

Model Based Synthesis of Embedded Software

Samar Abdi, Daniel D. Gajski, 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 multicore embedded systems. We propose a classification of multicore embedded systems based on their platform architecture. We identify key design decisions and models that are required for embedded system synthesis. We present a model based design methodology 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 multi-core MP3 decoder and JPEG encoder designs, 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. Over 3X productivity gain in overall multicore design is shown to result from automatic SW synthesis.

Index Terms—system modeling, communication SW synthesis, MPSoC, model based design, transaction level modeling, pin and cycle accurate implementation

I. 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 [1], 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.

The rest of the paper is organized as follows. We present an overview of current state of the art in SW synthesis in Section II. In Section III, we propose a classification of multicore embedded platforms. The design decisions and models needed to implement applications on such platforms are discussed in Section IV. We then delve into model semantics for SW synthesis in Section V. Automatic SW generation procedures, based on the semantics, are presented in Section VI. Experimental results on synthesis quality and design productivity are presented in Section VII, followed by conclusions and a future outlook for model based SW synthesis.

II. 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 [3] provide common API and middleware for automotive SW development. On the other hand, system level design languages such as SystemC [4] and SpecC [5] 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 [6] (Co-Design Finite State Machine), DESCARTES [7] (ADF and an extended SDF), Cortadella [8] (petri nets) and SCE [9] (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 [10] proposes component model for dependable automotive systems. Sangiovanni-vincentelli [11] 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 [12]–[14] 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) [16], [17] 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. Herrera [18] 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 [19] proposes generation of source code from SystemC mapped onto an RTOS, while Gauthier’s method [20] provides generation of application-specific RTOS and the corresponding application SW. Both techniques cannot be extended to multi-core platforms with inter-core communication synthesis. Yu [21] shows generation of application C code from concurrent SpecC, which requires the initial system modeling to be done in SpecC. The Phantom Serializing Compiler [22] translates multi-tasking POSIX C code input into sequential C code by custom scheduling, but is a purely SW core-specific optimization. Schirner [23] 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 a C/graphical specification.

III. MULTICORE PLATFORM CLASSIFICATION

Multicore platforms may be classified broadly based on the type of cores used and their connectivity. Typical multicore architectures are asymmetric, symmetric and heterogeneous networks. The type of platform chosen to implement a design depends on the application characteristics. In this section, we will look at the properties of each type of platform and the rationale behind selecting the platform architecture. Finally, we will present a generic

platform template that covers all types of platforms and simplifies automatic SW synthesis.

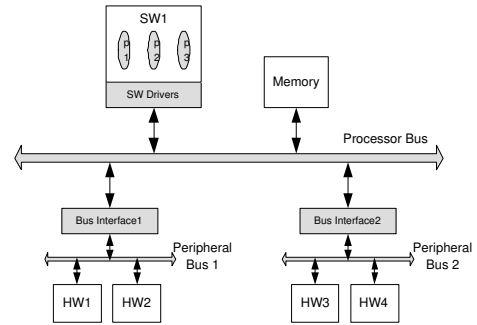


Figure 1. An asymmetric multicore platform.

A. Asymmetric Multicore Platform

Asymmetric multicore platforms are typically characterized by a single embedded processor with several hardware peripherals, as shown in Figure 1. The hardware peripherals are optimized for a given function that is executed frequently in the application. Alternately, a compute intensive function may be mapped to HW, if the SW core data path is inefficient for the function. The communication topology of the asymmetric platform depends on the interface of the hardware peripherals. Often, the hardware IP has a standard I/O interface that may not be compatible with the given processor bus protocol. In such cases, a bus interface component must be implemented to bridge the SW processor bus with the HW bus. This interface is responsible for protocol conversion and buffering of I/O data for the HW components.

Ideal application candidates are sequential applications that do not exhibit any parallelism or cannot be easily pipelined. Typically, such applications have a few frequently executed functions with low I/O. The discrete cosine transform DCT function used in various multimedia applications is one such example. It must be noted that the I/O to the peripheral must be minimal, so that the communication delays do not offset the performance gain from the HW acceleration.

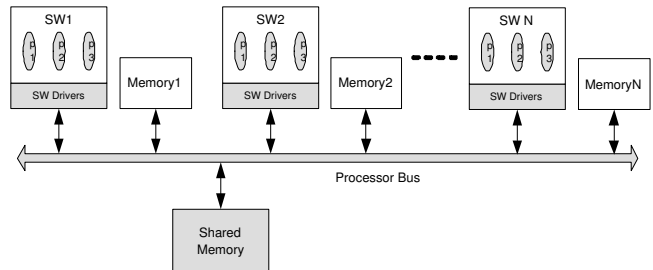


Figure 2. A symmetric multicore platform.

B. Symmetric Multicore Platform

Symmetric multicore platforms are characterized by several SW processor cores connected to a shared bus, as shown in Figure 2. Each core is of the same type. The processors communicate via the shared bus and a

common shared memory. Typically, such cores connect to the bus as *masters*, which restricts direct communication. Therefore the shared memory acts as a slave that buffers the inter-core transactions. Additional control must be added either in SW or in HW (in the shared memory) to implement synchronization between the cores.

Ideal applications for symmetric platforms have a high degree of concurrency, so that the concurrent functions may be implemented as processes on different cores. The concurrency may be either in control flow or data flow. Pipelined applications with evenly balanced stages are also ideal for symmetric multicore implementation. However, it must be noted that the shared bus may be a bottleneck if more than two cores are used. This is particularly relevant for pipelined applications. Different stages of the pipeline on different cores may attempt to access the shared bus at the same time. This may overload the bus and the time lost in arbitration may offset the performance gains from concurrency.

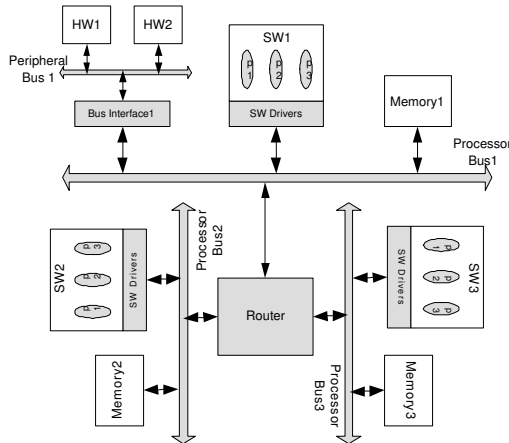


Figure 3. A heterogeneous multicore network.

C. Heterogeneous Multicore Network

The shared bus architecture of symmetric multicore platforms inhibits concurrent communication between different pairs of cores. Heterogeneous networks combine the benefits of symmetric and asymmetric platforms with the added benefit of efficient communication architecture. The platforms are characterized by several SW and HW cores, connected with independent compatible buses, as shown in Figure 3. The buses in turn are connected to routers or bus interfaces to provide protocol conversion and transaction routing, as applicable.

The typical applications suitable for such platforms are ones that have large amount of available parallelism as well as frequently executed functions. However, if the communication between the concurrent processes is low, a heterogeneous network may be an overkill due to the large area associated with an on-chip router.

D. Multicore Platform Template

It can be seen that different platform configurations require different types of services to satisfy the abstract

communication requirements of the application. Symmetric platforms require basic synchronization and memory access. Asymmetric platforms may require protocol conversion, which heterogeneous networks may require packetization and routing. In order to automate the synthesis of embedded SW on individual cores, we first need to define templates and semantics of the platform components.

We define a generic platform as a composition 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) buffering, (2) protocol conversion and (3) 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:

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

Abstract communication channels, between application processes, 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.

IV. SYSTEM-LEVEL DESIGN DECISIONS

There are several design steps involved in implementing a given application on a multicore platform. Each of these steps involves a design decision. These design decisions are used to configure the SW/HW platform, map the application to the platform and generate the SW and HW code needed for implementation. Design decisions must be made at various levels of abstraction, including physical-level, gate-level, RTL and system-level. In the context of embedded SW synthesis, we are primarily concerned with system-level design decisions.

Exploration of system level design choices is usually constrained by a number of factors, such as design methodology, legacy code, tool availability and project deadlines. Similarly, the order of design decisions depends

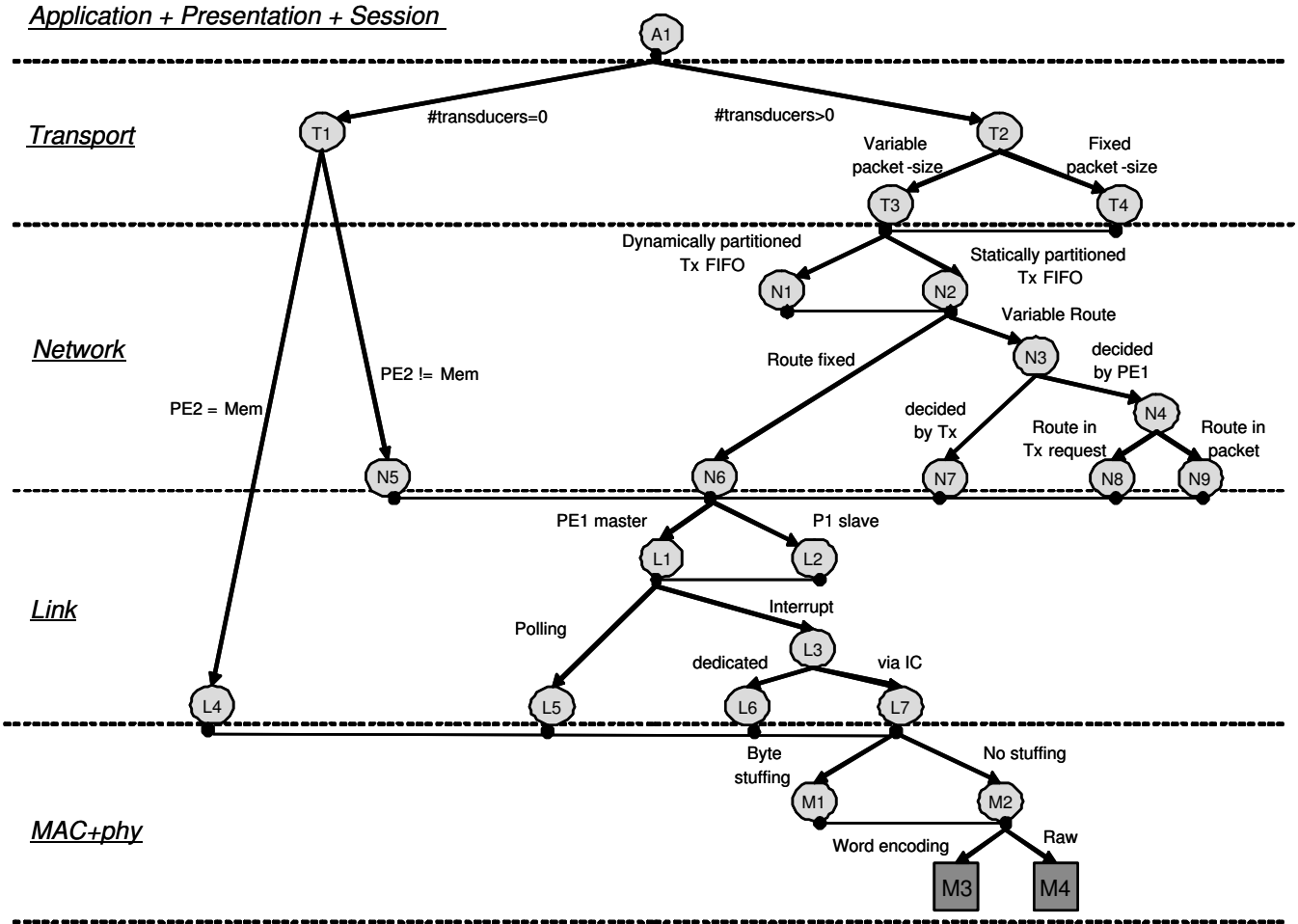


Figure 4. A system-level design decision tree.

on the above constraints. However, in any model based design methodology, each design decision transforms an intermediate model to reflect the affects of the decision. Therefore, the set of system-level design decisions refines the application model into a pin-cycle accurate model (PCAM). Embedded SW generation is part of this model refinement process.

The system level design decision tree in Figure 4 illustrates embedded SW synthesis in context of the ESE system level design methodology. The design decisions presented here are used to implement an abstract channel between two processes mapped to cores *PE1* and *PE2* in the platform. The decisions are required to generate the of the embedded SW code on the communicating cores. The nodes in the tree represent models in the design methodology. Each *edge* in the tree is labeled with the decision, used to transform the *source* model into the *destination* model. The starting point is the application model *A1*. These decisions may be grouped into relevant *OSI network layers* as shown.

It must be noted that the decisions shown in Figure 4 are only a subset of all decisions in ESE. Furthermore, not all the models in the tree are generated by ESE, as we shall see later. Generating and simulating unique models for every decision is impractical for a large multicore

design. Instead, TLM is the only intermediate model generated by ESE. The TLM incorporates all the decisions above and including the network layer.

A. Transport Layer Decisions

The objective of the transport layer is to determine the packetization of the data being transacted between the cores. The packet size is dependent on the route that this transaction may take. If the cores have a direct connection, and may access each other's local memory, then no transducers are needed. Therefore, packet size becomes irrelevant. The entire transaction is treated as a single packet in the resulting model *T1*. If transducers exist in the route, then the message must be split into packets of size less than the buffer size of all the transducers in the route. If the packet size is variable, then the code to determine it at run time must be generated in model *T4*. The transport layer decisions result in *T1*, *T3* or *T4*.

B. Network Layer Decisions

The network layer decisions implement all the routing-specific model transformations on the transport layer models. If the buffer of the first transducer (*Tx*) in the route is dynamically partitioned, then SW code must be generated in *PE1* to allocate the required packet size in

T_x . If the packet route is variable and determined by $PE1$, then additional SW must be generated to implement the routing algorithm in $PE1$. The route must be encoded in the packet or the T_x request buffer depending on T_x configuration. For fixed size packets and a fixed route, the only code needed is the T_x request. The models at the end of network layer decisions ($N5$ through $N9$) carry the SW for packetization and routing.

C. Link Layer Decisions

The network layer models are further refined by the link layer decisions that implement synchronization between $PE1$ and $T_x/PE2$. If $PE1$ is the master and the synchronization scheme is selected to be *Polling*, then code must be generated in $PE1$ to periodically check and reset the polling for each packet transfer. If interrupts are used for synchronization, then the respective handler code must be generated. If the transaction is a memory access, then no packet-level synchronization is required (model $L4$).

D. MAC and Physical Layer Decisions

The media access control (MAC) and physical layers are responsible for data transfer of packets on the bus. The packet bits are divided into bus words by the MAC layer decisions. If bit-stuffing is required to mark the packets, the code must be generated to add the marker bits. Any transformations for encoding the bus data, for example to implement error correction or detection, are performed based on the physical layer decisions. The final model at the end of MAC and physical layer decisions (represented by square nodes) carries all the system level design decisions for implementing the transaction from $PE1$ to $PE2$. This model is exported for implementation by traditional compilers and ASIC/FPGA synthesis tools.

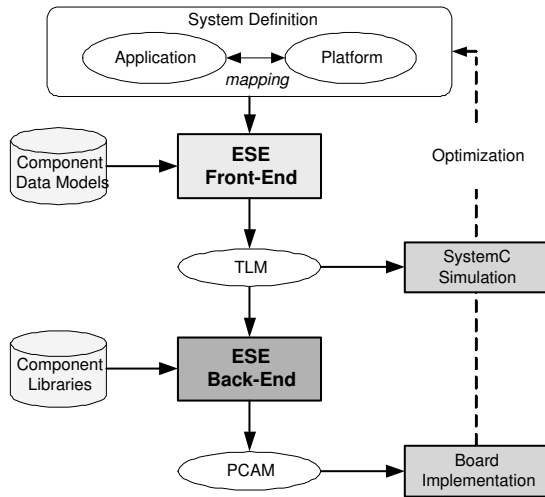


Figure 5. ESE Design Flow.

V. MODEL BASED DESIGN WITH ESE

The model based design methodology of ESE is shown in Figure 5. 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.

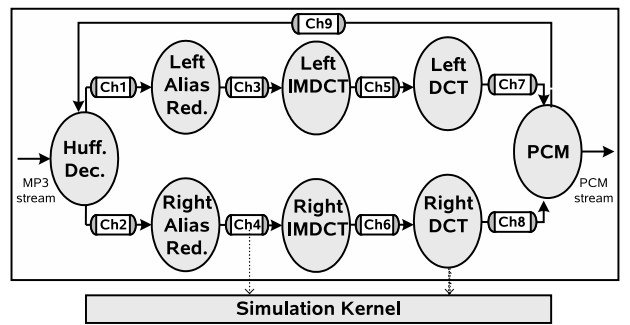


Figure 6. Application model.

A. Application Model

Figure 6. 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.

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* to *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 6. 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 $\langle i \rangle$ is the interface name:

- $\langle i \rangle_Send(void *data, int size)$ Synchronized send
- $\langle i \rangle_Recv(void *data, int size)$ Synch. receive
- $\langle i \rangle_Write(void *data, int size)$ Non-blocking write
- $\langle i \rangle_Read(void *data, int size)$ 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.

B. Transaction Level Model

The TLM is derived by mapping the application model in Section V-A 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

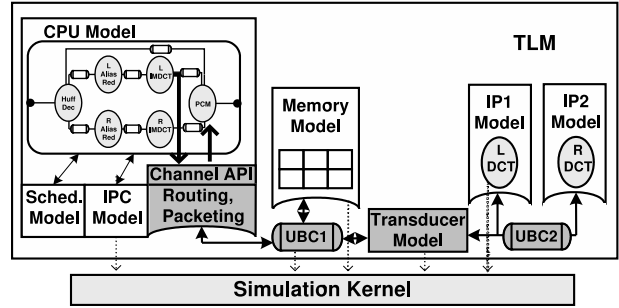


Figure 7. TLM resulting from application to platform mapping.

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 7 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, it has to be generated for every communication design change.

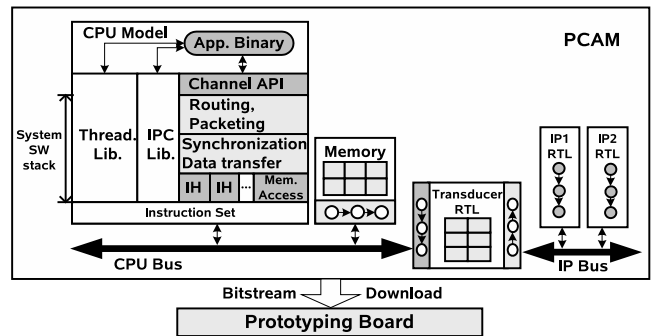


Figure 8. PCAM refined from TLM for board prototyping.

C. 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 development tools can then be used for real time debug.

Figure 8. 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 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, and networking.

The communication SW library consists of four layers as shown in Figure 8. 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.

VI. EMBEDDED SW GENERATION

In this section we describe the embedded SW synthesis and code generation from a set of design parameters. The parameters correspond to the design decision variables already discussed in Section IV. 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.

A. 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 III-D, 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.

B. 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:

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

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.

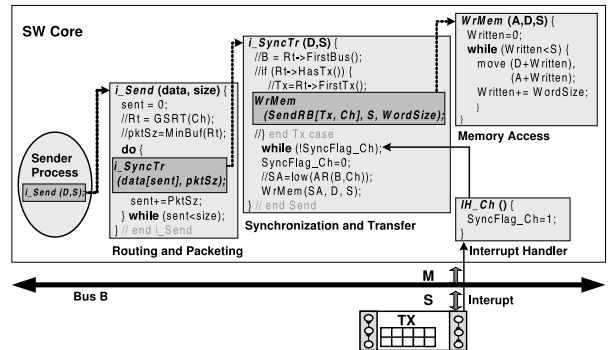


Figure 9. Embedded SW code example

C. Synchronization and Transfer

The routing of channel Ch determines the synchronization code generated inside the $i.SyncTr$ method. Given the

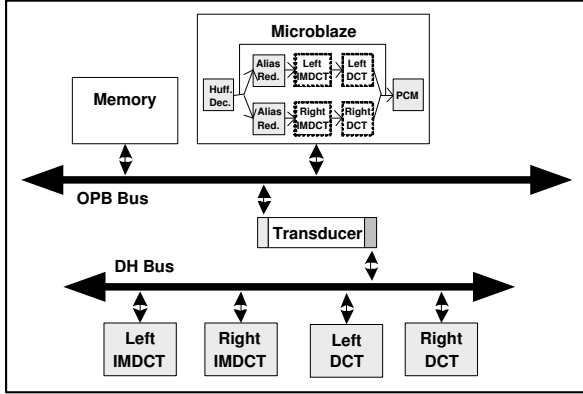


Figure 10. MP3 Decoder asymmetric platform: SW+2D+2I.

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 to 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 range $AR(B, Ch)$.

Figure 9 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 $SyncFlag_Ch$.

VII. EXPERIMENTAL RESULTS

In this section, we will present results on SW synthesis quality, productivity gain and design space exploration provided by ESE. We selected two applications; MP3 decoder and JPEG encoder to demonstrate SW synthesis for asymmetric, symmetric and heterogeneous network platforms. The PCAMs, carrying the embedded SW code, were implemented on the FF896 Virtex-II device using Xilinx EDK [16]. The performance of the synthesized designs was measured with an OPB timer on the board.

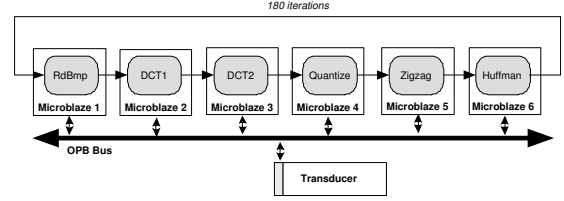


Figure 11. JPEG encoder 6-core symmetric platform

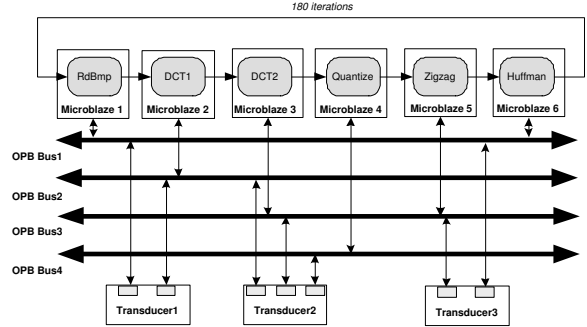


Figure 12. JPEG encoder 6-core network

A. Design drivers

Figure 10 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+1D*, *SW+2D*, *SW+2I* and *SW+2D+2I*, with parts of the application mapped to the hardware accelerators, as indicated by the mapping name. The inter-core bidirectional channels are routed over the OPB, DH buses and transducer Tx .

In order to demonstrate automatic system software synthesis for symmetric multicore platforms, we created several implementations of the JPEG encoder application. The application was mapped to a 6-core platform as shown in Figure 11. The key functions of the JPEG encoder are (*RdBmp*, *DCT*, *Quantize*, *Zigzag* and *Huffman*) are executed inside a loop. We created a pipelined model that reads the streaming bitmap image and encodes it frame by frame. Since DCT is twice as computationally intensive as the other functions, we split it into two functions, *DCT1* and *DCT2*, in order to balance the pipeline. We created several multicore JPEG implementations with *Microblaze* cores connected to a common OPB bus. The number of cores range from 2 to 6. The functions mapped to the same core were serialized into a single process.

In the pipelined JPEG implementation, majority of communication is attempted concurrently. Thus, the arbitration delay on the shared bus began to dominate the overall execution time. In order to optimize the communi-

	Design	SW (LOC)	Comm. cycles
Manually implemented SW	SW+1D	162	0.96M
	SW+2D	192	1.91M
	SW+2I	192	1.87M
	SW+2D+2I	252	3.79M
Automatically synthesized SW	SW+1D	168	0.95M (-0.74%)
	SW+2D	208	1.89M (-0.74%)
	SW+2I	208	1.86M (-0.62%)
	SW+2D+2I	288	3.76M (-0.68%)

TABLE I.
COMPARISON OF MANUAL VS. SYNTHESIZED MP3 DECODER SW

ation, we created a multicore network by adding 3 more *OPB* buses and 2 more transducers as shown in Figure 12.

B. Software Synthesis Quality

Table I 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 code is only marginally larger than manual implementation (between 6-36 LOC). The code size was different because the manual implementation shared the synchronization function for different application channels, while the synthesized code had unique synchronization function for each channel. However, the performance of the synthesized code, as measured by the on-chip timer, is marginally better ($\pm 1\%$) than manual implementation. The performance difference is seen because of the difference in synchronization implementation. The manual code had fewer total instructions, but incurred more instruction fetches for each communication call at run-time, resulting in slightly higher communication delay.

C. Productivity Gain

The most significant benefit of automatic SW synthesis is the huge productivity gain without loss in design quality. Table II shows the productivity gain metric for various multicore implementations of the MP3 and JPEG applications. The *Platform development time* (P) is the total time used in defining the multicore platform and mapping the application in ESE. This time includes creating the ESE project, partitioning the application and entering the design decisions. For the MP3 application, there was not much difference in isolating the DCT and IMDCT processes and instantiating the HW accelerators. However, for the JPEG example, it took significant time in pipelining the application and creating channels for communication between the pipeline stages. Furthermore, it took significant time in defining the routing and addressing of channels for the heterogeneous network implementation of JPEG (*6-core-net*).

The generated SW code size in lines of code (LOC) are also shown in Table II. The manual implementation of embedded SW for MP3 was done by a embedded design expert and took between 2 to 3.5 hours. Manual implementation of JPEG designs was not done due to resource constraints. However, based on our experience

Design	Slices	BRAMs	Cycles
2-core	1328	64	9.56M
2-core	4310	72	5.55M
3-core	5211	104	5.53M
4-core	6133	136	4.24M
5-core	7126	148	4.63M
6-core	8079	176	5.50M
6-core-net	13123	188	2.73M

TABLE III.
JPEG ENCODER DESIGN EXPLORATION

with MP3 SW design, we were able to predict that an expert embedded SW developer could code and validate approximately 80 LOC/hour. Using this metric, we estimated the time it would take a manual developer to implement the embedded SW for JPEG designs (M). The automatic SW synthesis in ESE tool less than 1 second for all the designs, due to the well defined semantics of our models. Since the code generation time is negligible, the productivity gain to be $(P + M)/P$. The results show that as the platform and mapping become more complex, the SW development time begins to dominate the platform design time. As a results we saw productivity gain ranging from 1.28X for simple asymmetric designs to over 4.7X for complex heterogeneous networks.

D. JPEG Design Space Exploration

We performed design space exploration to optimally implement the JPEG encoder application. As presented earlier, we created various multicore platform configurations. These design projects were developed in ESE and automatic SW synthesis was used to generate the code. Table III shows the FPGA area (in terms of slices and BRAMs) and the encoding time (in millions of cycles) for the various designs. We were able to conclude two suitable design choices as highlighted in the table. The 4-core design was found to be optimal for a symmetric multicore implementation. The network implementation used much higher area but provided the benefits of pipelining and concurrent communication. From Table II, we can see that the productivity gain for JPEG exploration, from SW synthesis alone, is 3.4X. The productivity is expected to be even higher for more complex designs.

VIII. CONCLUSIONS

We presented a model based technique and methodology for synthesis of embedded SW for heterogeneous multicore 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 and JPEG encoder applications

	Design	SW code size (LOC)	Platform dev. time (hrs.)[P]	Manual SW dev. time (hrs.)[M]	Productivity [Gain = (P+M)/P]
Asymmetric MP3 decoder	SW+1D	162	5	2	1.28 X
	SW+2D	192	5	2.5	1.33 X
	SW+2I	192	5	2.5	1.33 X
	SW+2D+2I	252	5	3.5	1.37 X
Symmetric Shared-bus JPEG Encoder	2-core	408	3	5	2.67 X
	3-core	490	3.5	6	2.71 X
	4-core	772	4	10	3.5 X
	5-core	1020	5	12	3.4 X
	6-core	1224	6	15	3.5 X
JPEG Network	6-core-net	2093	7	26	4.71 X

TABLE II.
PRODUCTIVITY GAIN FROM AUTOMATIC SW SYNTHESIS

demonstrated the applicability of our technique to large industrial embedded systems. Our automatic embedded SW synthesis reduces overall design time, without loss of design quality compared to 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.

ACKNOWLEDGMENT

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 Chandrariah for the C reference of the MP3 Decoder, and Gunar Schirner for discussions on hardware dependent software.

REFERENCES

- [1] "Embedded System Environment[online]. Available: <http://www.cecs.uci.edu/~ese/>."
- [2] "Automotive Open System Architecture[online]. Available: <http://www.autosar.org/>."
- [3] "OSEK[online]. Available: <http://www.osek-vdx.org/>."
- [4] "SystemC, OSCI[online]. Available: <http://www.systemc.org/>."
- [5] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, January 2000.
- [6] F. Balarin and et al., *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*. Kluwer, 1997.
- [7] S. Ritz and et al., "High-level software synthesis for the design of communication systems," *IEEE Journal on Selected Areas in Communications*, April 1993.
- [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] A. Gerstlauer, D. Shin, J. Peng, R. Dömer, 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*, vol. 26, no. 9, Spetember 2007.
- [10] 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*. Washington, DC, USA: IEEE Computer Society, 2007, p. 2.
- [11] 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.
- [12] A. Pinto, L. P. Carloni, and A. L. Sangiovanni-Vincentelli, "Constraint-driven communication synthesis," in *Proceedings of the Design Automation Conference*, 2002, pp. 783–788.
- [13] K. K. Ryu and V. Mooney, "Automated bus generation for multiprocessor soc design," in *Proceedings of the Design Automation and Test Conference in Europe*, 2003, p. 10282.
- [14] S. Pasricha, Y.-H. Park, F. J. Kurdahi, and N. Dutt, "System-level power-performance trade-offs in bus matrix communication architecture synthesis," in *CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*. New York, NY, USA: ACM, 2006, pp. 300–305.
- [15] T. Makkelaianen, "Hds from system-house perspective," in *Hardware dependent Software Workshop at DAC*, 2007.
- [16] "Xilinx Embedded Development Kit[online]. Available: <http://www.xilinx.com/>."
- [17] "Altera SOPC Builder[online]. Available: <http://www.altera.com/>."
- [18] 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.
- [19] 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*, vol. 10, no. 4, December 2005.
- [20] 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*, 2001, pp. 679–685.
- [21] H. Yu, R. Dömer, and D. Gajski, "Embedded software generation from system level design languages," in *Proceedings of the Asia-Pacific Design Automation Conference*, 2004, pp. 463–468.
- [22] 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*, 2005, pp. 742–747.
- [23] 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*, 2008, pp. 271–276.