Componentizing Hardware/Software Interface Design

Kecheng Hao and Fei Xie
Department of Computer Science, Portland State University, Portland, OR 97207
{kecheng, xie}@cs.pdx.edu

***Abstract*—Building highly optimized embedded systems demands hardware/software (HW/SW) co-design. A key challenge in co-design is the design of HW/SW interfaces, which is often a design bottleneck. We propose a novel approach to HW/SW interface design based on the concept of bridge component. Bridge components fill the HW/SW semantic gap by propagating events across the HW/SW boundary and raise the abstraction level for designing HW/SW interfaces by abstracting processors, buses, embedded OS, etc. of embedded system platforms. Bridge components are specified in platform-specific Bridge Specification Languages (BSLs) and compiled by the BSL compilers for simulation and deployment. We have applied our approach to two different embedded system platforms. Case studies have shown that bridge components greatly simplify component-based co-design of embedded systems and system simulation speed can be improved three orders of magnitude by simulating bridge components on the transaction level.**

I. Introduction

Embedded systems are required to be high-performance and low-cost. They often have stringent constraints in power consumption, memory usage, real-timeness, etc. [1], which require maximally exploiting the resources available. Traditional design methods, developing hardware and software components separately, are insufficient to meet such requirement. This generates great demands for hardware/software (HW/SW) co-design and, to evaluate the design, co-simulation. HW/SW interface design is a central task for co-design. It is usually considered as an error-prone and time-consuming task, since the designer needs to consider not only how to propagate events across the HW/SW semantic boundary, but also the effect of the embedded system platform.

We propose a novel approach to HW/SW interface design based on the concept of bridge component, which simplifies co-design and accelerates the speed of co-simulation. HW/SW interfaces are componentized into bridge components. Bridge components raise the level of abstraction for designing HW/SW interfaces and propagate events across the HW/SW boundary. As a system is designed, hardware, software, and bridge components are treated as components on the same level. Since our approach is component-based, the systems designed following our approach will have two advantages: (1) The systems are highly flexible. Any components can be substituted by new components, even the HW/SW interface. This means the system can be easily ported onto a new platform. (2) The HW/SW interfaces are highly reusable. Any new system based on the same platform can reuse the existing bridge components. Bridge components are configurable abstractions of the platform. The designer can configure the platform via bridge components to satisfy design requirements. Bridge components are specified in a platform-specific Bridge Specification Language (BSL) and compiled for co-simulation and deployment by the BSL compiler. To accelerate the speed of co-simulation, bridge components can be simulated on the transaction level.

We have realized our component-based approach to HW/SW interface design on two different embedded system platforms, the Mica [2] platform for networked sensors and the Microsoft Invisible Computing (MIC) [3] platform. We have developed the BSLs for both platforms and experimented our approach on a family of networked sensor systems. The case study has shown that our approach simplifies HW/SW interface design and significantly speeds up system simulation by simulating bridge components on the transaction level.

The main contribution of this work is the concept of componentizing HW/SW interfaces through bridge components. The key advantages of our approach include:

- HW/SW interface design is simplified. Component-based HW/SW interfaces are flexible and reusable.
- The bridge components provide a focal point for configuring the embedded system platform.
- Systems designed following our approach are highly portable by replacing the bridge components.
- Two complementary simulation levels are provided for different simulation purposes. The systems are compiled for deployment directly without extra manual efforts.

The remainder of this paper is organized as follows. In Section II, we provide the background of HW/SW co-design and discuss related work. In section III, we introduce a unified component model for embedded systems, which provides the component-based context for the bridge component concept. In section IV, we discuss how to specify bridge components using BSL. In section V, we introduce how bridge components facilitate co-simulation. In section VI, we discuss a case study and present the experimental results. In section VII, we conclude this paper and discuss future work.

II. Background and Related Work

HW/SW co-design aims at satisfying system-level design objectives by exploiting the synergism of hardware and software through their concurrent design [4]. There has been much research on co-design [5], [6]. In [7], an approach to HW/SW interface generation based on a parametrized library was

978-3-9810801-5-5/DATE09 © 2009 EDAA

proposed. To realize the communication between hardware and software at the hardware level, a software process is replaced by a new process that is behaviorally equivalent, however, is specified in Hardware Description Languages (HDLs). This approach has been implemented in the CoWare [8] environment. In [9], Hardware Procedure Calls (HPCs) were proposed to abstract the platform details of transaction-level communication by providing a flexible middleware for modeling embedded software on top of transaction-level models.

Transaction-level modeling with a general purpose system-level design language such as SystemC is seen as the state-of-the-art of embedded system co-design [10]. Designing interfaces using SystemC needs manual efforts converting SystemC design into compilable software code and synthesizable hardware design. In our approach, software code and hardware design can be generated from bridge components automatically. Therefore, the system is readily compilable for simulation and deployment.

Ptolemy [11] is a framework for simulating and prototyping heterogeneous systems, currently available as Ptolemy II [12]. Ptolemy focuses on component-based heterogeneous modeling. It uses tokens as the underlying communication mechanism. Directors regulate how actors in the design behave and how tokens are used to communicate between them. Models of computation for various domains have been realized based on this semantics. Our approach does not require a particular underlying semantics and allows hardware and software components be designed in their native semantics.

III. Unified Component Model for Embedded Systems

Before we discuss how we componentize HW/SW interface design, we first provide the context for bridge components, a unified component model for embedded systems [13]. This model as shown in Figure 1 unifies hardware and software component models. An embedded system is assembled from

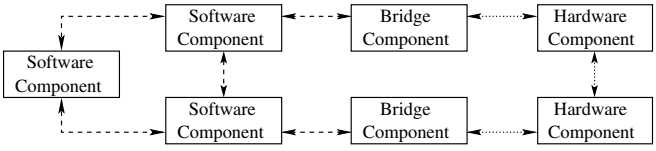

Fig. 1. Unified Component Model

components. There are three types of primitive components: *software components*, *hardware components*, and *bridge components*. Bridge components interact with hardware and software components and fill the semantic gap between hardware and software components by propagating events across the HW/SW semantic boundary. Three types of composite components may be defined: *software components*, *hardware components* and *hybrid components*. Sub-components of a composite software component are all software components and sub-components of a composite hardware component are all hardware component. A hybrid component contains hardware, software, and therefore bridge sub-components.

Components. A component M is a triple (E, I, P) where E is the design or implementation of M, I is an interface including the semantic entities for M to interact with its environment, and P is a set of temporal properties that are defined on I and have been verified on E. For a software component, E can be specified in C or other software design/programming languages and I can be a function based interface or other type of software interfaces. For a hardware component, E can be specified in Verilog or other HDLs and I can be a signal based interface. For a bridge component, E is specified in a platform-specific BSL and I is a dual interface: a hardware interface and a software interface (see Section IV). For specification of P, we refer the reader to [14].

Ports. We have refined the component interface specification with the port concept. A port groups events (e.g., functions and signals), which together realize a certain functionality. Depending on whether a component is providing or utilizing the functionality, the port can be a "provides" or "uses" port in a component interface. Each event in a port has an input or output direction. Whether an event in a port is an input or output of a component also depends on whether the port is provided or used. If a component provides a port, its events conform to the directions as specified in the port; otherwise, its events reverse the directions.

IV. Bridge Specification Language

HW/SW interface design is usually considered the most difficult and time-consuming task of co-design. To design the HW/SW interface, the designer needs to consider not only the event conversion between hardware and software, but also the embedded system platform including the processors, the buses, the embedded OS, etc. Our approach componentizes the HW/SW interface and abstracts the platform through bridge components, with which the designer can easily configure the platform and specify the event conversion between hardware and software. Another advantage of our approach is that it builds upon component-based development. The designer can easily reuse bridge components when building a new system on the same platform. We specify bridge components using a Bridge Specification Language (BSL). A bridge component specification includes: (1) dual interface specification; (2) bridge transactor specification; and (3) platform configuration. BSLs are platform-specific. We have developed the BSLs for the Mica platform [2] for networked sensors and the Microsoft Invisible Computing (MIC) [3] platform.

A. Dual Interface

Since a bridge component connects hardware and software components, it has dual interfaces: a hardware interface and a software interface. Figure 2 shows the interfaces of the bridge component of a networked sensor system.

We design the bridge component interfaces based on ports. Hardware ports follow the hardware semantics. A hardware port includes a list of signals. Software ports follow the software semantics. A software port is a group of functions that together provide a certain service. If the port is provided

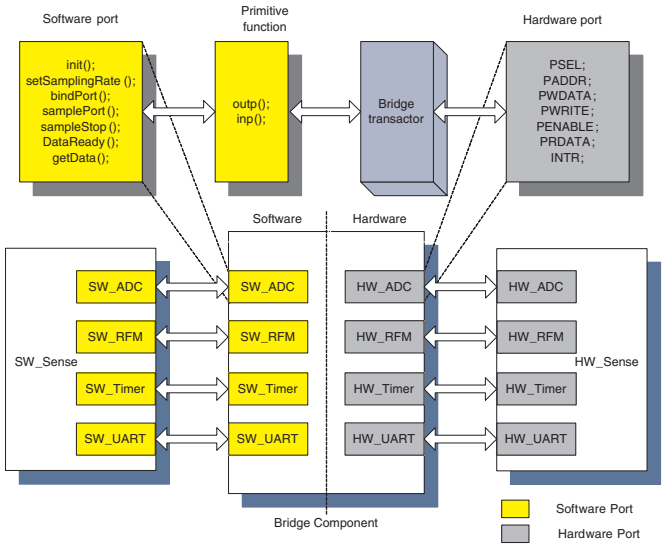

Fig. 2. Dual Interfaces of Bridge Component

```
Transactor {
  void outp(uint_8 val, uint_8 address) {
        PSEL = 1;
        PADDR = address;
        PWDATA = val;
        PWRITE = 1;
        @ (posedge PCLK);
        PENABLE = 1;
        @ (posedge PCLK);
        PSEL = 0;
        PENABLE = 0;
  }
  uint_8 inp(uint_8 address) {
        PSEL = 1;
        PADDR = address;
        PWRITE = 0;
        @ (posedge PCLK);
        PENABLE = 1;
        @ (posedge PCLK);
        PSEL = 0;
        PENABLE = 0;
      return PRDATA;
  }

  HW_Timer.INTR ⇒ SW_Timer.Fire()   1;
  HW_ADC.INTR  ⇒ SW_ADC.DataReady()   1;
}
```

Fig. 3. Transactors and Interrupt Mappings

by the bridge component, the implementation of the port is defined in the bridge component. If the bridge component uses a port, the bridge component can call the functions in the port directly. The implementation of the software port of the bridge component is specified in the programming language for the software components, e.g., nesC [15] for the Mica platform and C for the MIC platform. The interaction with hardware is encapsulated in *primitive functions*. The primitive functions are invisible to software components and can only be called within the bridge components. The primitive functions are defined in the BSL transactors (see below).

B. HW/SW Event Conversion

A key challenge in HW/SW interface design is how to fill the semantic gap between hardware and software. We employ *transactors* to propagate events across the HW/SW boundary. Our transactor concept has a similar, but different meaning compared with the transactor concept in SystemC. In SystemC, transactors inter-connect components on different levels such as Transaction Level (TL) and Register Transfer Level (RTL). In our approach, transactors inter-connect hardware and software components that are developed in different languages and follow different semantics.

Figure 3 illustrates the transactors of the networked sensor system. How these transactors inter-connect the hardware and software components in this system is shown in Figure 2. A transaction can be invoked from either the software side or the hardware side. A transaction invoked by a software event will generate a sequence of hardware signals. In the Mica and MIC platforms, the software components interact through function calls. The software components interact with the hardware devices by calling primitive functions. The primitive functions *outp* and *inp* are used to write/read data to/from the registers of the peripherals. These two functions are library functions from the AVR processor, which can be compiled by the AVR compiler *avr-gcc*. The peripherals are connected to the processor by the APB bus in the networked sensor system. therefore, the write/read operation follows the APB protocol. The transactors *outp* and *inp* have the same declarations as the processor library functions. The bodies of the transactors are specified in the HDL of the Mica platform, i.e., Verilog or VHDL. When these two transactors are called, a hardware APB write/read signal sequence is generated.

The transaction invoked from the hardware side is implemented through hardware interrupts. When the hardware generates an interrupt signal, an Interrupt Service Routine (ISR) is invoked to handle the interrupt. In the BSLs for the Mica and MIC platforms, the hardware interrupt signals are directly mapped to the ISRs by ⇒. The interrupt signals usually have different priorities. The designer can specify the interrupt priorities in the mapping. Figure 3 also illustrates the mapping from the interrupt signals to the ISRs. When the interrupt signal *HW_ADC.INTR* is generated, the ISR *SW_ADC.DataReady()* will be invoked to handle the ADC data. The number *1* indicates that this interrupt is at the highest priority level.

C. Platform Configuration via Bridge Components

We use bridge components to abstract an embedded system platform and allow the platform to be configured via bridge components. For instance, the designer can specify the device addresses on the system bus and the scheduling algorithms for the OS. Such information will affect the system execution, however, is invisible from the outside of the bridge components. Our approach allows the designer to configure the platform to satisfy the design requirement and the configuration can be changed easily. This improves the design flexibility.

Figure 4 shows the configuration of the networked sensor system based on the Mica platform. The system uses

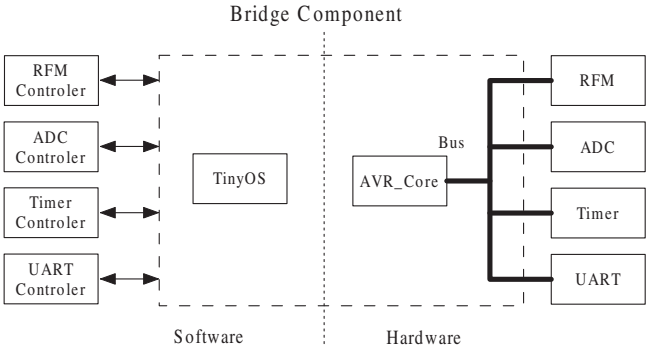

Fig. 4. Configuration of Networked Sensor System

TinyOS [16] as the embedded OS. Figure 5 shows the BSL specification of the configuration. Since the bridge components

```
Bridge Configuration {
  Hardware {
    CPU AVR { source{avr.v}; }
    BUS APB { source{apb.v}; master AVR ;
      slave ADC    (0x04,0x07); slave UART (0x09,0x0C);
      slave RFM    (0x0D,0x0F); slave Timer  (0x23,0x33);
    }
  }
  Software { SCHEDULER Sched { source{TinySchedulerC.nc}; } }
}
```

Fig. 5. BSL Specification of Sensor System Configuration

abstract the platform including its hardware and software, we use the keywords *Hardware* and *Software* to indict the hardware and software configurations, respectively. We use the keyword *source* to specify the source file for each platform component. The BSL compiler can retrieve the platform components according to the specification. For the hardware configuration, we have three types of hardware components: CPUs, buses, and devices, identified by the keywords *CPU*, *BUS*, and *DEVICE*. For CPUs and devices, the designer only needs to specify the source code path for these components. For buses, the designer needs to specify the bus masters and slaves, and the slave addresses. For the software configuration, currently we only provide keywords *SCHEDULER* and *HEAP* to define the scheduling and heap algorithms for the embedded OS. In the configuration shown in Figure 5, for the platform hardware, the APB bus is utilized to connect the AVR processor with the low-speed peripherals and no platform device is included. For the platform software, TinyOS is configured to use a priority-based scheduling algorithm.

V. Co-Simulation Using BSL

Selecting a platform is usually the first step for embedded system design, which will determine the execution environment of the systems. The designer selects the platform according to the system requirements. Each application domain commonly has some typical platforms that are designed to satisfy the constraints of this domain. A platform typically includes the processors, the buses, the memory models, the embedded OS, and furthermore a library of hardware and software components that have been developed on the platform and are available for reuse. Bridge components abstract the embedded system platform. The designer can configure the platform via the bridge components by specifying the bridge configuration in BSL. The BSL compiler compiles bridge components into their hardware and software implementations. For co-simulation, the BSL compiler configures the platform and sets up the co-simulation environment.

A. Co-Simulation Environment Setup

For different embedded system platforms, the ways to configure the co-simulators are often different. However, given a platform, there is much commonality in configuring the co-simulator for different systems based on this platform. Figure 6

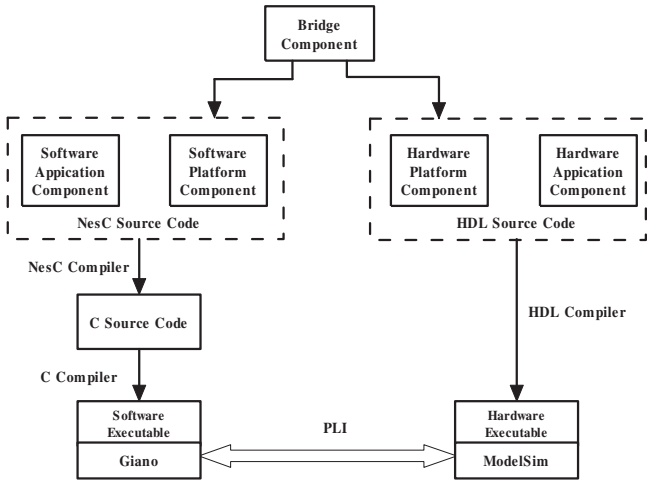

Fig. 6. Co-Simulation Environment Setup Flow

illustrates the co-simulation environment setup flow for the Mica platform. To set up the co-simulation environment, the BSL compiler retrieves the hardware platform components in Verilog, e.g., the processor and the bus, and the software platform components in nesC, e.g., the embedded OS. In the MIC platform, the software platform components are instead specified in C.

In our study, we employ ModelSim [17] and Giano [3] as the foundations for our system co-simulator. ModelSim is a hardware simulator that is capable of simulating hardware designs written in Hardware Description Languages (HDLs) such as Verilog, VHDL, and SystemC. Giano a is full-system real-time simulator. It incorporates simulation of processors, I/O sub-systems, and peripherals of a system. ModelSim can be attached to Giano and be responsible for simulation of reconfigurable FPGAs. The communication between hardware and software components is done through the Programming Language Interface (PLI) between Giano and ModelSim. The PLI is masked by and configured via the bridge components.

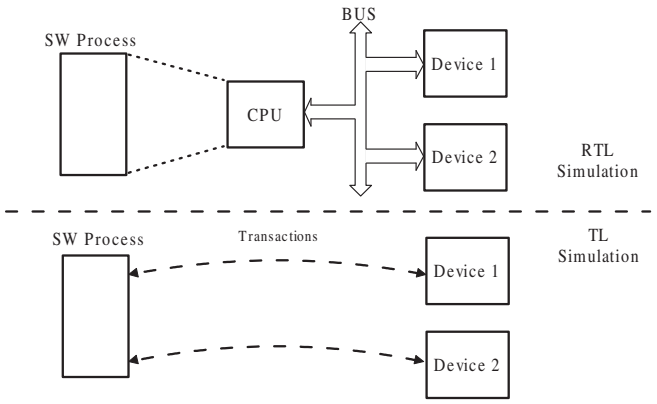

Fig. 7. Two Simulation Levels for Bridge Components

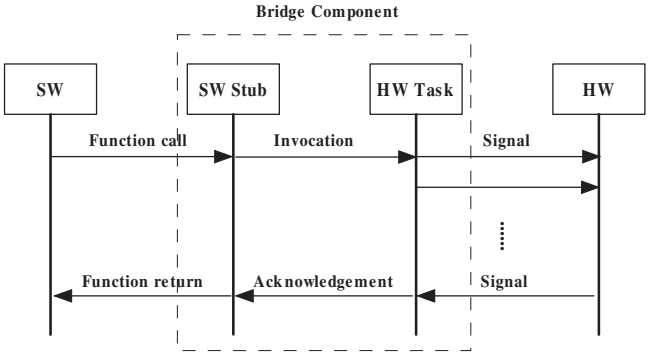

Fig. 8. Execution of a Transactor

B. Two Simulation Levels for Bridge Components

The bridge components can be simulated on two different levels: Register Transfer Level (RTL) and Transaction Level (TL), as shown in Figure 7. For the RTL simulation, the processor and the bus are included in the system co-simulation and they are configured according to the bridge components. The BSL compiler also generates software code from the transactors. In the software code generated, invocations of the primitive functions whose bodies are specified in HDLs are converted to invocations of the corresponding library functions of the processor. The software code is merged with the code of software components, which together are compiled by the native compiler for the processor to executables that can be loaded onto Giano. The hardware components are compiled by the HDL compiler into executables that can be loaded onto ModelSim. This mode of compilation is also used to generate the deployable of the system, instead of loading onto Giano and ModelSim, the software and hardware images are loaded into the memory and used to program FPGAs, respectively.

For the TL simulation, the platform components such as the processor, the bus, and the OS are excluded to accelerate the simulation speed. The hardware and software components are connected directly by the transactors, which convert between software events and hardware signals. The BSL compiler will compile each transactor into two parts: a software stub and a hardware task. The software stub has the same name as the software primitive function. The hardware task is generated from the transactor body which is specified in HDLs. Figure 8 illustrates the execution of a transactor. When a software component invokes a transactor to communicate with the hardware, the software stub of the transactor is invoked. The stub will then invoke the hardware task to generate the hardware signal sequence. After the hardware task is completed, the software stub returns.

Each transaction is an atomic operation and cannot be interrupted by any other event, even a hardware interrupt. One transaction cannot be executed twice at the same time. It follows the First-come, First-served (FCFS) policy. If two software components try to invoke the same transaction concurrently, the component that invokes first will win the race. The second component receives an invocation failure notice and can try again later. However, different transactions can be invoked concurrently by different components. For instance, one transaction samples the data from the hardware ADC component and another transaction receives data from the hardware network components. The concurrent execution of the hardware components depends on the platform.

VI. Case Study and Evaluation

We have instantiated our approach on two different platforms, the Mica platform and the MIC platform. To support HW/SW interface design on these two platforms, we have developed two platform-specific BSLs. Table I lists the platform components included in these two platforms. We have applied our approach to a family of networked sensor systems based on the Mica platform, which includes those from both the TinyOS [16] distribution and the CodeBlue [18] distribution and re-engineered this family of networked sensor systems following our component-based approach.

We have simulated this family of systems by simulating their bridge components on the two different levels, RTL and TL. Table II shows that the simulation can be sped up three orders of magnitude by simulating the bridge components on the TL level rather than the RTL level. The TL simulation would be infeasible if HW/SW interfaces were specified directly on the RTL level. Since hardware and software components are designed in their implementation languages, they can be synthesized or compiled for deployment without extra effort.

Compared to the existing co-design approaches and tools, our approach has three main advantages: (1) Our approach componentizes the HW/SW interface through bridge components, which provide a focal point for configuring the system platform and abstract away the details in platform configuration. Without this abstraction, the designer needs to fully understand the details about the whole platform, for instance, where is the scheduling algorithm of the embedded OS and how it can be changed. This is a very error-prone process and makes it challenging to work on an unfamiliar platform. In our approach, the designer does not need to go through these details and he/she only provides the system configuration

TABLE I
PLATFORM COMPONENTS

Platform	OS	Scheduler	Heap	SW Language	Compiler	CPU	BUS	HW Language
Mica	TinyOS	FCFS PriorityScheduling	N/A	nesC	avr-gcc msp430-gcc	AVR MSP430	APB I^2C	Verilog VHDL
MIC	MMLite	FCFS RoundRobin ConstraintScheduling	BitMapHeap FirstFitHeap	C	arm-gcc mips-gcc cl	ARM MIPS PowerPC	ASB APB AHB	Verilog VHDL

TABLE II
EXPERIMENTAL RESULTS

System	Simulation Speed			Lines of code		
	RTL (Sec)	TL (Sec)	Speed Up	BSL Size	Manual Work	Reduction
Blink	2.311	0.002	x1156	126	805	84.4%
BlinkTask	2.613	0.002	x1307	126	805	84.4%
BAPBase	1.355	0.001	x1355	490	1151	57.4%
CntToLed	5.547	0.002	x2774	126	805	84.4%
CntToRfm	5.336	0.002	x2668	412	1052	60.8%
GenericBase	1.12	0.007	x160	490	1151	57.4%
Oscilloscope	1.741	0.002	x871	382	1047	63.5%
RfmToLeds	1.542	0.001	x1542	490	1151	57.4%
Sense	1.644	0.004	x411	296	961	69.2%
SenseTask	1.887	0.004	x472	296	961	69.2%
SenseToLeds	1.587	0.003	x529	296	961	69.2%
SenseSounder	1.902	0.007	x272	324	1011	68.0%
SenseToRfm	6.837	0.008	x855	490	1151	57.4%
CodeBlue	7.456	0.011	x678	490	1151	57.4%

information via the bridge components. The platform can then be configured automatically. It can greatly shorten the system design process. (2) Our approach simplifies HW/SW interface design. Table II shows lines of code required to specify the HW/SW interfaces using BSL are less than half of these required if using hardware and software design/programming languages directly. To specify the interfaces directly, the designer needs to specify both hardware and software to build the communication channel. For co-simulation, the designer also needs to build PLI runtime functions to propagate events across the HW/SW boundary and the stub functions on both hardware and software sides to handle the event exchanges. This process is often time-consuming. In our approach, this process is significantly simplified. All the details such as PLI functions are hidden from the designer. He/she only needs to design the transactors. (3) The HW/SW interfaces designed following our approach are highly reusable and portable. The designer can easily reuse the bridge components when building a new system on the same platform or port an existing system to a new platform just by replacing the bridge components. In our experiment, the 14 systems reuse 6 bridge components, this can shorten the system development time, and these systems can be easily ported to the MIC platform. The case studies have shown that our approach leads to major simplification of co-design.

VII. CONCLUSION AND FUTURE WORK

We have presented a component-based approach to HW/SW interface design utilizing the concept of bridge component. Bridge components fill the HW/SW semantic gap and raise the level of abstraction for designing HW/SW interfaces. Bridge components are designed in a platform-specific BSL and compiled for simulation and deployment by the BSL compiler. We have successfully applied our approach to two different platforms, and re-engineered a family of networked sensor systems. The case study has shown that our approach can simplify the complexities of HW/SW interface design and accelerate the simulation speed by three orders of magnitude. Our future work is to integrate formal co-verification into our component-based approach to HW/SW interface design.

ACKNOWLEDGMENT

This research was partially supported by Semiconductor Research Corporation Contract #1356.001 and by National Science Foundation Grants #CNS-0613930 and #CNS-0720546.

REFERENCES

[1] D. Gajski, S. Narayan, F. Vahid, and J. Gong, *Specification and design of embedded systems*. Prentice-Hall, 1994.
[2] Crossbow, "Mica," http://www.xbow.com.
[3] A. Forin, B. Neekzad, and N. Lynch, "Giano: The two-headed system simulator," Microsoft Research, Tech. Rep. MSR-TR-2006-130, 2006.
[4] G. Micheli and R. Gupta, "Hardware/software co-design," *Proc. of IEEE (Special Issue)*, 1997.
[5] F. Balarin and et al., *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*. Kluwer, 1998.
[6] W. Wolf, "A decade of hardware/software codesign," *Computer*, vol. 36, no. 4, 2003.
[7] S. Vercauteren, J. V. D. Steen, and D. Berkest, "Combining software synthesis and hardware/software interface generation to meet hard real-time constraints," in *DATE*, 1999.
[8] CoWare, "http://www.coware.com."
[9] W. Klingauf, R. Günzel, and C. Schröder, "Embedded software development on top of transaction-level models," in *CODES+ISSS*, 2007.
[10] F. Ghenassia, *Transaction-Level Modeling with SystemC*. Springer, 2005.
[11] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogenous systems," *International Journal in Computer Simulation*, vol. 4, no. 2, 1994.
[12] Ptolemy Project, "Ptolemy II," http://ptolemy.berkeley.edu/ptolemyII.
[13] F. Xie, G. Yang, and X. Song, "Component-based hardware/software co-verification for building trustworthy embedded systems," *Journal of Systems and Software*, vol. 80, no. 5, 2007.
[14] F. Xie and H. Liu, "Unified property specification for hardware/software co-verification," in *COMPSAC*, 2007.
[15] D. Gay, P. Levis, R. Behren, M. Welsh, B. E., and D. Culler, "The nesc language: A holistic approach to networked embedded systems," in *PLDI*, 2003.
[16] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister, "System architecture directions for networked sensors," in *ASPLOS*, 2000.
[17] Mentor Graphics, "ModelSim," http://www.mentor.com.
[18] V. Shnayder, B. Chen, K. Lorincz, T. R. F. FulfordJones, and M. Welsh, "Sensor networks for medical care," Harvard University, Tech. Rep. TR-08-05, 2005.