

Automatic Generation of Hardware dependent Software for MPSoCs from Abstract System Specifications

Gunar Schirner, Andreas Gerstlauer and Rainer Dömer

Center for Embedded Computer Systems

University of California, Irvine, USA

{hschirne,gerstl,doemer}@uci.edu

Abstract— Increasing software content in embedded systems and SoCs drives the demand to automatically synthesize software binaries from abstract models. This is especially critical for Hardware dependent Software (HdS) due to the tight coupling.

In this paper, we present our approach to automatically synthesize HdS from an abstract system model. We synthesize driver code, interrupt handlers and startup code. We furthermore automatically adjust the application to use RTOS services. We target traditional RTOS-based multi-tasking solutions, as well as a pure interrupt-based implementation (without any RTOS).

Our experimental results show the automatic generation of final binary images for six real-life target applications and demonstrate significant productivity gains due to automation. Our HdS synthesis is an enabler for efficient MPSoC development and rapid design space exploration.

I. INTRODUCTION

With the high degree of implementation freedom, designing a modern complex MPSoC is challenging. Traditional development flows inadequately address the vast exploration space offered by the current manufacturing capabilities. System-Level-Design is accepted as one approach to address the complexity challenges. Transaction Level Models (TLM) are widely used for design space exploration and early development. However, such TLMs typically are written manually [15]. Moreover, mostly a TLM is not reused to generate the final implementation [13]. This gap hinders industry from taking full advantage of Electronic System Level (ESL) design. To increase productivity a design flow is needed that spans from an abstract, untimed, and platform-agnostic specification down to an actual implementation on real hardware [13].

In this paper, we present our ESL flow [3] that addresses these issues by using a two-step approach as shown in Figure 1. It first generates a TLM for design space exploration and then uses the same TLM to automatically synthesize the software. Our flow establishes a seamless solution from abstract specification to final software implementation for a MPSoC.

Our design flow allows the user to describe the application independent of the actual implementation (e.g. HW/SW split, implementation of communication). Then, the user can explore the design space by entering architecture decisions into the design flow. The flow automatically generates a TLM that reflects the designers choices [11]. The generated TLM allows for a fast and accurate validation, performance evaluation, prototyping, and debugging of the complete system. Based on the

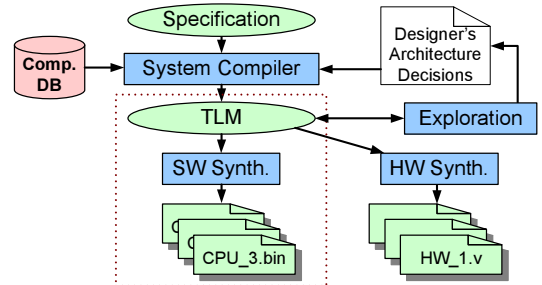


Fig. 1. System Design Flow Overview.

performance results, the designer can then iteratively update the architecture decisions.

Once the designer has determined a suitable platform using the TLM, the same TLM then serves as an input to backend SW generation. Since the TLM is synthesis-complete, SW generation can produce a final binary for each processor in the system fully automatically. It extracts information to generate processor internal communication, external communication, and synchronization from the TLM, and it adjusts for selected choice of multi-tasking.

The two-step approach of first generating the TLM exposes all designer decisions in an early stage of the design. Automatically generating the final binaries avoids a break in the design flow, enables rapid implementation and avoids errors through manual implementation. This makes intermediate TLM critical for exploration, since it reflects all decisions. In this paper, we focus on the synthesis of hardware dependent software from the generated intermediate TLM (highlighted by dotted line).

A. Related Work

System level modeling is important as a means to improve the SoC design process. System Level Design Languages (SLDLs) for capturing systems, jointly with HW and SW, have been developed (e.g. SystemC [12], SpecC [8]). They have been extensively used for modeling software (SW) and its execution in abstract form [17, 10] and in ISS-based co-simulation [2, 5]. However, these approaches focus on simulation without providing an automated path to implementation.

Traditionally, SW synthesis has been addressed from very specific input models and with limited target architecture support, e.g. POLIS [1] (Co-Design Finite State Machine), DESCARTES [20] (ADF and an extended SDF), Cortadella *et al.* [4] (petri nets). With the input choice, these solutions clearly favor a particular application type. These approaches do not provide an intermediate model for exploration that is as versatile as our TLM. Instead, our solution uses a flexible generic C-programming model as an input, produces a TLM for prototyping and provides a path to the final binaries.

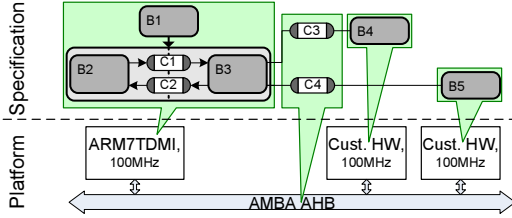


Fig. 2. Example specification with architecture mapping.

Herrera *et al.* [14] describe SW generation from SystemC by overloading SystemC library elements in order to reuse the same model for specification and target execution. However, the approach partly replicates the simulation engine on the target and imposes strict input requirements. Our proposed solution shows neither limitation. Krause *et al.* [18] generate source code from SystemC mapped onto an RTOS. In addition to RTOS mapping, our approach generates detailed communication and synchronization code, and the final target binary.

Gauthier *et al.* [9] describe a method for generating application-specific operating systems and the corresponding application SW. Their work focuses on the OS portion and does not address external hardware (HW). Our solution, on the other hand, explicitly includes heterogeneous external HW. Yu *et al.* [25] show generation of application C code from an SLDL, however without showing the final target binary. Our approach includes the communication synthesis, multi-task adaptation and the generation of the final binary image.

The Phantom Serializing Compiler [19] translates multi-tasking POSIX C code input into flat C code by grouping blocks to Atomic Execution Blocks and custom scheduling them. It is oriented toward a pure SW solution. In contrast, we address SW synthesis in a system context, specifically taking HdS and external communication into account.

II. SOFTWARE ENABLED DESIGN FLOW

We provide a two-step design flow that generates a system TLM for performance estimation and early MPSoC development. Furthermore, the TLM is used to automatically generate SW binaries for all processors in a heterogeneous MPSoC.

The input to the flow is captured in an untimed, platform agnostic algorithmic form using the SLDL SpecC¹ [8]. We assume computation to be grouped in behaviors (or processes). Behaviors are connected via point-to-point channels, selected from a feature-rich set of standardized channel types. These channels allow for synchronous and asynchronous, blocking and non-blocking communication (e.g. FIFO), as well as for synchronization only (e.g. semaphore, mutex, barrier). The upper portion of Figure 2 shows a simple specification example. It contains sequential and parallel executing behaviors. Behavior *B2* and *B3* communicate through channels *C1* and *C2*. *C1* and *C2* are of type "double handshake" (blocking, synchronous, non-buffered). *C3* and *C4* are finite depth FIFO channels.

As a second input, we require the architecture decisions from the designer by an intuitive GUI. Architecture decisions include allocation of processing elements (PEs) (e.g. processors, HW components), mapping of behaviors to PEs. For behaviors on a processor, we require mapping information to tasks and their parameters (e.g. priority). The designer specifies also the mapping of communication to busses. Example mapping decisions are visualized in the bottom portion of Figure 2.

Based on this two inputs, the system compiler [3] automatically generates a system TLM that reflects the architecture de-

¹We use the SpecC SLDL for our experiments. The concepts however, are applicable to other SLDLs such as SystemC as well.

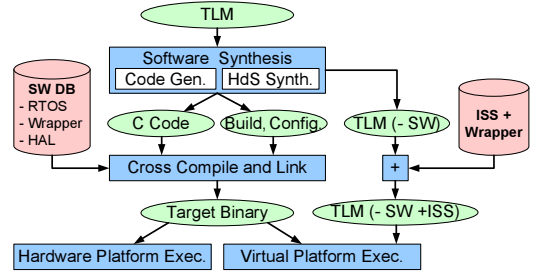


Fig. 3. Software synthesis flow.

isions. The TLM allows for system exploration, performance analysis and debugging (see example in Figure 4). The TLM simulates multiple orders of magnitude faster than a ISS-based model with few percent timing inaccuracy [21].

The same TLM serves as input for the back-end HW and SW synthesis. The SW synthesis produces the final SW binaries, executable on a set of processors composing the platform. It generates the application code, and all drivers for communication in a heterogeneous system. Interrupt and polling-based synchronization is supported, as selected by the designer. The SW application executes on an off-the-shelf RTOS or by using an interrupt driven system for small applications. The generation of the TLM has been described in [11]. This paper focuses on SW synthesis from the generated TLM.

III. SOFTWARE SYNTHESIS OVERVIEW

The SW synthesis, Figure 3, uses the TLM as an input. The TLM reflects all architecture decisions: allocation of computation to processing elements, selection of the scheduling policy and mapping to tasks. Communication is mapped to a set of busses and protocols, and its parameters are defined (e.g. addresses, synchronization type and interrupt allocation). Therefore, the input TLM contains all structural and functional information needed for the target implementation.

Our *software synthesis* is divided into C code generation and HdS synthesis. Our C code generation, similar to [25], translates the hierarchical model in SpecC SLDL to C code, resolving structural hierarchy, port and channel connectivity.

HdS synthesis generates code for processor internal and external communication, including drivers and synchronization (polling or interrupt). It also generates code for multi-tasking execution. To create the complete binary SW image, it finally generates configuration and build files (e.g. Makefile) which select and configure database components (e.g. RTOS, RTOS-port). Using a cross compiler, the final target binary (or binaries) is created, which can execute on the target processor(s) or alternatively on a virtual platform. Our SW synthesis also creates the virtual platform by removing all SW running on each processor from the TLM and replacing it with an Instruction Set Simulator (ISS) model.

IV. HARDWARE DEPENDENT SOFTWARE SYNTHESIS

The HdS synthesis uses the system TLM as an input (see example Figure 4). It was generated by the system compiler based on the architecture decisions. The behaviors *B1*, *B2* and *B3* execute on the processor, while *B4* and *B5* are each mapped to a HW accelerator. The TLM contains hierarchical behaviors, channels and additional HW to reflect the platform.

The HdS synthesis parses the input TLM into an abstract syntax tree and then operates on this tree for code generation. We distinguish three synthesis aspects: communication synthesis, multi-task synthesis and the generation of the final target image. The following sections describe each aspect.

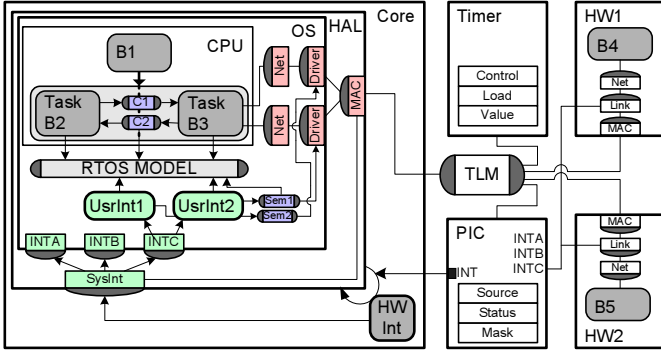


Fig. 4. Processor and application TLM.

A. Communication Synthesis

The communication synthesis deals with processor internal and external communication. In particular, it creates the driver code for communication between the software and external HW. It also generates the code for synchronization, inserts stubs in the application code, and generates interrupt handlers and/or polling code.

Internal Communication. Internal communication takes place between tasks on the same processor. The channels $C1$, $C2$, $Sem1$ and $Sem2$ are used for internal communication in Figure 4. These are instances of our standard channels as also used in the specification. To realize the particular communication on the target system, the abstract standard channels are replaced with a target-specific implementation². For example, a blocking synchronous communication channel is realized on an RTOS-based system with a semaphore, two events and a *memcpy* using the services of our RTOS Abstraction Layer (RAL), see Section B.

External Communication. To support heterogeneous systems, we follow the ISO/OSI layering model [16] for our external communication. The channels $C3$ and $C4$ in the initial specification (Figure 2), which perform external communication are refined by our system compiler to stacks of half channels (*Net*, *Driver*, and *MAC* in Figure 4). Also, corresponding counterparts are inserted in the HW components ($HW1$ and $HW2$). At the top of the stack, the typed data of the initially abstract channel is marshalled into a flat untyped stream, a common representation that can be interpreted regardless of a node’s endianness and padding rules. The type information is extracted from the user type definition captured in the SpecC SLDL and marshalling code using standard conversion functions is generated.

The next half channel *Driver* contains information about the channel’s system-wide addressing and maps the end-to-end channel to a set of point-to-point links. The slave in our example is connected to the processor bus, allowing a direct communication. However, complex communication schemes spanning multiple bus hierarchies are possible. Then, user messages are packetized to minimize buffer requirements of intermediate communication partners. Depending on the information in the *Driver* channel, the corresponding source code is generated.

The driver also implements a channel-specific synchronization mechanism, which will be explained in the next section. The *Driver* transfers the data using Media Access Control (MAC) layer services. According to the platform definition, the HdS synthesis includes a processor-specific MAC implementation, which in a simple case may use the processor’s memory interface.

²Note, the simulation environment is *not* recreated on the target.

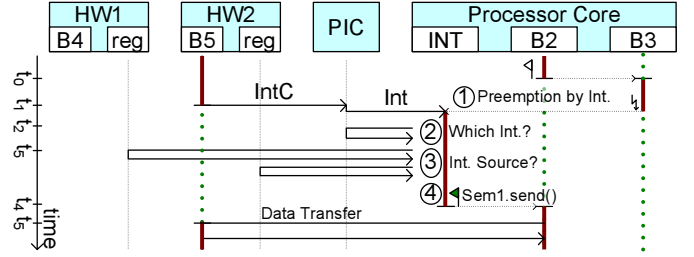


Fig. 5. Events in external communication.

Synchronization. For a typical master/slave bus, external synchronization is required for a slave to indicate it being ready for a data transfer (e.g required data is available). The designer chooses the type of synchronization for each channel to be either polling- or interrupt-based, and may share interrupts between sources. These choices are reflected in the TLM.

If *polling* was chosen, polling code is synthesized as part of the driver code. It accesses the slave’s polling flag using MAC services analogous to external communication. Our HdS synthesis generates polling code that uses RTOS services to maintain the user selected polling period.

In case of *interrupt* synchronization, the TLM contains a set of channels and behaviors modeling this synchronization. See the set of behaviors an channels $UsrInt1$, $UsrInt2$, $INTC$, and $SysInt$ of Figure 4. $IntC$ is shared between $HW1$ and $HW2$. A semaphore channel ($Sem1$, $Sem2$) connects the interrupt handlers with the driver code. To implement interrupt-based synchronization, our HdS synthesis generates a chain of correlated code, which we describe using an event sequence when sending a message from $B5$ to $B2$ (Figure 5).

At t_0 , $B2$ expects the message, waits on the semaphore $Sem1$ and yields execution to $B3$. At t_1 , $B5$ starts sending and signals $INTC$. Hence, the Programmable Interrupt Controller (PIC) sets Int , which triggers the interrupt chain in the processor (labeled 1 through 4).

1. The low-level assembly interrupt handler (part of the RTOS port stored in database) preempts $B3$, stores the current context and calls the system interrupt handler.
2. The System Interrupt Handler ($SysInt$ in the TLM) communicates with the PIC, determines the highest priority pending interrupt, and then invokes the application-specific interrupt handler ($INTC$ in the TLM). The $SysInt$ code is one element of the Hardware Abstraction Layer (HAL) stored in the database.
3. The application-specific $INTC$ determines the source of the shared interrupt by reading a status register in $HW1$ and $HW2$. It then calls the corresponding User Interrupt Handler ($UsrInt2$).
4. $UsrInt2$ calls the semaphore $Sem1$ to release the driver code execution in $B2$. The semaphore channel uses internal communication services (Section A).

After releasing semaphore $Sem1$, the interrupt handler terminates, $B2$ is scheduled and reads the data from $HW2$.

For HdS synthesis, we implement this chain on the processor. The code for 1+2 is taken from the database, 3+4 are generated (3 based on $INTC$ and 4 on $UsrInt2$). Our HdS synthesis generates startup code to register $INTC$ to the system interrupt handler using the TLM’s architectural information. It also generates code to instantiate the semaphore channel and inserts proper calls into the driver code.

B. Multi-Task Synthesis

In order to execute multiple tasks on the same processor, *multi-task synthesis* generates code that uses the underlying

multi-task engine. We support the traditional execution on top of an off-the-shelf RTOS and furthermore provide an alternative of interrupt-based multi-tasking on a naked processor without an OS.

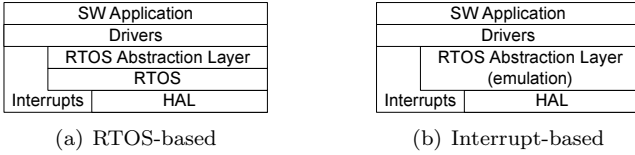


Fig. 6. Software stack.

RTOS-based Multi-Tasking. Our *multi-task synthesis* makes use of a canonical OS interface, which we call the RTOS Abstraction Layer (RAL), see Figure 6(a). The very thin RAL (few hundred lines), abstracts from the particular OS’s function names and parameters. We have chosen the RAL approach to limit the interdependency between synthesis and target RTOS. Also, the processor internal communication uses RAL services to implement our standard channels. To ensure a generic API, we investigated different RTOS APIs (uCOS-II, vxWorks, eCos, ITRON, POSIX).

The input TLM contains mapping of behaviors to tasks (*Task B2*, *Task B3*) and their scheduling parameters. For RTOS-based multi-tasking, the HdS synthesis extracts the task control information from the TLM and generates task creation calls to the RAL. It also realizes the task’s parameter set of the TLM (e.g. priority, stack size) on the target. Our HdS synthesis translates SLDL statements for parallel execution into fork/join services of the RAL.

Interrupt-based Multi-Tasking. In the second case, targeting to a naked processor, the software execution is performed without an RTOS. Instead, interrupts are utilized to provide multiple flows of execution. We support this alternative for systems where RTOS execution is not desirable (very few tasks, execution on a DSP, footprint limitations). Our motivating example of a speech codec implemented on a DSP is shown in Section A.

For this interrupt-based alternative, the RAL (Figure 6(b)) implements an emulation, providing a subset of the RTOS services needed for software execution (e.g. events, processor suspension and interrupt registration).

We assume that each task is composed of a sequence of computation (C), synchronization (S) and data transfers (T) as shown in the example in Figure 7(a). If only interrupts are used for synchronization, then the task main function is transformed into a state machine shown in Figure 7(b). Each synchronization point (e.g. S_1 , S_2) starts a new state (ST_1 , ST_2). The state machine transitions to the next state upon successful synchronization (receiving of interrupt I_1 or I_2). Additional states are inserted to represent conditional execution and loops (ST_0 initialization; ST_1 loop head).

The task’s state machine is then executed in the interrupt handlers, which were initially chosen for synchronization. The task priorities can be preserved by choosing the interrupts according to priority. The lowest priority task executes in the main task (T_{main}), the startup task of the processor.

Listing 1 outlines the produced C implementation. For explanation assume that ST_1 is the current state and that computation C_1 just finished. Next, the synchronization S_1 is checked (line 10). In case it has not yet occurred, the state machine terminates (line 11) and so does the interrupt handler. Receiving the next interrupt I_1 sets $S1.ready$ (line 2) and executes the state machine again (line 3). It then passes con-

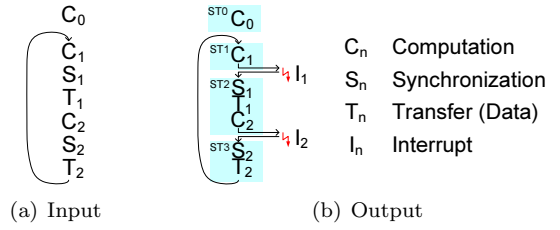


Fig. 7. Reactive task template.

ditional (line 10), receives the data, and executes computation C_2 . The do-while-loop transitions from ST_3 to ST_1 without terminating the interrupt handler.

Listing 1 Interrupt-based multi-tasking excerpt.

```

1 void intHandler_I1 () {
2   release(S1); /* set S1 ready */
3   executeTask0 (); /* task state machine */
4 }
5 void executeTask0 () {
6   do { switch(State) {
7     /* ... */
8     case ST1: C1 (...);
9               State = ST2;
10    case ST2: if (attempt(S1)) T1_receive (...);
11              else break;
12              C2 (...);
13              State = ST3;
14    case ST3: /* ... */
15    } } while (State == ST1);
16 }

```

C. Binary Image Generation

The generation of a complete target binary is the final aspect of HdS synthesis. It generates configuration and makefiles, which control compilation and linking of generated code and database components, as illustrated in Figure 8.

Identifying the dependencies of each component is important for an efficient database. It enables a flexible composition of the final binary, while minimizing code duplication inside the database. The matrix of arrows in Figure 8 symbolizes the dependencies when selecting a component. The most specific element is the RTOS port, since it depends on RTOS, processor, and cross-compiler (call frame and stack layout). The software synthesis generates a customized Makefile, which selects the components and generates the target binary. Automating this step avoids duplication of the system configuration and minimizes the user effort.

V. EXPERIMENTAL RESULTS

To evaluate our approach, we have applied it to six real-life examples. We will describe two examples in more detail.

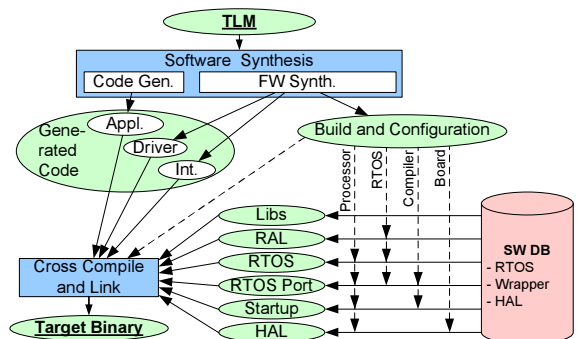


Fig. 8. Synthesis of target image.

A. Interrupt-based Implementation Example

We start by showing a concrete example of an interrupt-based multi-tasking implementation. We implemented a GSM 06.60 [7] transcoder on a Motorola DSP 56600 assisted by a HW accelerator that performs the codebook search, and four HW blocks for I/O. Since the DSP only executes two tasks (and an RTOS port was not available for the DSP), we applied our interrupt-based multi-tasking approach. The encoder executes in T_{main} and the higher priority decoder in the interrupt handler $IntB$. Figure 9 shows the state machine for the decoder task, which consists of 4 states. $ST1$ and $ST2$ have been created due to synchronization (S_1, S_3), which uses the interrupt $IntB$. $ST0$ and $ST3$ are inserted to accommodate initialization and post processing. A speech frame consists of four sub-frames. Therefore, $ST2$ is repeated four times.

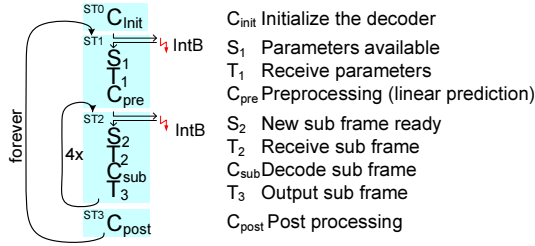


Fig. 9. State machine for GSM decoder.

Figure 10 shows the time line for transcoding one sub-frame. At its start the processor is suspended, waiting for input data. The encoder is triggered at t_1 through $IntC$ and the event $e1$. After feeding the codebook accelerator, the encoder suspends on $e2$ waiting for results. Later at t_3 , $IntB$ signals availability of a sub-frame for decoding. The decoder state machine is executed in the $IntB$ handler in the state $ST2$. It reads the input data (T_2), decodes it (C_{sub}) and writes the results (T_3) to the output HW. The latter needs no synchronization, since the output HW is always available. At t_4 , the decoder is preempted by the higher priority $IntC$ announcing the codebook data availability through $e2$. The encoder resumes at t_6 and finishes at t_7 . The cycle repeats at t_8 with the next sub-frame. In total, 3451 interrupts are triggered, see Table II.

B. Exploration Example

We use an automotive example to illustrate the exploration capabilities with respect to comparing the two multi-tasking approaches. We model an Electronic Control Unit (ECU) containing an ARM7TDMI processor. It executes three tasks: anti-lock break control, RPM computation, and engine fan controller. Six sensors and actuators are connected to two different CAN busses, further three are attached to the processor bus inside the ECU (Figure 11).

We generated code for both approaches, first toward execution on top of the RTOS uCOS-II [22], and second for interrupt-based execution. In the latter case, the fan control was mapped to T_{main} and the other two tasks were converted to state machines. Table I compares the results.

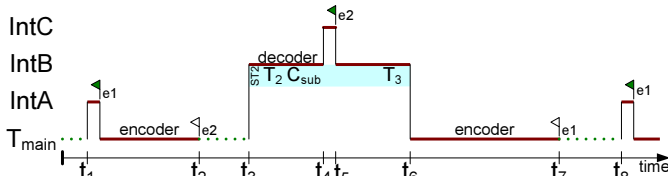


Fig. 10. GSM transcoding execution.

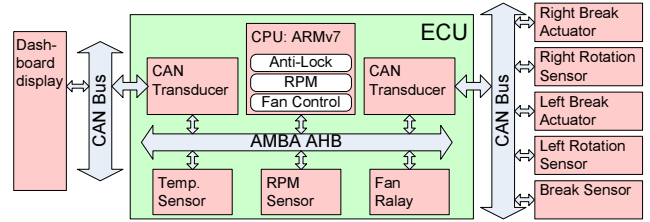


Fig. 11. Automotive Example Application.

The automotive example profits from the interrupt-based solution. Avoiding the RTOS code reduces footprint, since a specific simpler code is used instead. Also, the allocated stack size is reduced, since all tasks share the same stack. The CPU busy cycles drop from 6.7 MCycles to 5.1 MCycles. The RTOS startup is avoided and fewer cycles are needed for the OS functionality (e.g for event handling and context switching) due to simplicity.

As one detail, we analyze the interrupt latency, which we define for this paper as delay between triggering the interrupt wire to the first bus transaction retrieving the data. The latency for the RPM task (until reading the RPM sensor) is shorter (1001 instead of 1794 cycles), due to execution in the interrupt handler itself. Additionally, we counted the occurring interrupts. The number is reduced from 1478 to 1027, since the interrupt-based solution does not use the timer. The number of interrupts for data synchronization remains constant.

Our automotive example clearly benefited from the interrupt-based execution. We position it, where applicable, as an alternative in special cases (very few tasks, strict optimization requirements, or unavailability of an RTOS). Since either implementation can be generated automatically, such an exploration becomes easily possible.

C. Synthesis Results

To show the benefits of automatic HdS generation for a range of applications, we have applied our HdS synthesis to six target applications. The first two are the GSM and the car ECU. Additionally, we examined a JPEG encoder, a SW MP3 decoder, an MP3 decoder with 3 hardware accelerators and a combined system with MP3 decoding and JPEG encoding. Table II summarizes these results. The top section quantifies each target applications' complexity. It ranges from the simple JPEG with 2 I/O blocks to the $Mp3$ HW, which uses 2 I/O blocks, 3 HW accelerators and 4 busses.

Next, the table shows the number of generated lines of code for application and HdS, each for the RTOS-based and the interrupt-based multi-tasking. We have not implemented the GSM in an RTOS-based solution, since we had no RTOS port available for the DSP. Also, we have not realized the $Mp3$ HW + JPEG example in the interrupt-based form, since it uses services we do not intend to replicate with interrupts. In examples with HW support, the HdS code is larger, due to the extra communication. In general a significant amount of code is generated (e.g. 1186 lines for $Mp3$ HW + JPEG). In all

TABLE I
AUTOMOTIVE EXAMPLE RESULTS

Multi-tasking	RTOS-based	Interrupt-based
Footprint	36224 Bytes	21052 Bytes
Alloc. Stacks	4096 Bytes	1024 Bytes
CPU Busy Cycles	6.706 MCycles	5.106 MCycles
Latency RPM Task	1794 Cycles	1001 Cycles
# Interrupts	1478	1027

TABLE II
SW SYNTHESIS AND EXECUTION RESULTS

Example	GSM	Car	JPEG	Mp3 SW	Mp3 HW	Mp3 HW + JPEG
Complexity						
IO/HW/Bus	4/1/1	9/2/3	2/0/1	2/0/1	2/3/4	6/3/4
SW Behaviors	112	10	34	55	54	90
Channels	18	23	11	10	26	47
Tasks/ISRs	2/3	3/5	1/2	1/3	1/8	3/14
Lines of Code, RTOS-based						
Application	-	153	818	13914	12548	13480
HdS	-	649	210	299	763	1186
Lines of Code, Interrupt-based						
Application	5921	210	797	13558	12218	-
HdS	377	575	187	256	660	-
Execution, RTOS-based						
CPU Cycles	-	6.7M	127.7M	185.8M	44.5M	174.6M
CPU Load	-	0.9%	100.0%	100.0%	30.9%	86.6%
Interrupts	-	1478	805	4195	1144	1914
Execution, Interrupt-based						
CPU Cycles	42.0M	5.1M	126.7M	182.3M	43.3M	-
CPU Load	42.5%	0.7%	100.0%	100.0%	30.5%	-
Interrupts	3451	1027	726	4078	1054	-

examples, our HdS synthesis completes in less than a second. Manually writing the HdS would take 12 to 79 hours (assuming 15 lines of correct code per hour [24]³). This translates to a tremendous productivity gain of 44,000x to 120,000x.

To validate the correctness of the generated code, we executed each synthesized target binary on a virtual platform with an integrated ISS (a Motorola ISS for the DSP; SWARM ISS [6] for the ARM). Each application executes functionally correct, yielding an output matching the specification. Table II shows execution statistics of the ISS cosimulation. As in the car example, fewer CPU cycles (busy cycles only) are used in the interrupt-based solution. However, the relative improvement is marginal for the larger computation dominated applications. Similar to before, avoiding the OS timer tick reduces the number of processed interrupts.

VI. CONCLUSIONS

In this paper, we have presented our systematic HdS synthesis approach. Our HdS synthesis consist of three parts: communication synthesis, multi-task synthesis and binary image generation. It generates communication drivers, interrupt handlers and adjusts for the target multi-tasking. Our approach supports targeting toward an existing RTOS. Furthermore, it offers an alternative to use interrupts for multi-tasking if an RTOS-based execution is undesirable.

Our HdS synthesis is an integral part of our ESL flow. Beginning from an abstract model, our flow automatically generates a system TLM based on the designer's architecture decisions. From the generated TLM, our HdS synthesis automatically generates the binaries for each processor in the system. Together, a complete ESL flow for software is provided.

We have demonstrated our ability to automatically generate the final binary image from an abstract specification using 6 real-life target applications: different media applications and a control system. Our HdS synthesis allows to target different processors, platforms and applications.

Automating the tedious and error prone process of manual firmware development results in significant gains in designer

³Note [24] reports 27 lines in extreme programming (for a programmer pair), including debugging and testing time. Note also, we assume a validated specification is available from which we synthesize equivalent correct code automatically.

productivity. Thus it enables rapid exploration of the embedded software design space. In future, we plan to extend the interrupt-based multi-tasking to cover additional OS services and to further extend our SW database.

Acknowledgments. The authors thank the SCE research team at the Center for Embedded Computer Systems at UC Irvine for their support.

REFERENCES

- [1] F. Balarin et al. *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*. Kluwer, 1997.
- [2] L. Benini et al. MARM: Exploring the Multi-Processor SoC Design Space with SystemC. *VLSI Signal Processing*, 2005.
- [3] Center for Embedded Computer Systems, UC Irvine. SoC Environment (SCE). <http://www.cecs.uci.edu/~cad/sce.html>.
- [4] J. Cortadella et al. Task Generation and Compile Time Scheduling for Mixed Data-Control Embedded Software. In *DAC*, Los Angeles, CA, June 2000.
- [5] CoWare. Virtual Platform Designer. www.coware.com.
- [6] M. Dales. *SWARM 0.44 Documentation*. Department of Computer Science, University of Glasgow, Nov. 2000. www.cl.cam.ac.uk/~mwd24/phd/swarm.html.
- [7] European Telecommunication Standards Institute (ETSI). *Digital cellular telecommunications system; Enhanced Full Rate (EFR) speech transcoding*, 1996. GSM 06.60.
- [8] D. D. Gajski et al. *SpecC: Specification Language and Design Methodology*. Kluwer, 2000.
- [9] L. Gauthier, S. Yo, and A. A. Jerraya. Automatic Generation and Targeting of Application-Specific Operating Systems and Embedded Systems Software. *IEEE TCAD*, 20(11), Nov. 2001.
- [10] P. Gerin et al. Scalable and Flexible Cosimulation of SoC Designs with Heterogeneous Multi-Processor Target Architectures. In *ASPDAC*, Yokohama, Japan, Jan. 2001.
- [11] A. Gerstlauer, D. Shin, J. Peng, R. Dömer, and D. D. Gajski. Automatic, Layer-based Generation of System-On-Chip Bus, Communication Models. *IEEE TCAD*, 26(9), Sept. 2007.
- [12] T. Grötter, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer, 2002.
- [13] S. Heinen. HdS from Semiconductors Perspective. In *Hardware dependent Software Workshop at DAC*, San Diego, CA, 2007.
- [14] F. Herrera et al. Systematic Embedded Software Generation from SystemC. In *DATE*, Munich, Germany, Mar. 2003.
- [15] S. Hong et al. Creation and Utilization of a Virtual Platform for Embedded Software Optimization: An Industrial Case Study. In *CODES+ISSS*, Seoul, South Korea, Oct. 2006.
- [16] International Organization for Standardization (ISO). *Reference Model of Open System Interconnection (OSI)*, second edition, 1994. ISO/IEC 7498 Standard.
- [17] T. Kempf et al. A SW performance estimation framework for early System-Level-Design using fine-grained instrumentation. In *DATE*, Munich, Germany, Mar. 2006.
- [18] M. Krause, O. Bringmann, and W. Rosenstiel. Target software generation: an approach for automatic mapping of SystemC specifications onto real-time operating systems. *Design Automation for Embedded Systems*, 10(4):229–251, Dec. 2005.
- [19] A. Nacul and T. Givargis. Lightweight Multitasking Support for Embedded Systems Using the Phantom Serializing Compiler. In *DATE*, Munich, Germany, Mar. 2005.
- [20] S. Ritz et al. High-Level Software Synthesis for the Design of Communication Systems. *IEEE Journal on Selected Areas in Communications*, Apr. 1993.
- [21] G. Schirner, A. Gerstlauer, and R. Dömer. Abstract, multi-faceted modeling of embedded processors for system level design. In *ASPDAC*, Yokohama, Japan, January 2007.
- [22] uCos-II. <http://www.ucos-ii.com>.
- [23] I. Viskic, S. Abdi, and D. D. Gajski. Automatic generation of embedded communication SW for heterogeneous MPSoC platforms. In *LCTES*, Monterey, USA, June 2007.
- [24] W. A. Wood and W. L. Kleb. Exploring XP for Scientific Research. *IEEE Software*, 20(3), May 2003.
- [25] H. Yu, R. Dömer, and D. Gajski. Embedded Software Generation from System Level Design Languages. In *ASPDAC*, Yokohama, Japan, Jan. 2004.