Model Based Synthesis of Embedded Software

Daniel D. Gajski, Samar Abdi, Ines Viskic

Center for Embedded Computer Systems
University of California, Irvine, CA 92617
{gajski, sabdi, iviskic}@uci.edu

Abstract. This paper presents SW synthesis using Embedded System Environment (ESE), a tool set for design of multi-core embedded systems. We follow a design process that starts with an application model consisting of C processes communicating via abstract message passing channels. The application model is mapped to a platform net-list of SW and HW cores, buses and buffers. A high speed transaction level model (TLM) is generated to validate abstract communication between processes mapped to different cores. The TLM is further refined into a Pin-Cycle Accurate Model (PCAM) for board implementation. The PCAM includes C code for all the communication layers including routing, packeting, synchronization and bus transfer. The generated embedded SW provides a library of application level services to the C processes on individual SW cores. Therefore, the application developer does not need to write low level SW for board implementation. Synthesis results for an multi-core MP3 decoder design, using ESE, show that the embedded SW is generated in order of seconds, compared to hours of manual coding. The quality of synthesized code is comparable to manually written code in terms of performance and code size.

1 Introduction

Multi-core embedded systems are being increasingly used to meet the complexity and performance requirements of modern applications. Embedded application developers need a library of communication services to validate and debug their multi-threaded code. On the other hand, system designers need to provide board prototypes and system SW for application development. Model based design is widely seen as an enabler for early application development before the prototype is ready. Software simulation models for multi-core embedded systems may be created at various levels of abstraction for different purposes. Models at higher abstraction levels, such as TLM, execute faster and are therefore better for application development. However, with higher abstraction, there are fewer design details to allow realistic estimation of design metrics. Pin-cycle accurate models (PCAMs) provide accurate performance estimates and are required for prototyping. However, they are too slow to use for application development. Furthermore, PCAMs require an implementation of **core, platform and application-specific** system SW services on top of the SW core's instruction set. Some of these services are available directly in an RTOS for the SW core. Others, such as external communication methods, must be manually written or may require RTOS configuration.

Integrated design environments, such as ESE [3], are needed to transform application level models into platform specific TLMs for exploration and PCAMs for implementation. In this paper we will discuss the model based design methodology of ESE, with focus on embedded SW synthesis. Our methodology and synthesis technique allows automatic transformation of application level models with abstract message passing communication into PCAMs with an embedded SW stack of communication services. The automation not only cuts design time, but results in modular embedded SW that is consistent with the application level model.

2 Related Work

There has been significant research in model based design for embedded systems in the recent years. Standardization approaches such as AUTOSAR [2] and OSEK [4] provide common API and middleware for automotive SW development. On the other hand, system level design languages such as SystemC [5] and SpecC [9] allow multi-core system modeling with simulation speeds suitable for SW development. Such efforts have provided the groundwork for developing and deploying model automation tools such as the one presented in this paper.

There has also been much work in embedded system modeling frameworks and SW code generation from specific input languages. POLIS [7] (Co-Design Finite State Machine), DESCARTES [19] (ADF and an extended SDF), Cortadella [8] (petri nets) and SCE [10] (SpecC) provide limited automation for SW generation from certain models of computation. In contrast, our approach provides a C based input with multi-core support and has been demonstrated with actual board implementation.

Modular communication modeling has been proposed for application domains such as real-time systems and platforms such as heterogeneous multi-core systems. Kopetz [13] proposes component model for dependable automotive systems. Sangiovanni-vincentelli [21] has proposed a three phase simulation model for platform based design. These approaches tackle security, dependability and heterogeneity at the system level, but require underlying SW services and tools to implement the models. Communication optimization techniques [18, 20, 17] on the other hand have dealt primarily with platform and application transformations using simulation models. In contrast, our communication SW synthesis focuses on code generation for accurate optimization feedback and is fast and flexible enough to incorporate application and platform modifications on the fly.

Hardware dependent SW [15] has been a topic of active research lately and our work contributes to it. Commercial vendors provide a board support package (BSP) [6, 1] with their board IDEs, but such software is customized for the limited set of IP cores available or synthesizable on the board. Most academic approaches so far have dealt with porting of simulation models on RTOS, discounting external communication. Herrara [12] proposes overloading SystemC library elements to reuse the same model for specification and target execution, but partly replicates the simulation engine on the host and thereby imposes strict input requirements. Krause [14] proposes generation of source code from SystemC mapped onto an RTOS, while Gauthier's method [11] provides generation of application-specific RTOS and the corresponding application

SW. Both techniques cannot be extended to muti-core platforms with inter-core communication synthesis. Yu [23] shows generation of application C code from concurrent SpecC, which requires the initial system modeling to be done in SpecC. The Phantom Serializing Compiler [16] translates multi-tasking POSIX C code input into sequential C code by custom scheduling, but is a purely SW core-specific optimization. Schirner [22] also proposes hardware dependent synthesis from SpecC models but only considers platforms with single core connected to several peripherals. In contrast to all the above techniques, ESE provides generation of core, platform and application-specific embedded SW for multi-core systems, starting from an abstract C based application model.

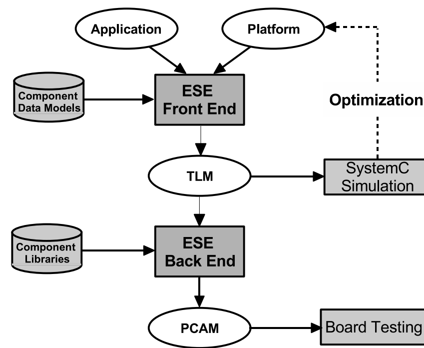

Fig. 1: ESE Design Flow.

3 Model Based Design with ESE

Our model based design methodology is shown in Figure 1. We start with an application model that consists of C processes communicating via synchronized point-to-point handshake channels and shared variables. The platform definition is a graphical net list of processing elements (PEs), buses and transducers. Processes and variables in the application model are mapped to the PEs in the platform. Channels are mapped to routes in the platform. If the route includes a buffer, then the communicated data may need to be broken up into smaller packets according to the buffer size limitations. The above design decisions and data models of PEs, buses and RTOSes are used by the *ESE Front-End* to generate a TLM. The TLM models the PEs as SystemC modules connected to the communication architecture model consisting of bus channels and buffer modules. The original application processes are encapsulated as SystemC threads instantiated inside the PE modules. The point-to-point channel accesses of the application model are mapped into equivalent packet transactions routed over the communication model.

The step of refining the TLM into a PCAM is performed by the *ESE Back-End*. The component data models in TLM are replaced with respective implementation libraries in the PCAM. Synchronization is modeled in the TLM via abstract SystemC flags and events. The flag and event accesses must be transformed into interrupts or polling in the PCAM. Similarly, the packet transactions over the bus channels in the TLM must

be transformed into equivalent arbitration and data transfer cycles on the system buses. The transformations applied to the model result in various C functions per SW core. These functions form the embedded SW library for that core. If there are HW IPs in the platform, they will require RTL interface blocks for the same functions, with platform specific timing constraints. In this section, we will discuss the above models in greater detail to provide an idea of the input and output of the embedded SW synthesis process.

3.1 Platform Template

In order to automate the synthesis of embedded SW, we first need to define the platform components and connections. The platform is composed of processing elements (PEs), memories, buses and transducers. PEs are our generic term for HW and SW cores on which application processes are mapped. Memories are storage cores that do not have any active thread of computation. Shared variables in the application are mapped to memories. Buses are generic communication units that can act as point-to-point links or shared buses with arbitration. Buses have well defined protocols and may connect to compatible ports on a given core.

Transducers are generic interface cores that provide functionality of (1) protocol conversion and (2) store-and-forward static routing. Transducers consist of internal buffers and may connect to incompatible buses via different ports. For each bus connection, they have an IO interface and a *Request Buffer*. This request buffer stores all send/receive requests made to the transducer for storing and forwarding data on a channel. Thus, they allow sending data from one PE to another if the two PEs are not connected to a common bus. A route in the platform is a sequence of buses and transducers with the following regular expression:

$$PE_{sender} \rightarrow Bus_0 \rightarrow [\, Transducer_i \rightarrow Bus_i \rightarrow \,]^* \; PE_{receiver}$$

Channels in the application are mapped to routes in the platform. As a result, each transducer in the platform may have several channels routed through it. For each such channel, the transducer defines (1) a unique buffer partition to be used by data on that channel, (2) a unique bus address for a send request, and (3) a unique bus address for a receive request. Since transactions on a channel are sequential, the partitioning of transducer buffers guarantees safety and liveness of implementation, provided the application model is safe and live.

3.2 Application Model

Figure 2. shows the application model of an MP3 Decoder. The decoding algorithm is captured with a set of eight concurrent processes, each executing sequential C code. Process *Huffman Decoder* inputs MP3 stream organized in frames, performs Huffman decoding, re-quantization and frame reordering. The frames are then classified into either left or right stereo stream and processed separately. *Left* and *Right Alias Reduction* processes reduce the aliasing effects in frames, while the *Left and Right IMDCTs* convert the frequency domain samples to frequency sub-band samples. The two *DCT* processes transform the individual frequency sub-bands into PCM samples and send them to the PCM process for correction verification.

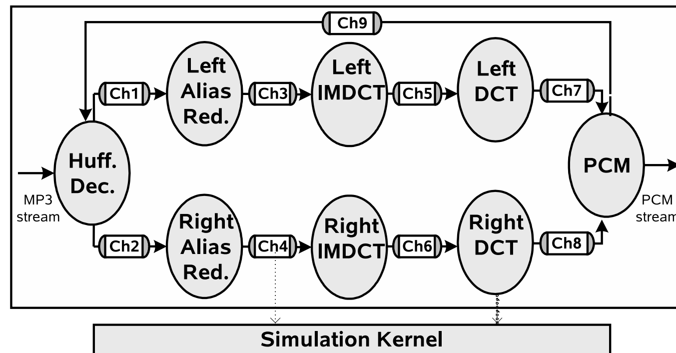

Fig. 2: Application model.

Communication in application model is enabled with calls to (a) $send/recv$ methods for direct process communication, and (b) $read/write$ methods for accessing variables shared between processes. The $send/recv$ methods are encapsulated in *process-to-process channels* with no message buffering. Instead, process-to-process channels follow handshake synchronization semantics, where the receiver process blocks until the sender has sent the communicated data. All communication in MP3 Decoder is modeled using process-to-process channels *Ch1* through *Ch9*.

On the other hand, the communication with $read/write$ methods is unblocking. The shared variables are in the global scope and are accessed with unsynchronized *access channels*. The two communication mechanisms are sufficient to model more complex communication services such as FIFOs, mutexes, mailboxes or events. Therefore, the synthesis of the basic communication models of handshake channels and shared variable access channels is necessary and sufficient for implementing any inter-process communication service at this level of abstraction.

The set of processes, variables and channels are built on top of the SystemC simulation kernel, as shown on Figure 2. The processes execute as concurrent threads on the simulation kernel. The process to process channels use the notify-wait semantics of the kernel events to implement handshake synchronization. The shared variables are modeled as passive SystemC modules that export read and write interfaces, which are used to connect them to the access channels. Interfaces are also defined for processes to allow connection to channels. A well defined interface template provides a communication API with the following functions, where $< i >$ is the name of used interface:

- $< i >$*_Send(void *data, int size)* Synchronized send
- $< i >$*_Recv(void *data, int size)* Synchronized receive
- $< i >$*_Write(void *data, int size)* Non-blocking write
- $< i >$*_Read(void *data, int size)* Non-blocking read

By separating the communication interface from the rest of the computation code, we are able to successively refine only the interface implementation code. The API provided to the application developer stays the same throughout SW synthesis.

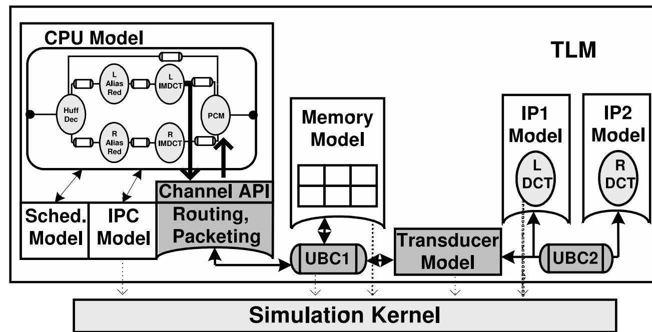

Fig. 3: TLM resulting from application to platform mapping.

3.3 Transaction Level Model

The TLM is derived by mapping the application model in Section 3.2 to an embedded platform. The platform components are modeled with a well defined SystemC code template. PEs are modeled as SystemC modules that instantiate application processes. The system buses are modeled with a *universal bus channel* (UBC), that provides methods for synchronized send/receive, non-blocking read/write and memory service. Memories are modeled as SystemC modules with a local array. Transducers are modeled as SystemC modules with local buffer and controller threads for each bus interface.

Figure 3 shows the TLM of the MP3 Decoder. Processes *Left* and *Right DCT* are mapped to the HW units ($IP1$ and $IP2$), while all other processes reside in a SW core (*CPU*) model. The route between the core and the HW units includes two UBCs and a *Transducer*. Access to units from the SW core is modeled with *Channel API* that encapsulate routing and packeting methods. These methods in turn are implemented with the UBC functions. Routing includes programming the *Transducer* with encoded route using UBC *write* method. Packeting divides the message into data packets of selected size. Since multiple processes are mapped to the SW core, a dynamic scheduler model that exports a threading API emulates processor multitasking.

Channels between processes in the SW core are implemented with an inter-process communication (IPC) model. The IPC and scheduler model are only core dependent and can be included into the TLM from a library. However, the external communication code is application, platform and core dependent. Therefore, its has to be generated for every communication change in the design.

3.4 Pin-Cycle Accurate Model

The TLM is refined into a PCA model that is used for board implementation. Board design tools such those from Xilinx and Altera can be used to convert PCAMs into bitstreams for board implementation. Board debugging tools can then be used to run and debug the design in real time.

Figure 4. shows the PCAM of the MP3 Decoder. The platform consisting of one SW core and two IP units connected with two buses and a transducer is now modeled in synthesizable RTL. The six MP3 Decoder processes mapped to a SW core are compiled

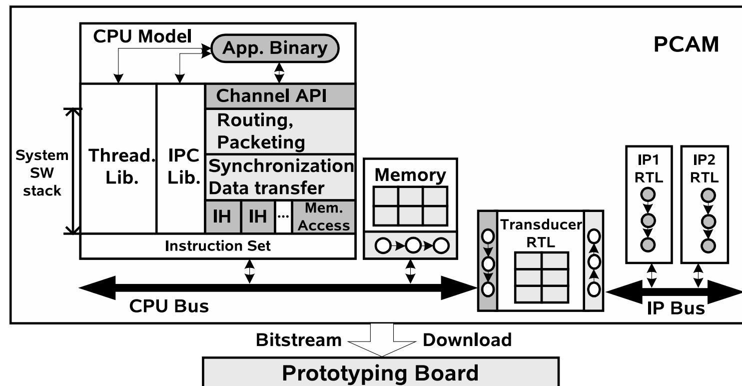

Fig. 4: PCAM refined from TLM for board prototyping.

with the appropriate C compiler (e.g. Xilinx compiler for Microblaze core) and linked with the system SW libraries for download. The processes mapped to hardware can be either synthesized using C-to-RTL tools or replaced with the respective RTL IP. The system SW stack includes the threading and IPC libraries of the RTOS, and the external communication library generated by our synthesis tool. The RTOS itself may consist of several other services such as file handling, memory management, standard C library, networking and so on.

The communication SW library consists of four layers as shown in Figure 4. The lowest layer consists of a set of interrupt handlers (IHs) and memory access functions. Each application level handshake channel requires synchronization that may be implemented as interrupt or polling. For interrupt based synchronization an IH is implemented per handshake channel. For polling implementation, a memory mapped flag is implemented in the slave device that is periodically checked by the master SW core. The memory access functions also provide basic IO to the peripherals. The synchronization and data transfer layer consists of C methods that use the IHs and memory access methods to manage packet level synchronization and bus word transfers. The higher level layers for routing and packeting and the channel API are imported directly from the TLM. In summary, the communication in PCAM is implemented with core specific C methods as opposed to SystemC kernel methods in TLM.

4 Embedded SW Generation

In this section we describe the embedded SW synthesis and code generation from a set of design parameters. The design parameters are determined from the application and platform decisions as well as core properties and are treated as constants for SW code generation. Two layers of communication functions are generated,namely for routing/packeting and synchronization/transfer. These functions are specific to the interface of the application process. An example shows a typical code synthesized for a *Send* interface.

4.1 Communication Design Parameters

In order to automate the communication SW code generation, we define a set of communication specific system parameters. Based on our platform template, explained in Section 3.1, we define a *Global Static Routing Table (GSRT)*. The GSRT stores the mapping of each application level channel to a platform route. For each channel *Ch*, routed through a transducer *Tx*, we define *BufferSize(Tx, Ch)* to be the buffer partition size in bytes for *Ch* on *Tx*. We also define the transducer send and receive request buffer addresses per channel as *SendRB(Tx, Ch)* and *RecvRB(Tx, Ch)*, respectively. The above parameters are required to generate routing and packeting layers for the SW core.

For each channel Ch, routed over a bus B, we define $SyncType(B, Ch)$ to be the synchronization method to be used for ch for the route segment at B. The two possible synchronization methods are *Interrupt* and *Polling*. For direct memory accesses that do not require routing through transducer, synchronization is not required. A synchronization flag table is maintained for each core. Each channel *Ch* gets a unique entry *SyncFlag_Ch* in this table. For interrupt based synchronization, we also define a binding from the interrupt source to the flag and the handler instance. For polling, the flag is bound to an address in the slave PE. Finally, for the data transfer implementation, we define the bus word size and the low to high address range for each channel Ch on bus B as $AR(B, Ch)$. For each SW core we also define *WordSize* as the number of bytes per word.

4.2 Routing and Packeting

The communication functions are synthesized for each interface *i* that is bound to a channel *Ch*. Since we allow only static routing, a route object *Rt* is stored in the *GSRT* corresponding to each channel. Note that the GSRT does not need to be part of the communication library, since the routing per channel is static. The route for *Ch* determines the channel packet size as follows:

$$PktSz = Min\ (\ \forall Tx \in Rt,\ BufferSize(Tx, Ch)\ )$$

Hence, packet size is the largest data size that can fit into any transducer buffer allocation for Ch. Again, note that $PktSz$ is a constant per channel, due to static routing.

The code generated for the interface communication method is a do-while loop, with a temporary variable to keep track of already sent/received data. A lower level method *i_SyncTr* is called by the routing/packeting layer to synchronize with the corresponding process and send or receive each packet.

4.3 Synchronization and Transfer

The routing of channel *Ch* determines the synchronization code generated inside the *i_SyncTr* method. Given the route object *Rt*, as obtained from the GSRT, we determine the first bus *B* in *Rt*. We also determine if *Rt* contains any transducers. If so, we assign *Tx* to be the first transducer in *Rt*. The first step of packet synchronization is top make a transducer request for the transaction. This is done by generating code to write the packet size (in bytes) into the request buffer at the address given by the parameter $SendRB(Tx, Ch)$ or $RecvRB(Tx, Ch)$, depending on the transaction type. Once the

request is written, the transducer initiates lower level synchronization via interrupt or polling, just like any other slave core.

Lower level synchronization is implemented by generating code for busy waiting over flag *SyncFlag_Ch* in the *i_SyncTr* method. The flag is either set by the interrupt handler for *Ch* or by the corresponding slave core, in case of polling. The busy-wait code is followed by *resetting* the synchronization flag. Finally, data transfer is performed by generating a call to the core-specific *WrMem* or *RdMem* functions. These functions write or read data of given bytes using bus transactions of size *WordSize*. The starting address of the transfer is obtained from the address range *AR(B,Ch)*.

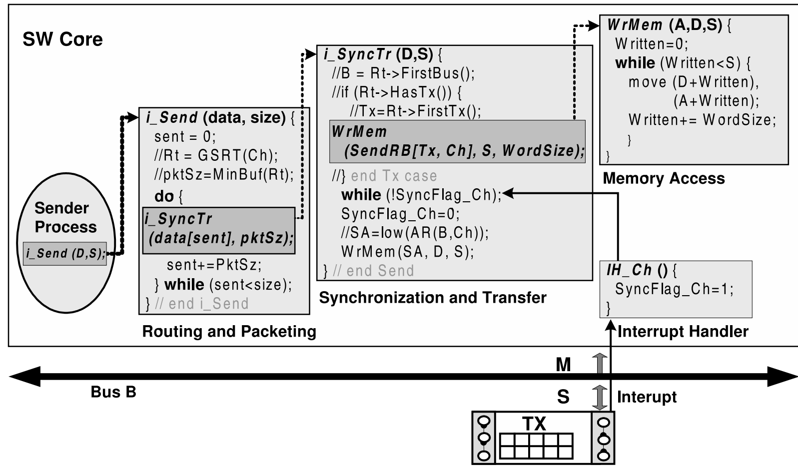

Fig. 5: Embedded SW code example

Figure 5 shows an example for the embedded SW code generated for send method of interface *i*. The sender process is mapped to a SW core, and its interface *i* is connected to bus *B*. Interface *i* is bound to channel *Ch* that is routed over *B* and transducer *Tx* and onto the destination core. Interrupt signal (*Interrupt*) from the transducer to the SW core is used for synchronization, and is bound to handler *IH_Ch* and flag *SyncFlag_Ch*.

5 Experimental Results

Figure 6 shows a multi-core design with an MP3 decoder application mapped to a platform consisting of one SW core (*Microblaze*) and four HW cores (*Left/Right DCT and IMDCT*) used as accelerators. The HW cores use a *DoubleHandshake (DH) Bus* interface, while the SW core is connected to the *Open Peripheral Bus (OPB)*. Since the two bus protocols are incompatible, a transducer is used to interface between the cores. The block diagram of the stereo MP3 application with left and right channel decoding blocks is shown inside *Microblaze*.

We created four mappings of the application, that we refer to as *SW+1DCT, SW+2DCT, SW+2IMDCT* and *SW+2DCT+2IMDCT*, with parts of the application

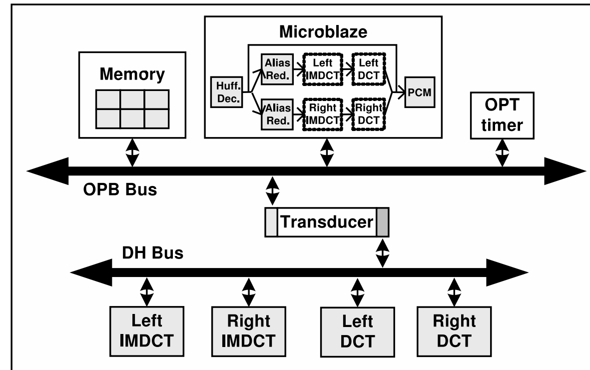

Fig. 6: MP3 Decoder Platform: SW + 2 DCT + 2 IMDCT.

	Design	Code size(in bytes) (% diff.)	Total comm. delay (in cycles) (% diff.)	Total comm. delay (in ms)
Manually implemented PCAM	**SW+1DCT**	171,362	957,060	35.45
	SW+2DCT	160,640	1,914,120	70.89
	SW+2IMDCT	163,492	1,875,588	69.46
	SW+2DCT+2IMDCT	153,420	3,789,708	140.36
Automatically generated PCAM	**SW+1DCT**	172,072 (+4.14%)	949,932 (-7.44%)	35.18
	SW+2DCT	161,280 (+3.98%)	1,899,864 (-7.44%)	70.04
	SW+2IMDCT	164,132 (+3.91%)	1,863,972 (-6.19%)	69.04
	SW+2DCT+2IMDCT	153,624 (+1.33%)	3,763,836 (-6.83%)	139.40

Table 1: Comparison of manual vs. synthesized PCAMs of the MP3 Decoder

mapped to the hardware accelerators, as indicated by the mapping name. As the DCT and IMDCT processes are moved from SW core to the HW cores, the inter-core bidirectional channels are routed over the OPB, DH buses and transducer Tx. The communication SW on *Microblaze* for PCAMs of the different designs are generated using our SW synthesis tool. Xilinx EDK [6] is used to convert our generated PCAMs into bitstream for implementation on the FF896 Virtex-II device. The decoding performance for all the synthesized designs is measured with an OPB timer on the board, using a common MP3 input file.

Table 1 shows a comparison between manually implemented and automatically synthesized PCAMs using quality metrics of SW code size and communication delay. It can be seen that the synthesized SW binary is only marginally larger than manual implementation (between 1-4%). However, the performance of the synthesized code, as measured by the on-chip timer, is 6-9% better than manual implementation. The code quality difference was because the manual implementation shared the synchronization function for different application channels, while the synthesized code had unique synchronization function for each channel. Therefore, the manual code had fewer total instructions, but incurred more instruction fetches for each communication call at run-time.

Table 2 shows a comparison of lines of code between manual and synthesized embedded SW. Due to difference in synchronization implementation, as mentioned above,

	Design	Code size (in lines) (% diff.)	Development Time (% diff.)
Manual communication library	**SW+1DCT**	162	5 h + 2 h
	SW+2DCT	192	5 h + 2.5 h
	SW+2IMDCT	192	5 h + 2.5 h
	SW+2DCT+2IMDCT	252	5 h + 3.5 h
Synthesized communication library	**SW+1DCT**	168 (+3.70%)	5 h + 0.14 s (-28%)
	SW+2DCT	208 (+8.33%)	5 h + 0.14 s (-33%)
	SW+2IMDCT	208 (+8.33%)	5 h + 0.14 s (-33%)
	SW+2DCT+2IMDCT	288 (+13.83%)	5 h + 0.14 s (-37%)

Table 2: Comparison of manual vs. synthesized communication SW

we can see that synthesized source code is marginally larger than manual code. The development time includes the 5 hours that it took to define the application level channels and the design parameters. It took 2-4 hours to implement and test the manual communication code. In contrast, with the given parameters, our synthesis tool generated the embedded SW library in fraction of a second. This resulted in an overall development time savings of 33% on average.

6 Conclusions

We presented a model based technique and methodology for synthesis of embedded SW for heterogeneous multi-core systems. The novelty of our work lies in defining embedded system models at different abstraction level with clear synthesis semantics. Application level models were defined as a set of processes communicating via message passing channels and shared variables. A well defined, yet highly flexible, platform template and associated design parameters were presented. We also presented a synthesis procedure to generate core, application and platform specific embedded SW for the design. Synthesis results for an MP3 decoder example demonstrated the applicability of our technique for large industrial size embedded systems. Our automatic embedded SW synthesis reduces overall design time, while consistently providing better performance and negligible increase in code size over manual implementation. For future work, we are investigating SW synthesis from dependability and security oriented application models. We are also working extending our model based design framework with application and platform templates for real-time architectures such as time triggered network.

Acknowledgments. This work builds on several years of system level design research at Center for Embedded Computer Systems, UC Irvine. We wish to thank Hansu Cho for providing the Verilog implementation of transducers, Pramod Chandraiah for the C reference of the MP3 Decoder, and Gunar Schirner for discussions on hardware dependent software.

References

1. Altera SOPC Builder[online]. Available: http://www.altera.com/.

2. Automotive Open System Architecture[online]. Available: http://www.autosar.org/.
3. Embedded System Environment[online]. Available: http://www.cecs.uci.edu/~**ese**/.
4. OSEK[online]. Available: http://www.osek-vdx.org/.
5. SystemC, OSCI[online]. Available: http://www.systemc.org/.
6. Xilinx Embedded Development Kit[online]. Available: http://www.xilinx.com/.
7. F. Balarin and et al. *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*. Kluwer, 1997.
8. J. Cortadella and et al. Task generation and compile time scheduling for mixed data-control embedded software. In *Proceedings of the Design Automation Conference*, June 2000.
9. D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, January 2000.
10. A. Gerstlauer, D. Shin, J. Peng, R. Domer, and D. D. Gajski. Automatic, layer-based generation of system-on-chip bus communication models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(9), Spetember 2007.
11. L. Guthier, S. Yoo, and A. Jerraya. Automatic generation and targeting of application specific operating systems and embedded systems software. In *Proceedings of the Design Automation and Test Conference in Europe*, pages 679–685, 2001.
12. F. Herrera, H. Posadas, P. Snchez, and E. Villar. Systematic embedded software generation from systemc. In *Proceedings of the Design Automation and Test Conference in Europe*, 2003.
13. H. Kopetz, R. Obermaisser, C. E. Salloum, and B. Huber. Automotive software development for a multi-core system-on-a-chip. In *SEAS '07: Proceedings of the 4th International Workshop on Software Engineering for Automotive Systems*, page 2, Washington, DC, USA, 2007. IEEE Computer Society.
14. 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), December 2005.
15. T. Makkelainen. Hds from system-house perspective. In *Hardware dependent Software Workshop at DAC*, 2007.
16. A. C. Nacul and T. Givargis. Lightweight multitasking support for embedded systems using the phantom serializing compiler. In *Proceedings of the Design Automation and Test Conference in Europe*, pages 742–747, 2005.
17. S. Pasricha, Y.-H. Park, F. J. Kurdahi, and N. Dutt. System-level power-performance tradeoffs in bus matrix communication architecture synthesis. In *CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, pages 300–305, New York, NY, USA, 2006. ACM.
18. A. Pinto, L. P. Carloni, and A. L. Sangiovanni-Vincentelli. Constraint-driven communication synthesis. In *Proceedings of the Design Automation Conference*, pages 783–788, 2002.
19. S. Ritz and et al. High-level software synthesis for the design of communication systems. *IEEE Journal on Selected Areas in Communications*, April 1993.
20. K. K. Ryu and V. Mooney. Automated bus generation for multiprocessor soc design. In *Proceedings of the Design Automation and Test Conference in Europe*, page 10282, 2003.
21. A. Sangiovanni-Vincentelli and et al. A next-generation design framework for platform-based design. In *Conference on Using Hardware Design and Verification Languages (DVCon)*, February 2007.
22. G. Schirner, A. Gerstlauer, and R. Dömer. Automatic generation of hardware dependent software for mpsocs from abstract system specifications. In *Proceedings of the Asia-Pacific Design Automation Conference*, pages 271–276, 2008.
23. H. Yu, R. Dömer, and D. Gajski. Embedded software generation from system level design languages. In *Proceedings of the Asia-Pacific Design Automation Conference*, pages 463–468, 2004.