Reusable and Architecture-neutral Virtual Communication Interface for Embedded Systems

Amjad Mohsen
Fraunhofer Institute for Integrated Circuits IIS
Nordostpark 93, 90411 Nuernberg, Germany
amjad.mohsen@iis.fraunhofer.de

Jochen Brandt
Fraunhofer Institute for Integrated Circuits IIS
Nordostpark 93, 90411 Nuernberg, Germany
jochen.brandt@iis.fraunhofer.de

ABSTRACT
In this paper, a functional model of a virtual communication interface (VCI) for embedded systems is presented. It hides the underlying communication technology from the application. The communication bus can then be changed without worrying about its users and without the need to rewrite the application. The VCI is co-designed in hardware/software to improve performance considering architectural issues and the needs of the application. Communication buses widely used in embedded systems are used to test the applicability of this VCI. The experimental results demonstrate that with careful co-design, the VCI is not only reusable but also can remarkably improve the overall performance.

Categories and Subject Descriptors
C.3 [**Special-purpose and Application-based Systems**]: *Real-time and embedded systems.*

General Terms
Design, Performance, Measurement.

Keywords
Embedded systems, communication interfaces, hardware/software co-design, design reusability, API, optimization.

1. INTRODUCTION
To cope with the increasing complexity of embedded systems and the non-stopping pressure of time-to-market, the productivity of design engineers and systems programmers has to be improved remarkably. Design tools such as CatapultC are now available to help designers explore significant part of the design space at the electronic system level (ESL) [1]. When used properly, such tools could remarkably reduce the design cycle while considering design quality as well.

On the software side, object-oriented programming languages such as Java and C++ are developed to support, among other objectives, code reusability. Different from C++, languages such as Java support the "Write Once and Run Anywhere" (WORA) model. Programmers can develop an application in Java and then run it on any platform which provides an implementation for the Java Virtual Machine (JVM). In recent years, it is remarkably noticed that Java is invading embedded systems which may reduce the development time. Nevertheless, Java code may have lower performance than native code.

Especially in embedded systems, aspects related to communication should be considered by the application developer. In this case, applications continue to depend on the underlying communication protocol/bus even by using languages which support the WORA model. Java, for example, offers I/O streams to transfer and manipulate different types of data only. Nevertheless, no control command can be transferred and/or manipulated. A wide variety of communication buses are used especially in control/automation arenas such as I^2C, SPI and CAN [2,3,4]. Applications must always be modified corresponding to the adopted bus. A new common interface can hide communication details and extend the concept of modular design further.

In computer networks, abstracting network resources is quite common to access remote resources without worrying about differences in physical interfaces. For example, by using Jini from SUN, network resources can be abstracted [5]. Another machine independent network environment is provided by TINA [6]. However, these did not handle lower layer protocols and could not be used in embedded systems where simple and custom buses are usually used. Another abstraction model called a virtual bus (VBUS) was presented by Toshiaki et al. in [7]. The goal was to access a distributed resource independent of its location and access method. The defined application programming interfaces (API) to access the VBUS were implemented in software (C/C++). However, embedded systems communication protocols are usually simpler in architecture and service routines which differ from computer networks. Moreover, no hardware/software co-design approach was used to improve the performance of the VBUS. Another set of APIs is defined as part of the IEEE 1451 to permit the access of transducer data through a common set of interfaces whether the transducer is connected to systems or accessed through networks [8]. Nevertheless, these remain to some extent limited by the application area.

In this paper, a parameterized VCI is developed which addresses a wide variety of embedded communication buses. The application developer always calls the VCI using "standard" APIs and the way in which these are interpreted and mapped onto the physical

APRES'09, October 11, 2009, Grenoble, France.

bus is hidden and automatically done by a configuration tool. Because the VCI is usually implemented separately, further bus-specific optimizations can efficiently be applied. Besides the abstract model of the VCI, a hardware/software co-designed and co-optimized prototype is developed and tested using communication buses widely used in embedded systems. Up to our knowledge, this issue was not tackled concretely in embedded systems.

This paper divides into 4 sections: The next section overviews the abstract model of the VCI. Section 3 explains key implementation issues of the VCI. Selected experimental results are presented and discussed in section 4. Section 5 concludes the paper.

2. ABSTRACT MODEL OF THE VCI

The VCI provides an architecture-neutral hardware abstraction layer (HAL) to the application developer. The application developer uses a set of parameterized APIs in order to send and receive data as well as to issue abstract control commands. To do so, the application developer formulates a "communication form" (CF) independent of the underplaying communication bus. The CF undergoes integrated refinement in the VCI in order to generate bus-specific commands based on the formulated job. Once the CF is finished, the VCI sends an interrupt to the application with the necessary status information. Figure 1 shows the basic architecture of the VCI.

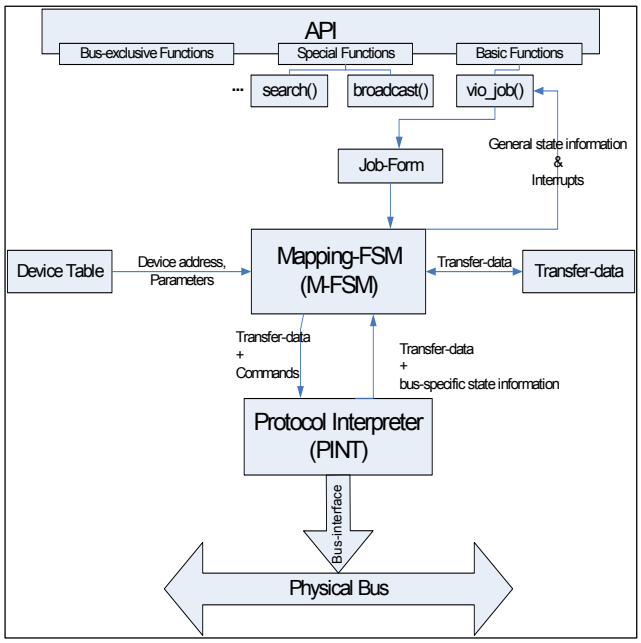

Figure 1. Basic architecture of the VCI

Physical interrupts and status signals sent by the physical bus are not forwarded directly to the application. Instead, these are received and processed by the VCI. The VCI maps the interrupts onto a set of abstract interrupt classes (AIC) which are "standard" to the application. The application always receives one of these AICs and responds independent of the underlying bus. The VCI then interprets the application response and generates a sequence of bus-specific commands.

The application developer creates a new CF which has a predefined format by providing abstract information on "what to do" not "how to do". The CF tells, for example, that the application needs to read/write from/to a device by providing a command and a device ID. Each device has a structure in memory to store needed information, such as device address and backed commands. A device table stores the addresses of all device-specific structures. To fill an entry in the device table, the device-specific configuration file is used. Based on the device ID, the device table is searched to find the address of the device-specific structure. To complete the CF, the application developer has also to provide a pointer to the first data in memory and how many bytes to exchange for read/write.

The CF is then processed by the VCI in two successive steps, namely mapping finite state machine (M-FSM) and protocol interpreter (PINT). The basic function of the M-FSM and PINT is to translate the architecture-neutral commands into architecture-specific commands and to guarantee correct operation of the bus. The efficiency of the whole VCI is directly dependent on the efficiency of the M-FSM and PINT. Therefore, the performance of applications which perform repeated I/O is directly influenced by how efficient the VCI is.

The M-FSM is a state machine that performs first stage processing on the CF and the corresponding information in the device table. It determines the architecture-specific commands fed to the PINT and receives/sends data from/to application. Data sent/received can be split or concatenated upon demand and bus width (w). To improve flexibility, the M-FSM is functionally partitioned into two layers: H- and L-layer[1]. The H-layer interacts with the application only and it is bus-neutral. Therefore, the H-layer is reusable. On the contrary, the L-layer is bus-specific. Figure 2 shows the structure of the M-FSM.

The H-layer specifies the sequence and conditions required according to the abstract commands given by the CF. For example, if CF tells that n bytes have to be written, the H-layer knows that n/w write operations are needed and the n bytes are available in memory address m. A sequence of abstract commands is then generated in the form: *config_bus (pars)*, *write (dev_address), write (word_1)…write (word_n)*. The L-layer interprets these into a sequence of bus-specific commands such as: *write register x, set flag y*. The PINT is directly controlled by the L-layer. The driver of the bus controller which is usually written in software can be replaced by the M-FSM. Moreover, a hardware/software co-design approach is used here which remarkably enhances the performance of the M-FSM over software drivers.

The PINT implements the bus-specific protocol of the bus. It determines when data/control can be sent and when to receive valid data. For example, assuming I^2C bus is used: The PINT sends the START bit, then the ADDRESS of the slave, then the internal register number; if any, then data (byte-wise), and then waits for the ACKNOWLEDGE bit sent by the slave. Having sent all data bytes, the PINT sends the STOP bit. The PINT then sends an interrupt to the M-FSM which maps it to one of the AIC and forwards it to the application.

[1] H/L-layer refer to high/low layer, respectively

The architecture of the VCI is flexible enough to enable the application developer to extend it by building custom API or AIC. Nevertheless, it is then the duty of the application developer to provide an interpretation of the custom API which enables the VCI to control the bus correctly. In this paper, only a limited set of APIs needed in many applications are developed, such as: configure, open, close connection, read data and write data.

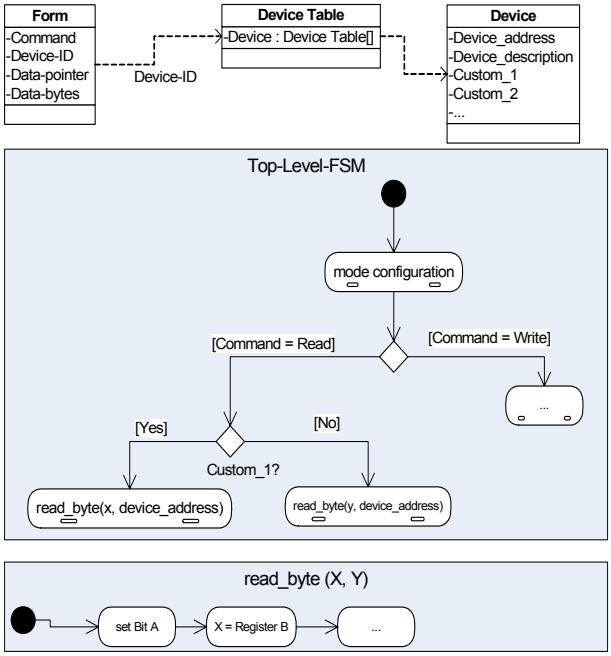

Figure 2. The basic structure of the M-FSM

3. VCI OPTIMIZATION AND PROTOTYPING

On one hand, the VCI enables the application to use different communication technologies as a black box using a common interface. On the other hand, the VCI represents an additional abstraction layer between the application and the underlying communication bus. To reduce the additional overhead induced by the VCI, the VCI is carefully co-designed by determining the right mix of hardware and software to improve performance. The performance of the VCI is further improved by using job-lists wherein the nature of applications can be considered.

A typical operational scenario in an embedded system which sends/receives data over an external bus is to read from external devices such as sensors and then to perform some computation in order to formulate a control command. In this case, the application developer has to formulate an CF to carry out a sequence of reads over the bus. The VCI then interrupts the application upon receiving the data. Frequent and consecutive short reads will interrupt the application frequently and negatively influence its performance even without using the VCI. The degree of performance degradation depends on the cost of handling interrupts and context switch.

The performance of the VCI is enhanced by concatenating consecutive reads/writes CFs in one or more job-lists. It is the duty of the application developer to decide how to group her/his CFs. A single API is then used to call the VCI, namely *process_joblist(form_type first_form)*. The address of the first CF is given by *first_form*. The VCI starts processing the first CF and then checks the pointer to the "next CF" and so on until reaching the last CF (next CF is *NULL*). The VCI then interrupts the application only when the whole job-list is finished. Therefore, the overhead and frequency of interrupting the application can be reduced.

Basically, the VCI can be implemented in hardware or in software as well. Anyhow, using a co-design approach can be a source of remarkable performance improvements at reasonable area costs [9]. In this paper, heuristic hardware/software co-design approach is adopted. The VCI is firstly implemented in software for comparison purposes. Then functional blocks of the VCI are migrated to hardware to improve system performance. Additional communication overhead between the functional blocks is taken into consideration during partitioning.

Partitioning the VCI in hardware/software considers also efficiency improvements enabled by job-lists: The API *process_joblist* is firstly interpreted by the M-FSM. Hardware M-FSM will receive only the address of the first CF, *first_form*, and then read/write the data directly from/to the main memory without the intervention of the application program which saves many processor cycles. The application is interrupted when the entire job-list is finished. Not only implementing the M-FSM in hardware reduces the overhead of processing job-lists but it is also much faster than software. A disadvantage of partitioning the VCI in this way is the required additional area. Anyhow, it is less than 5% of the available logic elements on the Altera Cyclone II development board. Usually, the PINT is provided as a hardware controller together with other bus-specific control logic. Therefore, only hardware implementation of the PINT is presented in this paper. It is worth to mention that job-lists together with hardware M-FSM can remarkable improve the overall performance and reduce the overhead at the application-level.

A prototype of the VCI is developed and tested on an Altera Cyclone II FPGA. Two different communication buses typically used in embedded automation systems are tested, namely: SPI and I^2C. An exemplar application is developed to access external devices using the proposed VCI without paying attention to the architecture-specific issues in the underlying communication technology. Besides verifying the applicability of the VCI, the prototype is used to measure the performance of the entire system to study the influence of the proposed optimization approaches. Selected experimental results are presented and discussed in the following section.

4. SELECTED EXPERIMENTAL RESULTS

A simple test application is developed to access external devices using the VCI prototype. The application is written only once and the SPI and I^2C buses are used. Using performance counters, the performance with and without VCI is then measured. Two different implementations of the VCI are evaluated and compared. Table 1 shows the required number of cycles and time in μs to transfer 256 Bytes using 100KB/Sec mode.

Table 1. Performance evaluation of the VCI

Type	I2C		SPI	
	Cycles	Time	Cycles	Time
Without VCI	3,474,606	40,878	3,182,512	37,441
Software VCI	3,553,242	41,803	3,336,008	39,247
Co-designed VCI	2,005,072	23,589	--	--

The results show that implementing the VCI in software degrades performance by a factor of 2.3% and 4.8% for I^2C and SPI, respectively. A basic reason for this expected performance degradation is the overhead of processing CFs in software by the M-FSM besides software handling of interrupts. It is worth to mention that performance degradation is inversely proportional to the data size being transferred using an CF. Transferring for example only 4 Bytes degrades performance by a factor of 17% and 51.5% for I^2C and SPI, respectively. 93 KB additional memory is needed by the software VCI. Co-designing the VCI in hardware/software remarkably improves the performance: Major portions of the M-FSM and the PINT are implemented in hardware. This accelerates the processing of CFs and reduces the overhead. As a result, the performance is improved by a factor of 42.3% compared to the reference implementation (without VCI) using I^2C at an additional cost of 1240 LUTs[2]. In the same manner, the performance improvement is proportional to the size of data being transferred. Unfortunately, the M-FSM hardware implementation of the SPI is not ready yet.

5. CONCLUSION AND FUTURE WORK

The proposed VCI provides the application with an architecture-neutral common interface. Code reusability is improved because the application should not be re-written when the communication bus is changed or updated. Despite the performance overhead introduced by the VCI which is inversely proportional to the data transferred size, co-designing the VCI in hardware/software improves the performance of the entire system remarkably. This improvement in performance is proportional to the size of data being transferred. Being able to group sporadic reads/writes using job-lists, the application can enhance the performance and reduce overhead remarkably.

In order to reconfigure the VCI automatically when the communication bus is changed, a builder is currently under development. More APIs, such as *search_dev,* are also under consideration.

6. REFERENCES

[1] Mohsen, A., and Bargothi, A. 2008. Custom Acceleration Solutions for Real-life Embedded Systems: Potential and Challenges. In Proceedings of Embedded World 2008 Exhibition and Conference (Nuernberg, Germany, February, 2008).

[2] I^2C Bus Specifications. Philips Semiconductors. Document number 9398393400II, 2000.

[3] Herveille, R. 2003. SPI Core Specifications. OpenCores.

[4] CAN Specifications. Robert Bosch GmbH, Germany, 1991.

[5] Edwards, W. 2000. Core Jini. Prentice Hall International.

[6] Lapierre, D. 1999. Tina: A Co-operative Solution for a Competitive World. Prentice Hall.

[7] Miyazaki, T., Takahara, A., Ishihara, S., Tani, S., Murooka, T., Fukazawa, T., Teramoto, M., and Matsuhiro, K. 2000. Virtual BUS: a Network Technology for Setting up Distributed Resources in Your Own Computer. Parallel and Distributed Processing Symposium, 535-540.

[8] IEEE 1451.0, Standard for a Smart Transducer Interface for Sensors and Actuators–Common Functions, Communication Protocols, and Transducer Electronic Data Sheet (TEDS) Formats. IEEE Instrumentation and Measurement Society, the Institute of Electrical and Electronics Engineers, Inc., New York, 2007.

[9] De Michali, G., and Gupta, R. 1997. Hardware/Software Co-design. Proceedings of the IEEE, 85(3):349-365.

[2] LUT: look up table