System-level Development of Embedded Software

Gunar Schirner
Electrical and Computer Engineering
Northeastern University
Boston, MA 02115
schirner@ece.neu.edu

Andreas Gerstlauer
Electrical and Computer Engineering
University of Texas, Austin
Austin, TX 78712
gerstl@ece.utexas.edu

Rainer Dömer
Electrical Engineering and Computer Science
University of California, Irvine
Irvine, CA 92697
doemer@uci.edu

Abstract— Embedded software plays an increasingly important role in implementing modern embedded systems. Development of embedded software, and of Hardware-dependent Software in particular, is challenging due to the tight integration with the underlying hardware architecture.

In this paper, we describe our system-level design approach that allows designers to develop software in form of a platform-agnostic specification. Our design environment enables exploration of different architectural alternatives and subsequently generates the software implementation. It generates the application code, communication drivers, and an adaptation to a chosen RTOS. It completes the process by producing the final target binary for each processor. Our experimental results demonstrate the automatic generation of the binaries for five control and media oriented applications.

I. Introduction

Embedded software is increasingly important in todays complex SoCs as it allows to flexibly realize complex features. However, software development cost now dominates the overall design cost. The productivity gap between software design capability and the potential in chip complexity/capacity is increasing [9]. The traditional approach of manual software implementation is tedious and error prone, as well as too time consuming to meet the shortened time-to-market demands.

One potential solution to increase productivity is to raise the level of abstraction for software development, hiding the complexity of low level implementation detail. System-level design has to accommodate software concerns enabling a seamless co-design of software and hardware. Moving to higher levels of abstraction reduces the complexity during development, enabling designers to focus on important algorithmic details without the burden of low level implementation details.

This article gives an overview of the software development aspect within our ESL flow, System-on-Chip Environment (SCE) [8]. The system specification is developed in an platform-agnostic format in our high-level development environment. In a separate process, the designer specifies the target platform and the mapping of the application to that platform. SCE then automatically generates a system model that implements the application on top of the specified platform. The system model, which is a transaction level model, serves as an early validation and performance evaluation platform, giving the designer feedback for optimizing the application and/or the platform to meet design constraints.

Based on the system model, our software synthesis then generates an implementation over the target platform. It generates the necessary application code and the necessary Hardware-dependent Software (HdS) to execute the application on the platform. The HdS includes communication drivers for external and internal communication, as well as a mapping to an underlying RTOS. Our generation process, using a cross compiler and linker, produces the complete target binary for each processor in the system. For early validation of those binaries, a system model with integrated Instruction Set Simulators (ISSs) can be used.

The remaining document is organized as follows. After introducing the relevant related work in Section II, Section III overviews the design environment SCE. Section IV then outlines our processor modeling for estimation of software performance. Section V introduces our software generation approach consisting of code generation and HdS generation. Section VI validates our approach with experimental results, and Section VII concludes and closes the article.

II. Related Work

Supporting the software design process has been the aim of significant research efforts with a wide range of approaches. To name a few, they range from high-level analysis and synthesis approaches that are based specialized Models-of-Computation. Examples include POLIS [1] (Co-Design Finite State Machine), DESCARTES [24] (ADF and an extended SDF), Cortadella *et al.* [5] (petri nets). Integrated Development Environments (IDEs), at another end of the spectrum, typically provide limited automation support but aim to simplify manual development (e.g. Eclipse [10] with its wide range of plug-in modules). The focus of this paper is the software development in context of system-level design, were models are captured in a System Level Design Language (SLDL) (e.g. SystemC [17], SpecC [13]) which enables jointly capturing of HW and SW aspects.

Abstract models are an important means for early prototyp-

978-1-4244-5766-3/10/$26.00 ©2010 IEEE

ing and performance estimation. SLDLs, with their generic C-programming model, are often used for modeling software and its execution environment in an abstract form [21, 15, 3]. For early validation of software binaries, ISSs have been integrated into abstract system models to create system co-simulation environments [2, 6]. Such, virtual platforms allow for a detailed analysis of the system before availability of real hardware, often revealing details not available on the target [20]. These approaches focus on simulation and validation, however, do not offer an integrated solution to generate the final implementation.

Selecting proper dynamic scheduling policies and parameters is essential for an efficient design. Exposing dynamic scheduling effects to guide early, abstract development has been studied. One set of approaches is centered around host compiled RTOSes [19, 11], in which a target RTOS is complied to run on the simulation host as part of the simulation environment. The user application then runs on top of the host compiled target RTOS. Emulating an RTOS API directly on top of the SLDL (without running the target RTOS) offers a higher level of abstraction. For example, Posadas et al. [23] present an abstract RTOS model with a POSIX API on top of SystemC. This approach also include dynamic target execution timing estimation through overloading each operator in the C++-based SystemC model.

Software generation enables transferring the results from abstract modeling into a real implementation. Herrara *et al.* [18] introduce SW generation from SystemC models by overloading SystemC library elements. This approach, in advantage, uses the same model both for specification and target execution, however, it potentially replicates aspects of the simulation engine on the target. Other approaches focus on the RTOS integration, such as Krause *et al.* [22] which generates source code from SystemC for application to RTOS mapping, and Gauthier *et al*. [14] for generation of an application-specific operating system.

III. Design Environment

Electronic System Level (ESL) design addresses the complexity challenges of designing a modern embedded system by raising the abstraction level for design and development. Figure 1 outlines our ESL flow that is implemented in the System-on-Chip Environment (SCE) [8]. Our flow uses a two step approach of first generating a system TLM for detailed performance estimation and early validation, and then using the same TLM in a second step for automatic generation of a SW and HW implementation.

The input to the system design flow is the *specification model*, shown on top. The specification is an untimed and platform-agnostic description of the system's algorithms and their dependencies captured in the C-based SLDL SpecC [13]. Note that although we chose SpecC for our experiments, the concepts shown are equally applicable to other SLDLs, such as SystemC, as well.

Computation and communication are separately captured using distinct language constructs. This separation enables an automatic refinement for mapping computation to separate processing elements and establishing the communication between

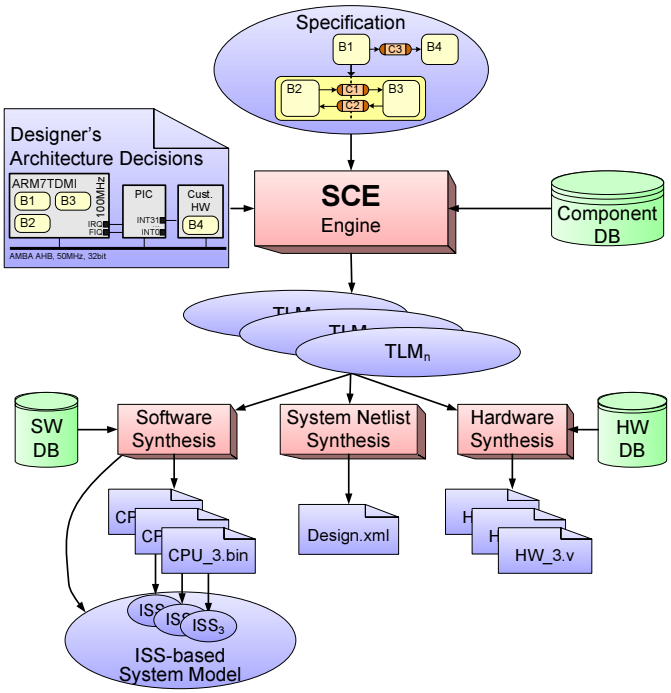

Fig. 1. System design flow overview

elements. Computation is grouped in behaviors, and communication is expressed in channels. Figure 1 contains a graphical representation of a simple *specification*. Boxes with rounded corners (*B1-B4*) symbolize behaviors. Each behavior basically contains C code, which is omitted for brevity. Behaviors can be composed hierarchically to allow complex structures. They can be behaviorally composed to execute in sequence, parallel, pipelined, or state machine controlled.

Behaviors are statically connected and communicate through direct point-to-point channels (*C1*, *C2*, *C3*). These channels are selected from a feature-rich set of standardized channel types, which allow for a wide range of communication mechanisms similar to what is found in an operating system. Communication primitives include synchronous and asynchronous communication, blocking and non-blocking communication (e.g. FIFO), as well as for synchronization only (e.g. semaphore, mutex, barrier).

A second input to our design flow contains the *designer's architecture decisions* describing the target platform as shown on the left of Figure 1. The architecture decisions include the allocation of processing elements (PEs), such as processors, and HW components, and the mapping of behaviors to PEs. The example shows the allocation of an ARM7TMI processor and one custom hardware component. The behaviors *B1*, *B2*, and *B3* are mapped to the processor. These behaviors are later wrapped into tasks and the designer can select important task parameters, such as priority and stack size.

In addition, the designer also selects communication refinement decisions. These decisions include bus system allocation, the mapping of channels to busses and the definition of essential communication parameters for each channel. For example, the user can select the synchronization scheme, such as polling or interrupt-based synchronization.

Based on the *specification* and the *designer's architecture decisions*, the SCE engine [8] then automatically generates a system TLM that reflects the architecture decisions. The SCE engine instantiates and connects components out of the component data base and populates them with the computation captured in the specification model. It refines the communication between processing elements from the standardized abstract channels to a communication based on the selected medium. The resulting model, the generated TLM (Figure 2), is the basis for system exploration, performance analysis and debugging.

After the designer has validated the TLM to meet performance and quality expectations, the same TLM serves then as input for the back-end HW synthesis and SW generation. The output for the hardware side is RTL for each PE as well as the connecting netlist. Within software synthesis, SCE produces a final SW binary for each processor in the platform. The binary includes the application code, all drivers for communication in a heterogeneous system as well as an off-the-shelf RTOS if selected. The generated binary can either directly execute on the target processor or on an ISS-based co-simulation model for binary validation.

The next sections describe the processor TLM, the software synthesis, and the binary validation using an ISS-based model.

IV. Processor Modeling

Abstract processor modeling is one approach to explore software execution performance during design space exploration. In addition, as outlined earlier, the processor model also serves as an input to software synthesis.

The SCE engine generates a system TLM based on the *designer's architecture decisions*, containing a processor model [26] for each processor in the system. Figure 2 illustrates a system TLM. The processor model is captured in the SpecC SLDL and abstracts above processor's Instruction Set Architecture (ISA). In result, the native execution on the simulation host enables highest simulation speeds.

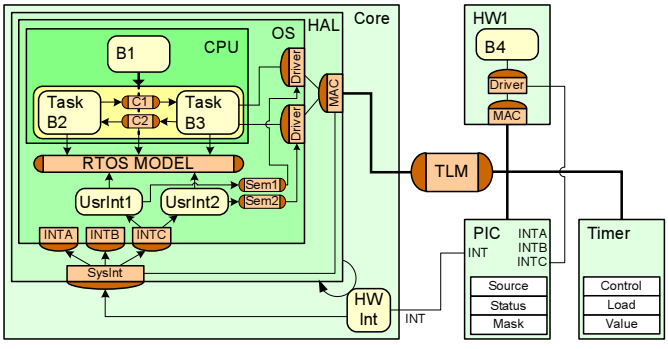

Fig. 2. System Transaction Level Model [12]

Our processor model is organized in layers. The innermost layer (*CPU*) captures the application itself. To emulate target specific execution timing, the SCE engine annotates the application with target-specific *wait-for-time* statements. The delays for those are statically determined through a profiling stage [4].

In order to explore the effects of dynamic scheduling, our processor model includes an abstract processor model [16] in the *OS* layer. It is implemented as a channel (*RTOS MODEL* in Figure 2). In order to execute on top of the OS model, each parallel executing behavior is refined into a task (e.g. *Task B2, Task B2*), defining task execution control and scheduling policy parameters (i.e. task priority). Each primitive inside a task that could potentially trigger scheduling is wrapped to interact with the abstract RTOS model. The OS model, in turn, maintains a state machine for each task, and operates waiting and running queues similar to an actual RTOS implementation. It implements a scheduler and serializes task execution according to the selected scheduling policy and parameters.

The processor model's next layer, the Hardware Abstraction Layer (*HAL*) contains a representation of the necessary drivers for external communication. The driver includes marshalling and demarshalling of user data to translate to and from a common untyped network data format. It furthermore includes low level drivers that implement external bus communication and synchronize with external sources. In particular, it implements the user selected synchronization mechanism, which is either polling- or interrupt-based. The example in Figure 2 uses interrupt synchronization. Together with the *Core* layer, our processor model contains the complete interrupt chain, starting from the external *PIC*, through the system interrupt handler *SysInt*, via the user interrupt handler (e.g. *UserInt2*), and finally through a semaphore *Sem2* that releases the user task executing the driver code. Communication outside the processor is modeled by a transaction level model of the bus, providing cycle approximate simulation with user transaction granularity.

Our processor model expresses the essential features of task mapping, dynamic scheduling, interrupt handling, low level firmware, and hardware interrupt handling. It exposes performance implications of the design choices already early in the process. It executes faster than real-time with a high accuracy [26]. Since it is generated automatically, it serves as an exploration platform in the design space exploration to analyze architectural and mapping alternatives. It allows to validate important scheduling parameters and synchronization options, thus is an important means for system-level software development.

V. Software Generation

Once the designer has achieved the expected performance and quality requirements with the system TLM, the same TLM is then used for software synthesis. To derive the embedded software from the TLM, software synthesis has to implement all SLDL language elements used inside the HAL (e.g. modules, tasks, channels, and port mappings) on the target processor. Instead of compiling the SLDL directly onto the target software platform, software synthesis generates target C code based on the information captured in the TLM to achieve compact and efficient software.

We divide *software generation* into C code generation and *HdS generation*, see Figure 3. C code generation deals with the code inside each task and generates flat C code out of the hierarchical model captured in the SpecC SLDL. HdS generation creates code for processor internal and external communication, adjusts for multi-tasking finally generates configuration and build files (e.g. Makefile) for the cross compilation process.

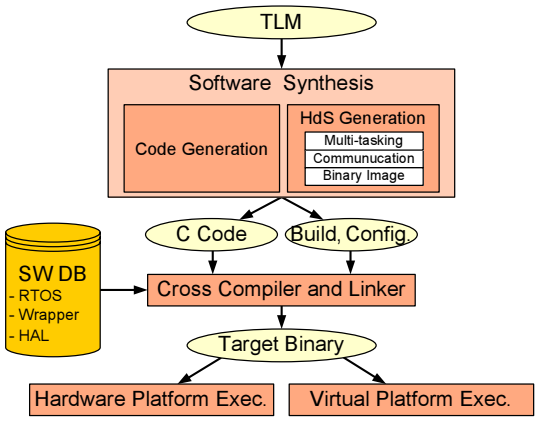

Fig. 3. Software generation flow [27]

A. Code Generation

Code generation [30] produces sequential C code for each task within a programmable component. It translates the application module hierarchy captured in the system TLM into the target programming language. The TLM's application modules use system level features of the SLDL that are not natively present in the targeted C language. Examples include hierarchy, concurrency, and communication encapsulation. In order to implement them on the target processor, code generation implements these SLDL features out of the available language constructs.

It translates the hierarchical composition of behaviors in the SLDL into flat C-code consisting of functions and data structures. ANSI-C does not provide an encapsulation for behavior local storage. Therefore, all behaviors's local variables are added to a behavior-representing structure. It also translates communication between behaviors within the same task into function argument passing. Summarizing, code generation solves similar issues as early C++ to C compilers that translated a class hierarchy into flat C code.

B. Hardware-dependent Software Generation

The second portion of software generation is Hardware-dependent Software (HdS) generation [25]. HdS generation reads the decisions captured in the system TLM, and implements those on the target system. As such, it generates code for processor internal and external communication, including bus access drivers and synchronization code (polling or interrupt). HdS generation also creates code to execute multiple tasks on the same processor, e.g. by targeting an off-the-shelf RTOS. Finally, it generates configuration and build files (e.g. Makefile) that control the cross compilation process to produce the final target binary.

The first aspect within the HdS generation the communication generation which deals with processor internal and external communication. For communication between tasks on the same processor, HdS generation replaces each standard communication channel that is used in the TLM, with an semantically equivalent target implementation, e.g. composed out of RTOS primitives.

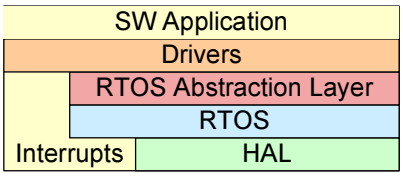

Fig. 4. Software stack for RTOS-based multi-tasking

For external communication, we conceptually follow an ISO/OSI layering model to support communication in heterogeneous systems. HdS generation produces all driver code that is necessary for communication with external PEs. This driver code implements marshalling/demarshalling, packetization, channel-specific synchronization and adds a bus specific Media Access Control (MAC) driver to access the actual processor bus. Among other options, HdS generation offers synchronization through interrupts. For this, it implements the complete interrupt chain on the target processor from the low level assembly interrupt handler that preempts the currently running task, over the system interrupt handler that communicates with the PIC to the user interrupt handler that finally releases the pending driver through a semaphore.

To support multiple tasks executing on the same processor, HdS generation can adjust the task code to run on top of an off-the-shelf RTOS. It replaces the abstract calls for task management in the TLM, with corresponding target implementations. To limit the interdependency between the code generation and the particular OS's API we introduce a very thin RTOS Abstract Layer (RAL) that implements a common canonical API on top of the actual RTOS's API. Figure 4 shows the resulting software stack.

HdS generation finally, produces build- and configuration files to control the cross compilation process and creates the final software binary. Figure 5 illustrates the process. The build process relies on a software database that captures the SW system components (e.g. an RTOS, RTOS port, HAL). The build and configuration files select and configure database components according to the decisions in the TLM. As such, they select a particular RTOS, with a suitable port to the selected processor. They include a hardware abstraction layer (HAL)

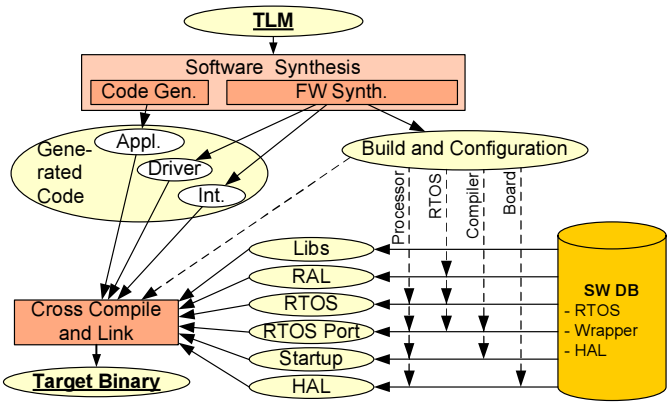

Fig. 5. Compilation and linking generating final binary

based on the target platform, consisting of low-level drivers for the timer, the programmable interrupt controller (PIC), and the bus accesses.

The final target binary is created using a cross compiler and can execute on the target processor or on an ISS-based virtual platform. A virtual platform allows validation and development of the final software binaries already before the availability of real hardware.

Figure 6 shows an ISS-based model for binary validation. To generate a virtual platform, our SW generation removes the internals of the abstract processor model starting with the *HAL* layer and replaces them with an ISS that is wrapped for integration into the system model. The ISS wrapper calls the ISS cycle-by-cycle and advances the system's simulated time according to the processor clock definition. It furthermore interfaces with the remaining design through bus accesses and interrupts.

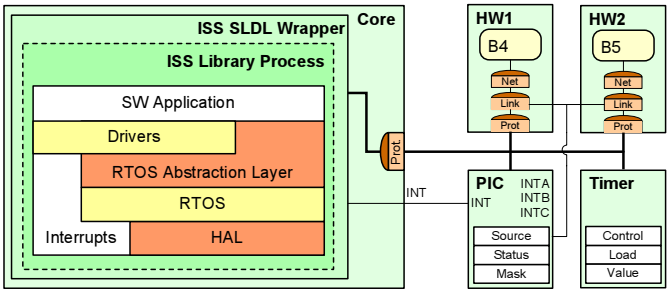

Fig. 6. Binary validation using ISS-based model

VI. Experimental Results

To evaluate the effectiveness of our high-level design approach, we have applied it to five real-life examples. As an control application example, we model an Electronic Control Unit (ECU) containing an ARM7TDMI processor that executes three tasks: anti-lock break control, RPM computation, and engine fan controller. The remaining four examples are multimedia examples. *JPEG* compresses a BMP image. *MP3 SW* decodes an MP3 stream solely on a ARM7TDMI. To demonstrate the flexibility, we use the same specifications in combined examples. To decrease CPU utilization, *MP3 HW* first maps a portion of the synthesis filter to a hardware accelerator. Finally, *MP3 HW + JPEG* combines both media applications. The complexities of the selected target architecture range from a single processor solution to the more complex *Mp3 HW + JPEG*, which uses 4 I/O blocks, 3 HW accelerators and 4 busses (see Figure 7).

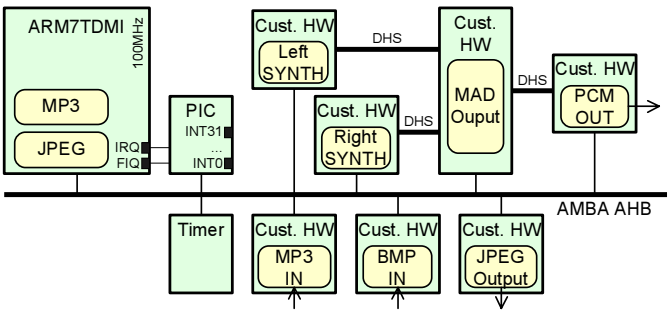

Fig. 7. Example target architecture.

We have used SCE for automatic generation of the system TLM for exploration and finally automatically generated the binaries for each application. To validate the correctness of the generated code, we executed each synthesized target binary on a virtual platform with an integrated ISS, SWARM ISS [7]. Each application executes functionally correct, yielding an output matching the specification.

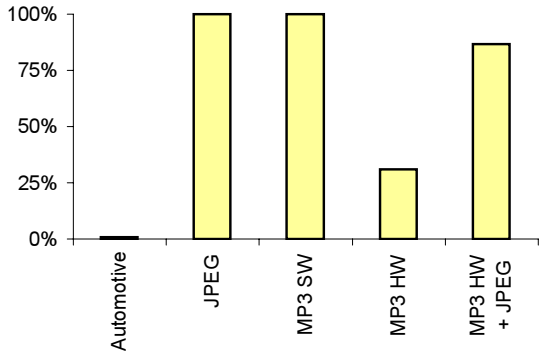

Fig. 8. Processor utilization

Figure 8 depicts the CPU utilization on the ARM7TDMI during the ISS-based simulation. It shows that the *automotive* control application only minimally utilizes the CPU. Both SW only media applications, however, are compute bound and result in 100% utilization. In case of the MP3 decoder, the deadline for frame decoding is exceeded. To meet real-time requirements and decrease the CPU utilization *MP3 HW* maps a compute intense portion of the synthesis filter to a HW accelerator. Here, the CPU utilization drops to 31% creating sufficient idle time for the combined example. After adding JPEG encoding the utilization reaches 90%[1]. Figure 9 illustrates the absolute number of computation cycles on the ARM processor with similar trends like the CPU utilization.

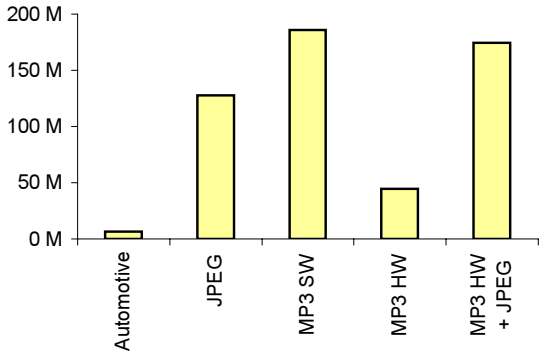

Fig. 9. Execution cycles [MCycles]

Next, Figure 10 shows the number of generated lines of code

[1] *MP3 HW + JPEG* still shows some idle time, since JPEG encoding finishes before the MP3 decoding.

for each application. It distinguishes between application and HdS code. The code size of the *automotive* example is dominated by the HdS as is it uses only a simple control algorithm, however, communicates with 9 sensors and actuators via 2 CAN busses. For the multi-media applications the amount of generated HdS increases with using HW accelerators due to the extra communication. A significant amount of HdS code is generated (e.g. 1186 lines for *Mp3 HW + JPEG*). In all examples, our software synthesis completes within a second. Assuming manually writing only the HdS code would take 12 to 79 hours (assuming 15 lines of correct code per hour [28]). This translates in a multiple thousand fold productivity gain.

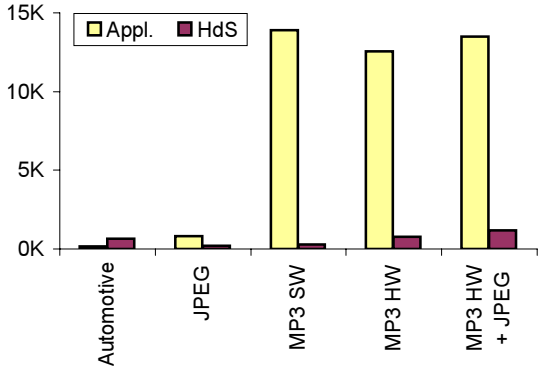

Fig. 10. Generated lines of code

VII. Conclusions

With the increase in system complexity and the increase in software content, manual software development and implementation becomes prohibidly expensive and increases the risk of missing the crucial time-to-market window. Embedded software generation is an essential aspect of implementing todays SoC as automatic generation avoids the tedious and error prone manual implementation.

We have presented in this paper the software aspect of our ESL environment. It implements a systematic approach for generating the final target binaries from an abstract specification model. The design process begins with an abstract model containing the application specification as a set of parallel sequential behaviors that communicate through standardized abstract channels. Our ESL environment automatically generates a system TLM based on the designer's architecture decisions. Using this system TLM, the software generation then automatically generates the binaries for each processor in the system. The software generation includes application code generation, communication generation, multi-task generation, and binary image generation. Based on the platform decisions, it generates communication drivers, interrupt handlers, and adjusts for the target multi-tasking.

Our experimental results show the generation of binaries for five real-life target applications that include different media applications and a control system. An automated software generation has many benefits. Primarily, it increases the productivity by eliminating the error-prone process of manual low-level software development. Automatic generation is not only much faster than a manual implementation, in addition it also allows the designer to focus on the essential algorithms, without the burden of implementation details. Furthermore, an automated generation enables rapid design space exploration, as it can quickly and easily produce alternative solutions.

Acknowledgments

The authors thank the SCE research team at the Center for Embedded Computer Systems at UC Irvine. The team's dedicated and creative work made the advances described in this paper possible.

References

[1] 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.

[2] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivier. MPARM: Exploring the Multi-Processor SoC Design Space with SystemC. *VLSI Signal Processing*, 41:169–182, 2005.

[3] A. Bouchhima, I. Bacivarov, W. Youssef, M. Bonaciu, and A. Jerraya. Using Abstract CPU Subsystem Simulation Model for High Level HW/SW Architecture Exploration. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, Shanghai, China, January 2005.

[4] L. Cai, A. Gerstlauer, and D. D. Gajski. Retargetable profiling for rapid, early system-level design space exploration. In *Proceedings of the Design Automation Conference (DAC)*, San Diego, CA, June 2004.

[5] J. Cortadella, A. Kondratyev, L. Lavagno, M. Massot, S. Moral, C. Passerone, Y. Watanabe, and A. Sangiovanni-Vincentelli. Task Generation and Compile Time Scheduling for Mixed Data-Control Embedded Software. In *Proceedings of the Design Automation Conference (DAC)*, Los Angeles, CA, June 2000.

[6] CoWare. Virtual Platform Designer. `www.coware.com`.

[7] M. Dales. *SWARM 0.44 Documentation*. Department of Computer Science, University of Glasgow, Nov. 2000. `www.cl.cam.ac.uk/~mwd24/phd/swarm.html`.

[8] R. Dömer, A. Gerstlauer, J. Peng, D. Shin, L. Cai, H. Yu, S. Abdi, and D. Gajski. System-on-Chip Environment: A SpecC-based framework for heterogeneous MPSoC design. *EURASIP Journal on Embedded Systems*, 2008.

[9] W. Ecker, W. Müller, and R. Dömer. Hardware dependent software: Introduction and overview. In W. Ecker, W. Müller, and R. Dömer, editors, *Hardware Dependent Software: Principles and Practice*. Springer, 2009.

[10] E. Foundation. Eclipse. `http://www.eclipse.org/`.

[11] T. Furukawa, S. Honda, H. Tomiyama, and H. Takada. A Hardware/Software Cosimulator with RTOS Supports for Multiprocessor Embedded Systems. In *Proceedings of International Conference on Embedded Software and Systems*, Daegu, Korea, 2007.

[12] D. D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner. *Embedded System Design: Modeling, Synthesis and Verification*. Springer, 2009.

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

[14] L. Gauthier, S. Yo, and A. A. Jerraya. Automatic Generation and Targeting of Application-Specific Operating Systems and Embedded Systems Software. *IEEE Transactions on Computer-Aided Design of Intergrated Circuits and Systems (TCAD)*, 20(11), Nov. 2001.

[15] P. Gerin, S. Yoo, G. Nicolescu, and A. A. Jerraya. Scalable and Flexible Cosimulation of SoC Designs with Heterogeneous Multi-Processor Target Architectures. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, Yokohama, Japan, Jan. 2001.

[16] A. Gerstlauer, H. Yu, and D. D. Gajski. RTOS Modeling for System Level Design. In *DATE*, Munich, Germany, March 2003.

[17] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[18] F. Herrera et al. Systematic Embedded Software Generation from SystemC. In *DATE*, Munich, Germany, Mar. 2003.

[19] S. Honda et al. RTOS-Centric Hardware/Software Cosimulator for Embedded System Design. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, Stockholm, Sweden, Sept. 2004.

[20] S. Hong, S. Yoo, S. Lee, S. Lee, H. J. Nam, B.-S. Yoo, J. Hwang, D. Song, J. Kim, J. Kim, H. Jin, K.-M. Choi, J.-T. Kong, and S. Eo. Creation and Utilization of a Virtual Platform for Embedded Software Optimization: An Industrial Case Study. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, Seoul, South Korea, Oct. 2006.

[21] T. Kempf, K. Karuri, S. Wallentowitz, G. Ascheid, R. Leupers, and H. Meyr. A SW performance estimation framework for early System-Level-Design using fine-grained instrumentation. In *DATE*, Munich, Germany, Mar. 2006.

[22] 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.

[23] H. Posadas, J. A. Adamez, E. Villar, F. Blasco, and F. Escuder. RTOS Modeling in SystemC for Real-Time Embedded SW Simulation: A POSIX Model. *Design Automation for Embedded Systems*, 10(4):209–227, Dec. 2005.

[24] S. Ritz, M. Pankert, V. Zivojnvic, and H. Meyr. High-Level Software Synthesis for the Design of Communication Systems. *IEEE Journal on Selected Areas in Communications*, Apr. 1993.

[25] G. Schirner, R. Dömer, and A. Gerstlauer. High-level development, modeling and automatic generation of hardware-dependent software. In W. Ecker, W. Müller, and R. Dömer, editors, *Hardware Dependent Software: Principles and Practice*. Springer, 2009.

[26] G. Schirner, A. Gerstlauer, and R. Dömer. Abstract, multifaceted modeling of embedded processors for system level design. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, Yokohama, Japan, January 2007.

[27] G. Schirner, A. Gerstlauer, and R. Dömer. Automatic Generation of Hardware dependent Software for MPSoCs from Abstract System Specifications. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, Seoul, Korea, Jan. 2008.

[28] W. A. Wood and W. L. Kleb. Exploring XP for Scientific Research. *IEEE Software*, 20(3), May 2003.

[29] S. Yoo, G. Nicolescu, L. Gauthier, and A. Jerraya. Automatic Generation of Fast Timed Simulation Models for Operating Systems in SoC Design. In *DATE*, Paris, March 2002.

[30] H. Yu, R. Dömer, and D. Gajski. Embedded Software Generation from System Level Design Languages. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, Yokohama, Japan, Jan. 2004.

[31] H. Zabel, W. Müller, and A. Gerstlauer. Accurate RTOS modeling and analysis with SystemC. In W. Ecker, W. Müller, and R. Dömer, editors, *Hardware Dependent Software: Principles and Practice*. Springer, 2009.