NbElementFIFOTagAddress	0xA010002C
R_NbElementMaxFIFOData	18304
R_NbElementMaxFIFOTag	9
RegExtItNumberAddress	0xA0116000
S_AddressHeadFIFODataAddress	0x80123E34
S_AddressHeadFIFOTagAddress	0x80112024
S_AddressTailFIFODataAddress	0x80123E38
S_AddressTailFIFOTagAddress	0x80112028
S_FIFODataAddress	0x80112034
S_FIFODataAddressSem	0x80123E48
S_FIFOTagAddress	0x80112000
S_FIFOTagAddressSem	0x80123E40
S_NbElementFIFODataAddress	0x80123E3C
S_NbElementFIFOTagAddress	0x8011202C
S_NbElementMaxFIFOData	18304
S_NbElementMaxFIFOTag	9
SoftPortType	PacketsOperations
SystemCDataBitWidth	0
SystemCDataType	VOID
SystemCPortBitWidth	0
SystemCType	va_inout_mac_MPI_P2P

Figure 4. Refinement of parallel programming model

The MPSoC design bug is divided into application SW bug, HW/SW interface bug, and HW architecture bug. Application SW bug is due to the limitation of simulation model on MPICH and MPI/SC used during the development of application SW (including the parallelization). For instance, the bug of stack overflow is not detected in high-level simulation on MPICH and MPI/SC since the stack on the target processor is not modelled in the simulation. Since the encoding algorithm is data dependent, such a bug could be detected with specific video frames. This means that long time execution with additional testbenches was needed to detect all the bugs on the prototyping platform. We classify, into application SW bug, a C library bug due to the lack/mismatch of supported functions (e.g. malloc).

Table 3. Details of bugs found in the low-level debugging

Type	Example bug	%	Bug source
Appl. SW	Data dependent computation	5	Insufficient stack space for appl. SW
	C library bug	12	Lack in existing C library
MP booting	Booting is not synchronized among processors.	12	Wrong platform configuration and initialization
μ-Kernel	Lost some interrupts	13	Lack of nested interrupt in ISR
	Wrong interrupt priority levels	5	Misuse of interrupt levels.
	Context switch does not work correctly.	5	Multitasking in system/IRQ stack
Parallel prog. model	Incorrect FIFO counter value causes deadlock.	13	Non implemented communication scenario
HW interface	Result of compressed video is not correct.	30	Wrong memory map assignment
Design environ ment	Abnormal execution of a portion of C code	5	Different data type handling by 'armcc' and 'gcc'

A HW/SW interface bug may be a hardware-dependent software bug or HW interface bug. Several kinds of hardware-dependent software bug exist. These may be related to multiprocessor booting, μ-kernel, and MPI code (i.e. parallel programming model) bug. A HW interface bug is due to the incorrect configuration and access of/to the HW architecture. It results mostly from designer's misunderstanding of HW architecture. For instance, a wrong configuration of memory map for interrupt control registers belongs to the HW interface bug.

A HW architecture bug is the conventional HW design bug including the bugs found in the design of sub-system (e.g. bus, interrupt controller, DMA controller) and global communication network (e.g. network-on-chip design). In this case study, since the HW architecture (i.e. the prototyping platform) is fixed, the HW architecture bug is not considered.

Table 3 shows some statistics about the bugs found. For each bug type, the table gives an example, the percentage of occurrence of the bug, and an example of the bug source. As shown in the table, 78% of bugs found on the prototyping platform are HW/SW interface bugs. In the table, bugs related with design environment represent mismatches between application SW design tools and MPSoC design tools. For instance, an example of such a bug is the difference of handling some data types between 'gcc' for application SW design and 'armcc' for the target processor.

It took 7 weeks to a team of three designers (application SW, hardware-dependent software and prototyping platform designers) to find all the bugs.

4.5 Design Cycle Analysis

Table 4 shows design cycle needed in each design step and the sizes of code written by the designer or generated by the tool to estimate the required design efforts. The table shows that debug on the prototyping platform took more than 50% of total design cycle in this case study. HW/SW interface debug consumed most of the design time in this case study.

Table 4. Design cycle in the case study

Design step	Design cycle	# of lines of code
Parallelization and validation	3 weeks	1029, manual code
Input specification for HdS generation	2 weeks	729, manual code
HdS generation	5 minutes	9336, generated code
Debug on the prototyping platform	7 weeks	40 bugs fixed

Two points need to be mentioned. First, in this case study, the HW design cycle was not counted since we used a prototyping platform. In the case that the HW design cycle is counted, the debug cycle for HW/SW interface may become more significant than in this case study. In fact, many symptoms are common to both hardware and software bugs. This makes the source of the bug harder to find and the bug fixing cycle longer. As reported in [19], even when the HW design cycle is counted, the debug may still be the most time consuming design step.

The second point to mention is the added value of automatically generating hardware-dependent software. As shown in Table 4, the tool generates 9336 lines of code (=master's 2371 + 3 slaves*2336/slave, from Table 2) for hardware-dependent software in 5 minutes. The generation uses existing components in the hardware-dependent software library. In the case of manual design without using the library components and the tool, the design time is estimated to be 31 weeks. The automatic generation of hardware-dependent software yields a significant reduction in total design cycle.

5. Lessons Learned

The majority of HW/SW interface bugs were caused by the misunderstanding of HW architecture. The designer has to handle too many parameters and to know the every details of HW architecture. This problem will become more significant as HW architectures and application SW become more complex. To ease the problem, tool supports are needed to detect/prune wrong combinations of parameters and to identify parameters that may give higher performance or lower design cost than others.

Using an automatic tool to design hardware-dependent software is also very helpful to reduce design cycle. Manual coding of hardware-dependent software would have caused more bugs in our case as well as longer design cycle.

HW/SW interface debug seems to be the most expensive step in total design cycle of MPSoC. Thus, shortening the debug time is a key to reduce the total design cycle. The prototyping platform that we used in this case study enables fast and accurate debugging since it supports cycle-accurate execution. However, for more complex systems, application-specific prototype platforms may be required. Building such platforms requires a lot of time that may be prohibitive to meet the time-to-market constraint. Thus, to shorten the design time, we need to move the debug task from prototyping platform to other existing debug environments applicable earlier in the design cycle.

A possible strategy would be to move debug tasks from the prototyping platform to debug environments based on HW/SW cosimulation. Many bugs can be found before the prototyping platform is ready. The following two types of cosimulation environments can be used for HW/SW interface debug [20].

- Cycle-approximate HW/SW cosimulation (ISS w/ implicit memory model + transaction level model HW)
- Cycle-accurate HW/SW cosimulation (ISS w/ explicit memory model + transaction level model HW)

Both use instruction set simulators (ISSs). The ISS is needed to debug the hardware-dependent software that contains assembly code specific to the target processor.

Cycle-approximate HW/SW cosimulation consists of ISS having the implicit processor memory model (i.e. memory image server model) and transaction level model (TLM) HW models. It gives a fast simulation (~100Kcycles/sec). However, it is not cycle-accurate since the processor memory is modelled inside of ISS. Thus, the contention to the processor memory (e.g. between DMA controller and processor accesses) is not simulated in a cycle-accurate way. Cycle-accurate HW/SW cosimulation (ISS w/ explicit memory model + TLM) is much slower (~1Kcycles/sec) than the first type since it simulates the processor memory as an external HW model in HW/SW cosimulation. However, it gives cycle-accurate simulation.

To exploit the cycle-approximate HW/SW cosimulation (ISS w/ implicit memory model + TLM HW model), we classify HW/SW interface bug into purely functional and timing-related bugs. The purely functional bugs can be detected by a functional or timed simulation. Examples are C library bugs, basic functionality of hardware-dependent software such as interrupt service routine for timer, etc. The timing-related bugs can be detected only by a specific temporal ordering of events in the system execution. Examples are multiprocessor booting bugs, μ-kernel bugs such as nested interrupt processing bug, parallel programming model bug that requires inter-processor communication, etc. Those bugs can be detected only by cycle-accurate HW/SW cosimulation or emulation. Note that many timing-related bugs can also be detected as purely functional bugs only if the testbench is well developed to enforce the specific order of events necessary to detect the bugs.

To exploit the cycle-accurate HW/SW cosimulation (ISS w/ explicit memory model + TLM HW models), we classify the testbench into two types: short and long execution time testbenches. Short execution time testbench includes testbench for multiprocessor booting, small testbench (which replaces the complex application SW) to debug the HW/SW interface. Long execution time testbench is the system testbench to detect the bugs after long operating time, e.g. bugs appearing after hundreds of image frames.

Although the cycle-accurate HW/SW cosimulation (ISS w/ explicit memory model + TLM HW models) runs slow, it can be exploited to run the short execution time testbenches. They will detect most of bugs of hardware-dependent software, i.e. parallel programming model, μ-kernel, and MP booting, and HW interface bugs related with the operation of hardware-dependent software.

After the usage of both simulation environments, when the prototyping platform is available, it may be used to run the long execution time testbenches.

6. Conclusion

We studied and reported the HW/SW interface debug of an MPSoC design of video encoder system, namely OpenDiVX. This study is a first step to understand the debug problems and to identify necessary design methods to shorten design cycle. This work has shown that the HW/SW interface debug was the most time-consuming step that takes most of design time. This is identified as a potential killer for application-specific MPSoC design. To investigate the bug sources and to find efficient methods to shorten HW/SW interface debug, we presented bug classifications and a strategy of HW/SW interface debug in MPSoC design.

7. ACKNOWLEDGMENTS

This work was supported by ToolIP, Medea+ project and ArchiFlex, French government funded project.

8. REFERENCES

[1] The Ptolemy project, http://ptolemy.eecs.berkeley.edu/
[2] E. Rijpkema, *et al.*, “Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip”, *Proc. DATE,* 2003.
[3] J. Liu and P. Chou, “Energy Optimization of Distributed Embedded Processors by Combined Data Compression and Functional Partitioning”, *Proc. ICCAD,* 2003.
[4] P. Lieverse, *et al.*, “A Methodology for Architecture Exploration of Heterogeneous Signal Processing Systems”, *Journal of VLSI Signal Processing,* vol. 29, pp. 197–206, *Kluwer Academic Publishers,* 2001.
[5] W. Cesario, *et al.*, "Component-Based Design Approach for Multicore SoCs", *Proc. DAC,* 2002.
[6] J.-Y. Brunel, *et al.*, "COSY communication IP's", *Proc. DAC*, 2000.
[7] H. Chang, et al., “Surviving the SOC Revolution - A Guide to Platform-Based Design”, *Kluwer Academic Publishers*.
[8] C. Berthet, “Going mobile: the next horizon for multi-million gate designs in the semi-conductor industry”, *Proc. DAC,* 2002.
[9] OpenDivX, Project Mayo, http://www.projectmayo.com
[10] The Message Passing Interface (MPI) standard, http://www-unix.mcs.anl.gov/mpi/
[11] ARM AP Integrator, http://www.arm.com
[12] OpenMP, http://www.openmp.org/
[13] SystemC, http://www.systemc.org
[14] CoCentric System Studio, http://www.synopsys.com/
[15] ConvergenSC, http://www.coware.com
[16] Platform Express, http://www.mentor.com
[17] Virtex-II Pro FPGAS, http://www.xilinx.com