mPlatform: A Reconfigurable Architecture and Efficient Data Sharing Mechanism for Modular Sensor Nodes

Dimitrios Lymberopoulos
Yale University
New Haven, CT, 06511
dkl22@pantheon.yale.edu

Nissanka B. Priyantha
Microsoft Research
Redmond, WA, 98052
bodhip@microsoft.com

Feng Zhao
Microsoft Research
Redmond, WA, 98052
zhao@microsoft.com

ABSTRACT

We present mPlatform, a new reconfigurable modular sensornet platform that enables real-time processing on multiple heterogeneous processors. At the heart of the mPlatform is a scalable high-performance communication bus connecting the different modules of a node, allowing time-critical data to be shared without delay and supporting reconfigurability at the hardware level. Furthermore, the bus allows components of an application to span across different processors/modules without incurring much overhead, thus easing the program development and supporting software reconfigurability. We describe the communication architecture, protocol, and hardware configuration, and the implementation in a low power, high speed complex programmable logic device (CPLD). An asynchronous interface decouples the local processor of each module from the bus, allowing the bus to operate at the maximum desired speed while letting the processors focus on their real time tasks such as data collection and processing. Extensive experiments on the mPlatform prototype have validated the scalability of the communication architecture, and the high speed, reconfigurable inter-module communication that is achieved at the expense of a small increase in the power consumption. Finally, we demonstrate a real-time sound source localization application on the mPlatform, with four channels of acoustic data acquisition, FFT, and sound classification, that otherwise would be infeasible using traditional buses such as I^2C.

Categories and Subject Descriptors: C.3 [Special-Purpose and Application-Based Systems]: Real-time and embedded systems

General Terms: Design, Performance

Keywords: Modular Architecture, Reconfigurable Sensor Node, CPLD, High Speed Data Bus.

1. INTRODUCTION

The diverse nature of sensornet applications, ranging from environmental and industrial monitoring to healthcare, smart homes and entertainment, requires reconfigurability and extensibility at the hardware platform level to meet application-specific needs. Traditionally, this is achieved by adding an application-specific daughter board with sensors and actuators to a main processing board that typically consists of a processor and a radio [12, 3, 13, 1, 9]. This one-application, one-platform approach serves the application needs well but at the cost of potentially redundant development effort. The resulting platform is often limited in its ability to accommodate the diverse computing needs of different applications with the same main processor board. This is evident from the sensor platforms that the research community has built, such as the MK2 sensor node [16] for localization, the ATLAS node [6] for smart homes, the iBadge node [8] for speech processing and localization, and the sensor platform from Hitachi [19] for long-term real-time health monitoring.

Recently, modular sensornet platforms have been proposed to enable plug-and-play customization of a single platform for several different application domains [17, 4, 2, 7]. These platforms, instead of aiming for minimal form factor and power consumption, focus on flexibility, scalability and reconfigurability of resources. They typically comprise a collection of commonly used hardware modules that share the same well defined interfaces. These interfaces allow the seamless interconnection of these modules in any order and combination. Each module provides some computation, storage, sensing, or communication resources. Users can choose a set of hardware modules that best meet their application/research needs and quickly create their own custom sensor network platform without having to build new hardware from scratch.

However, the design of modular sensor network platforms is as challenging as appealing. Since multiple hardware modules with different resources are interfaced together to form a sensor node, the need for sharing every module's data across the stack automatically comes up. This need tends to be one of the most important bottlenecks in modular architectures. The reason is that in most cases traditional serial buses such as I^2C and SPI or serial protocols such as $RS232$ are used to communicate data across the different modules in the stack. These serial buses have two main drawbacks. First, they do not scale well with the speed of commonly used embedded processors. For instance, the high speed mode of the often used addressable I^2C bus is $400KHz$, while the different microprocessors used in sensor nodes such as AVR, MSP430 and ARM7 can be clocked up to $4MHz$, $8MHz$ and $60MHz$, respectively. Second, these buses scale poorly with the number of modules in the stack since only a single pair of modules can use the bus at any given time.

Current state-of-the-art modular platforms [17] make use of switchable serial buses to address this problem. While this approach works well when the communicating pairs of modules in the stack are disjoint, it fails when multiple modules have to share data with the same hardware module in the stack. For instance, this approach does not work in the case of the Sound Source Localization application described in Section 4, where 4 MSP430-based hardware

IPSN'07, April 25-27, 2007, Cambridge, Massachusetts, USA.

modules have to exchange data with a single ARM-based hardware module at the same time. In addition, the switchable serial bus approach requires software to implement task-specific interfaces and channel configurations, thus hindering the reusability and reconfigurability of both hardware and software.

mPlatform addresses this problem by introducing a new flexible, efficient and reconfigurable communication channel architecture that better fits the needs of modular sensor network platforms. Our architecture is based on the following key design requirements:
Resource Efficiency: the communication channel should be able to operate at the maximum possible speed as defined by the communicating processors. For instance, the channel should be able to operate much faster when data is being exchanged between two ARM7 processors compared to two MSP430s.
Processor Independence: The communication channel should not be aware of the capabilities of the two communication end points. In other words, resource efficiency should be achieved automatically without having to explicitly configure the communication channel parameters according to the communicating processors capabilities. This allows the communication channel to transparently support a diverse set of processors that can vary from low-end CPUs such as AVR and MSP430 to more capable ARM7 and PXA processors.
Scalability: The end-to-end communication channel delay should not be significantly affected by the number of modules in the stack.
Fairness: Each module in the stack should use the bus independently of the other modules in the sense that it does not have to stall and wait for another module to complete its data exchange before it starts sharing its own data.
Reconfigurability: Users should be able to easily adjust or even completely re-design the communication channel in order to optimize it according to their application needs without having to modify the hardware.

mPlatform meets these design requirements by abstracting the communication channel from the communicating processors. The local processor on each module interacts with a parallel bus through a bus controller, implemented in a low-power CPLD. This approach decouples the communication channel from the local processor, allowing different processors running at different speeds to share the bus without impacting its throughput. To guarantee *fairness* a TDMA-based protocol implemented in the bus controller allows multiple processors to exchange data almost simultaneously. The high data rate enabled by the parallel bus combined with the TDMA protocol create a near real time inter-module communication channel that *scales well* with the number of modules in the stack. *Processor independence* and *resource efficiency* are achieved by enforcing an asynchronous interface over a separate parallel bus between the CPU and the bus controller. This enables the bus controller to be transparently interfaced to a processor running at any speed. The asynchronous nature of the interface enables the processor to transfer data at a speed usually limited by the processor clock speed because of the relatively high clock speed of the bus controller.

The advanced functionality of the new communication architecture comes at the expense of a small increase in the power consumption of the platform mainly due to the use of CPLD. However, the flexibility afforded by the CPLD outweighs the small power overhead, as we will detail in the evaluation section later. The mPlatform architecture is a research platform designed to facilitate rapid prototyping and experimentation. Complex programmable logic devices provide an abstraction of the hardware layer that drastically simplifies the tinkering at the protocol level requiring little change to hardware.

The rest of the paper focuses on the design and evaluation of the proposed inter-module communication mechanism. Section 2 provides an overview of the mPlatform architecture and its supporting software infrastructure. In Section 3 the communication channel architecture is described in detail and in Section 4 the experimental evaluation of the communication mechanism is presented. Section 5 discusses the related work and concludes the paper.

2. THE MPLATFORM ARCHITECTURE

mPlatform is a modular architecture that focuses on providing a Lego-like, plug-and-play capability to put together a sensor network platform tailored to specific application/research requirements. The hardware architecture design was driven by the following baisc guidelines:
Reconfigurability: The architecture needs to be easily reconfigurable to meet specific needs of a particular research project. For example, a data collection task with a low sampling rate may just require an 8-bit processor and a slow radio connection to a gateway to conserve power, while a physiological monitoring application on a body sensor network that alerts a remote physician upon detecting an abnormal condition will need more processing power to analyze the signals, enough storage for disconnected operation, and the ability to connect to multiple wireless networks. To enable reconfigurability, mPlatform was designed so that a wide range of processors, from MSP430 class processors up to PXA270 processors, can coexist on the same platform and efficiently communicate in any possible configuration.
Real-time event handling: Since a sensor platform is typically used in applications where it constantly interacts with the environment, the ability to handle real-time events is crucial. Examples include detection of an abnormally high temperature indicating a fire, detection of an abnormal physiological signal, and arrival of a radio packet.
Fine-grained power management: In many sensing and mobility applications nodes are powered by battery or salvaged energy sources. It is desirable to be able to shut down components when not in use and scale up or down operation voltage and/or frequency of the components in order to accommodate task needs while conserving energy and other resources.

2.1 Architecture Overview

The mPlatform is a collection of stackable hardware modules that share a well defined common interface. Physically, each module consists of a circuit board and connectors that enable other modules to be plugged on both top and bottom of that module. Some of the mPlatform modules are general purpose processing boards while others are special purpose boards such as radio boards for wireless communication, sensor boards for sensing physical phenomena, and power boards for supplying power to a stack of modules. Each board, except for the power board, has a local processor. Having a local processor on each module enables efficient real-time event handling, one of the major design goals of mPlatform. The processor-per-module approach also allows a more customizable aggregation of processing power appropriate for a given application. This is in contrast to some embedded system platforms that use a single processor to manage multiple add-on boards [12, 1, 13].

In addition to the components that implement a particular module's basic functionality, each module has a low power configuration processor that can be used to configure the stack of modules or even reprogram the main processing components on each hardware module. The MSP430 microprocessor from Texas Instruments ($MSP430F1611$) is used as the main processor in sev-

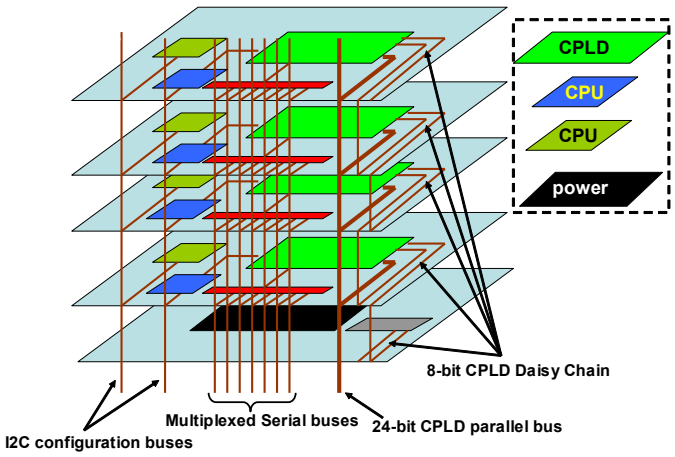
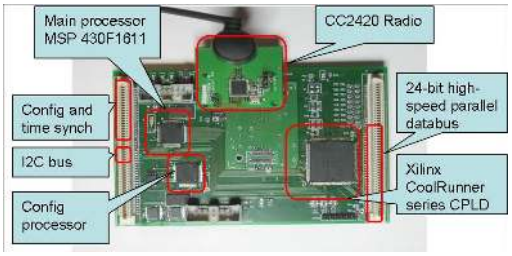


Figure 1: Overview of the mPlatform architecture.



(a)

(b)

Figure 2: (a) A typical MSP430-based hardware module. (b) A 4-module stack. At the lowest level is the power board. Next is an MSP430/CPLD board and an MSP430/CPLD/CC2420 board. The ARM7/CPLD board is on the top.

eral modules. Featuring $10KB$ of RAM, $48KB$ of Flash and a maximum speed of $8MHz$, it can provide basic data sensing and processing functionality at a very low power overhead as demonstrated in [13]. Because of its extremely low power consumption in idle and sleep modes, we use it as the configuration processor embedded on every hardware module.

On the other hand, the OKI ML67Q5003, an ARM7TDMI processor, is embedded on a different class of hardware modules that can be used to support more complex real-time data processing. It operates at a maximum frequency of 60MHz but an internal software controlled clock divider can slow down the processor in powers of two all the way down to 60/32MHz(1.875MHz) to conserve energy. Other attractive features of the OKI processor, despite its relatively high static power consumption [9], include the variety of power management features of all the peripherals on the chip, the availability of 7 hardware timers, the option to process external interrupts on all the general purpose IO pins as well as the relatively rich internal memory of $32KB$ RAM and $512KB$ Flash.

Figure 1 shows a high level block diagram of the mPlatform architecture. Each module connects to multiple buses for inter-processor communication through a uniform hardware interface that can be seen in Figure 2(a). The interface makes it possible to stack together any combination of hardware modules to implement a sensor platform for an application (Figure 2(b)). The centerpieces of this interface are a 24-bit wide parallel bus and an 8-bit wide daisy chain bus that are both interfaced to the processor through a high speed, low power CPLD. To achieve low power operation without compromising the performance of the communication channel we opted to use the Xilinx XC2C512 CoolRunner-II CPLD. This family of CPLDs can operate at a maximum speed of $200MHz$ and its power consumption can vary from approximately a few mW up to $260mW$ due to the embedded frequency scaling capabilities.

This hardware configuration expands the available data buses used in other modular sensor node platforms [17, 4, 2, 7, 14] with a CPLD-based communication channel with the following advantages:

1. The communication channel is abstracted by the CPLD. The processor only communicates with the CPLD and does not need to be aware of the bus implementation details. Since the bus is controlled by the CPLD and not the CPU, local processors on individual hardware modules can enter a deep sleep mode to reduce power consumption, while other processors on different hardware modules can actively communicate data over the bus.

2. The performance of the bus depends on CPLD's operating frequency and not on the operating frequency of the processor. This enables different hardware modules with different processors running at various clock speeds to share the same data bus without affecting its maximum speed.

3. The communication channel does not have to serialize/deserialize the data before/after transmission since parallel lines of up to 64-bits width can be used for data exchange.

4. The communication channel is easily reconfigurable since it is solely controlled by the CPLD. Programming the CPLD with a high level hardware description language such as Verilog HDL or VHDL facilitates the process of designing and using a new communication protocol without having to modify the actual hardware design. This approach significantly expands the flexibility of the existing state-of-the-art stack-based architectures by enabling programmers to optimize their communication channel according to the specific application requirements while using general purpose hardware modules.

For maximum flexibility, a set of switchable serial buses enables dynamic pair-wise communication between processors using standard serial protocols such as RS232 and SPI. A multi-master I2C bus is used to configure and manage the stack of modules. A separate multi-master I2C bus is used for the secondary processors that are responsible for configuring and managing the stack of modules.

2.2 Software Support

Enabled by this flexible and scalable inter-module communication mechanism, we have developed a light weight, priority based operating system to provide scheduling and run-time support for applications running on the mPlatform. To facilitate the application development on the multi-processor architecture, mPlatform supports a data-flow style programming abstraction for the applications [5]. In this abstraction, an application is specified as a collection of communicating tasks, and mPlatform provides the support for task naming, synchronization, and communication. Since tasks of a single application may span a number of processors in different modules, a uniform messaging interface, enabled by the underlying inter-module communication bus, enables tasks to communicate through message passing either locally within the module or across

the modules via the bus, all in a way transparent to the user. The interface also simplifies task migration from module to module, as the need to load balance or mitigate failures arises [5]. Again, this simple and effective abstraction to the inter-module communication is made possible because of the performance and flexibility of the communication bus.

One important feature of the multi-processor mPlatform software support is the ability to allocate and schedule the different tasks to available modules. Our task allocation algorithm does this by constraining the task assignment with power and deadline requirements [10]. This is accomplished by modeling resources such as power usage of processors and CPLDs as well as latency in processing and communication. The scheduling problem then becomes a constrained optimization. The development of the SSL application, to be discussed in Section 4, makes heavy use of the data-flow specification, message passing, and scheduling features of the mPlatform software infrastructure.

3. HIGH-SPEED INTER-MODULE COMMUNICATION CHANNEL DESIGN

Designing a communication protocol on top of the hardware architecture of mPlatform that meets the design requirements described earlier involves **(1)** defining how CPLDs on different modules are wired together, **(2)** specifying the protocol for sharing data among the CPLDs on different modules, and **(3)** defining the CPU interface through which the CPU reads/writes data from/to the CPLD on every module. We present the design choices and tradeoffs in each of these steps.

3.1 Hardware Configuration

We use a shared parallel bus architecture to connect the CPLDs on different modules together. In this configuration, all the CPLDs on different modules share a 24-bit data bus and several control signals. This approach has the major advantage of enabling direct communication between any pair of modules on the bus. This makes the communication delay between any pair of modules small and constant regardless of the module location in the stack. However, a common shared bus requires mechanisms for efficiently sharing the bus and for avoiding collisions due to multiple simultaneous transmissions. We use several control signals, including common reset and clock signals, to implement a Time Division Multiple Access (TDMA) based bus protocol that enables efficient sharing of the common bus.

Serial and daisy chain bus configurations are two common alternatives to the parallel shared bus. Apart from the reduction in the number of wires used for communication, a major advantage of the serial bus is the absence of data and clock skew (the offset of the data and clock signals on different wires) which is inherent in high speed parallel bus designs. However, one major drawback of the serial bus configuration is the need for a higher clock speed to achieve the same throughput as a parallel bus, since a parallel bus allows multiple bits to be sent within one clock cycle. Since this increased clock speed results in increased power consumption, and we did not notice any significant clock skew at the maximum operating frequency of the mPlatform bus, we decided not to use a serial bus configuration. In addition, the number of wires required for interfacing the CPLDs at the different modules in the stack is not a problem because of the large number of available general purpose I/O pins on the CPLD chip.

On the other hand, in a daisy chain configuration the CPLDs are connected sequentially, where each CPLD can directly communicate only with its immediate neighbors. Any non-adjacent communication has to involve a sequence of pair-wise communications. The local communication between CPLDs simplifies the communication interface and makes it lightweight. At the hardware level, the daisy chain limits (1) the load on the transmitter logic since there is only one receiver per transmitter, and (2) the data and clock skew since the length of wiring between modules is short. However, daisy chain configuration has performance and scalability issues. The communication between non-adjacent modules involves buffering and forwarding of data at intermediate modules. Consequently, the end-to-end communication delay and power consumption increase with the number of modules.

3.2 CPLD-Based Communication Protocol

The communication protocol for sharing the CPLD parallel bus is designed to **(1)** prevent collisions when multiple modules in the stack attempt to use the bus at the same time, and **(2)** multiplex the access to the communication channel so that every module can send its data almost immediately without blocking other modules in the stack. Note, that the problem of shared resource access, in particular to communication buses, is one of the most widely studied optimization problems [15]. We did not focus on implementing the optimum protocol. Instead, we focused on designing a protocol that takes advantage of the underlying efficient architecture and that is simple and generic enough to be used in a variety of different applications. However, any application specific optimized protocol can be implemented and run on top of the proposed bus architecture without modifying the hardware.

To eliminate collisions and enable fair sharing of the communication channel among modules we decided to use a TDMA protocol. The time is divided into identical slots and every module in the stack is assigned a single unique slot. The CPLD on each module is allowed to send data only during its assigned time slot; all the other modules that do not transmit during a given slot listen to the channel for valid data. This approach enforces fairness among the different modules since the communication channel access is equally divided across all the modules.

The duration of each slot is equal to the time required to successfully transmit and receive a single data packet over the CPLD bus. The bus is 24-bit wide, and hence the packet size. Of these 24 bits, the most significant byte contaaddressing information, and the remaining two bytes are the data payload. Each module in the stack is assigned a unique 4-bit address. The 4 most significant bits of the addressing byte specify the destination while the rest the source. A special broadcast address enables broadcasting over the bus. Therefore, independently of the addressing mode used, unicast or broadcasting, for every data packet 3 bytes are sent over the parallel bus: 1 address byte and 2 data bytes.

3.3 CPU Interface

An asynchronous interface over an 8-bit wide data bus is implemented for the communication between the processor and the CPLD. The main design considerations behind choosing an asynchronous interface over a synchronous one were the following:
A **synchronous interface** requires the processor and the CPLD to share a common clock. However, the maximum clock rate that can be used depends on the capability of the processor used at each module. Low-end processors (e.g. MSP430) can sustain on lower clock rates than high-end processors (e.g. ARM7). Therefore, in order to efficiently utilize the local processor's resources the CPLD design must be aware of the characteristics of processor at which it is interfaced to. This reduces the flexibility and ease of use of the CPLD bus by requiring different versions of the CPLD data bus design running on different modules.

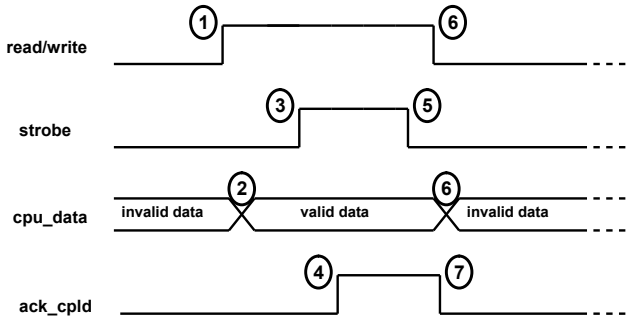


Figure 3: Asynchronous interface for writing a single byte to the CPLD.

An **asynchronous interface**, on the other hand, allows both of the communication end points to operate at their maximum speed without sharing any clock information. In addition, an asynchronous interface is simpler since less amount of state information has to be recorded and handled. This is very important when using small CPLDs like the Xilinx $XC2C512$ where the available design resources are limited. For instance, the implementation of an SPI interface on the XC2C512 CPLD would require almost 50% of the available resources. An asynchronous interface would minimizes the use of CPLD resources, leaving the rest for implementing other advanced features of the data bus, such as larger memory or support for multiple slots per module.

Figure 3 shows the timing diagram of the CPU interface for writing a single byte to the CPLD. The CPU first waits for the *ack_cpld* signal to become low. Next, it raises the *read/write* and then outputs the data byte on the data bus. It then raises the *strobe* signal to indicate that data is ready to be read by CPLD. The CPLD acquires the data on the 8-bit data bus and it raises the *ack_cpld* signal to indicate that it has read the data. After detecting the rising edge of the *ack_cpld* signal, the CPU lowers the *strobe* signal and then lowers the *read/write* signal. After detecting the falling edge of the *strobe* signal, the CPLD lowers the *ack_cpld* signal. The process of reading a byte from the CPLD is similar. The only difference now is that the *read/write* signal is kept low by the CPU and the data bus is controlled by the CPLD and not the CPU.

The fact that every packet sent over the CPLD bus is 24-bit wide while the bus shared between the CPU and the CPLD is only 8-bit wide requires the CPU to perform three consecutive byte read/writes in order to read/write a single data packet. The first byte written to the CPLD contains the destination address. When the value of the address byte is between $0x00h$ and $0x0Fh$, the data is transmitted as a unicast packet. If the address byte is equal to $0x0Fh$ the data packet is broadcast on the CPLD data bus. In both cases, the next two bytes represent the data payload to be sent over the bus. All the bytes written to the CPLD are first processed by the CPU interface module to include source address information, before they are written to the transmission FIFO. The CPU interface module modifies the address byte (first byte written) such that the 4 most significant bits correspond to the destination address and the least 4 significant bits correspond to the source address.

Besides unicast and broadcast transmission of data packets, the CPU interface allows the CPU to configure several parameters of the communication protocol. In particular, the CPU can set the address of the CPLD, the slot used by the CPLD as well as the total number of slots used. The slot assigned to the CPLD is set by setting the address byte to $0xA0h$. In this case the second byte defines the slot assigned to the CPLD and the third byte defines the total number of slots. The address of the CPLD can be set by setting

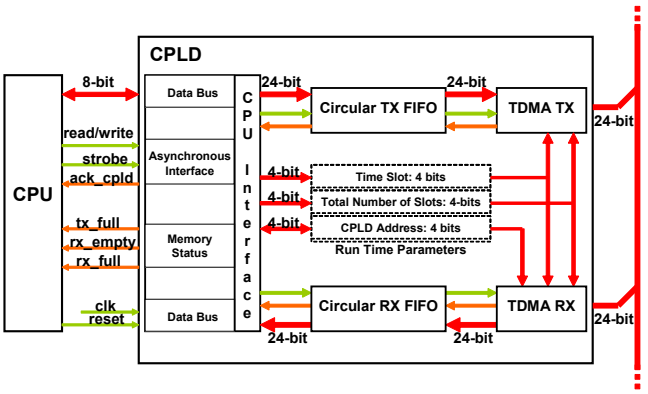


Figure 4: Overview of the inter-module communication channel architecture.

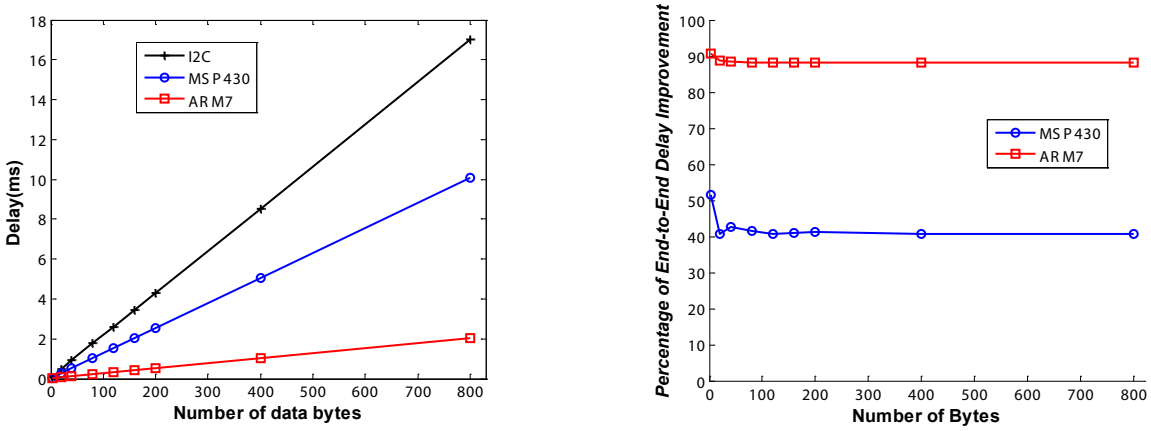
the address byte to $0xB0h$. In this case, the second byte defines the address assigned to the CPLD and the third byte is ignored. When setting the address and slot number information no data packets are transmitted over the parallel data bus.

3.4 Implementation

The architecture of the CPLD-based communication channel is shown in Figure 4. The CPU interface component is responsible for the communication between the CPLD and the local processor on each module. It allows the CPU to read/write packets from/to the CPLD and set the address, the slot and the total number of slots. Each data packet the processor sends to the CPLD is written to the transmission FIFO. The TDMA transmitter continuously checks the transmission FIFO; when the FIFO is not empty the transmitter reads the first available packet, waits for the assigned slot, and transmits the packet on to the bus. When its slot is deactivated, the transmitter surrenders the control of the data bus. The TDMA receiver module, running in parallel, is sniffing the data bus at the beginning of every slot. If a valid packet is on the bus and the address decoding is successful then the packet is written to the reception FIFO.

Every time a packet is written or read in any of the two memories the memory status signals directly connected to the processor and the other modules in the CPLD design are updated. In that way, the CPU as well as both the TDMA transmitter and receiver are always aware of the memory status (full or empty) and they can proceed reading packets when they are available. Both of the two memory modules are implemented as a circular FIFO that supports simultaneous read and write operations for maximum performance. The transmission FIFO is always written by the CPU interface module and it is always read by the TDMA transmitter module. Similarly, the reception FIFO is always written by the TDMA receiver and it is always read by the CPU interface module. Note that all the internal buses between the individual modules are 24-bit wide to minimize the number of clock cycles required for transferring a packet from the input to the output of the CPLD data bus. In that way, only 2.5 CPLD clock cycles are required to transfer a data packet from the CPU interface module to the TDMA transmitter and vice versa.

The TDMA-based communication protocol has been implemented in Verilog HDL and has been mapped to the $XC2C512$ CPLD from Xilinx. Our design makes use of approximately 60% of the resources when using a 2-packet transmission FIFO and an 8-packet reception FIFO and it can be clocked up to $68MHz$. In our implementation, each time slot of the TDMA scheme corresponds to two CPLD clock cycles which is equal to the time that it takes to



(a) (b)

Figure 5: (a) Measured end-to-end communication delay comparison for different number of communicating bytes between a pair of both ARM7 and MSP430 processors. (b) Percentage of end-to-end communication delay reduction of the CPLD bus over the I^2C bus for both the MSP430 and the ARM7 processors.

transmit/receive a 24-bit data packet. Since CPLD bus takes two $68MHz$ clock cycles to transfer a packet with a 2 byte data payload, the maximum throughput of the bus is $68 \times 8 = 544Mbits/s$; compared to the $400Kbits/s$ maximum throughput of the I^2C bus, this is a 3 orders of magnitude improvement.

3.5 Design Considerations

The TDMA mechanism presented in Section 3.2 assigns time slots to all the modules in the stack. Note that some modules might not always send data during their time slots. As a result, CPLDs at different modules might have to stall waiting for their slots to become active, even though the communication channel is not used by anyone. This could introduce delays in the communication channel that could possibly violate some of our basic design goals. However, the operating frequency of $68MHz$ that we have achieved using the Xilinx CoolRunner-II CPLDs minimizes the effect of data bus sharing on the communication delay. The reason is that the CPLD communication protocol is implemented in hardware that can operate at a much higher speed ($68MHz$) compared to the effective clock rates of the most commonly used microprocessors such as AVR ($< 16MHz$), MSP430 ($< 8MHz$) and ARM7 ($< 60MHz$). In practice, this means that the time it takes for the microprocessors to write a data packet to the CPLD is much higher than the time that corresponds to a time slot (2 CPLD clock cycles). For instance, as it will be shown in detail in Section 4, the time to write a packet to the CPLD ($21\mu s$) is approximately 715 times higher than the duration of a time slot ($0.029\mu s$). This allows the CPLD to transparently multiplex the data bus without hurting the performance of the communication channel.

4. EVALUATION

In this section we evaluate the mPlatform architecture. First, we experimentally characterize the performance of the CPLD-based communication channel. We examine the scalability of the TDMA protocol with respect to the number of modules in the stack, and compare the results with the I^2C bus. We also examine the power consumption of the CPLD bus implementation. Second, we demonstrate the functionality enabled by the mPlatform architecture using a case study: the Sound Source Localization (SSL) application [20].

4.1 Performance Evaluation

4.1.1 End-to-End Communication Delay

In Section 3 we showed that the maximum throughput of the CPLD bus is 3 orders of magnitudes larger than the maximum throughput of the I^2C bus. In this experiment we measured the end-to-end communication delay–defined as the time between the start of the transmission of the first byte and the end of the reception of the last byte for different number of bytes and for both the CPLD and the I^2C buses. For the CPLD bus, we measured the delays for a pair of MSP430F1611 processors running at $6MHz$, and a pair of OKI MLQ57003 processors running at $60MHz$. In both configurations, the CPLD was clocked at $32MHz$ and the number of slots was set to 2. A simple application was used to generate a random sequence of data bytes with predefined length varying from 2 to 800. These sequences of bytes were sent over both the I^2C and the CPLD buses and the end-to-end communication delay was recorded using an oscilloscope. To guarantee the maximum utilization of the I^2C bus, we implemented an interrupt driven I^2C driver and verified that the I^2C bus was operating at the maximum speed of $400KHz$.

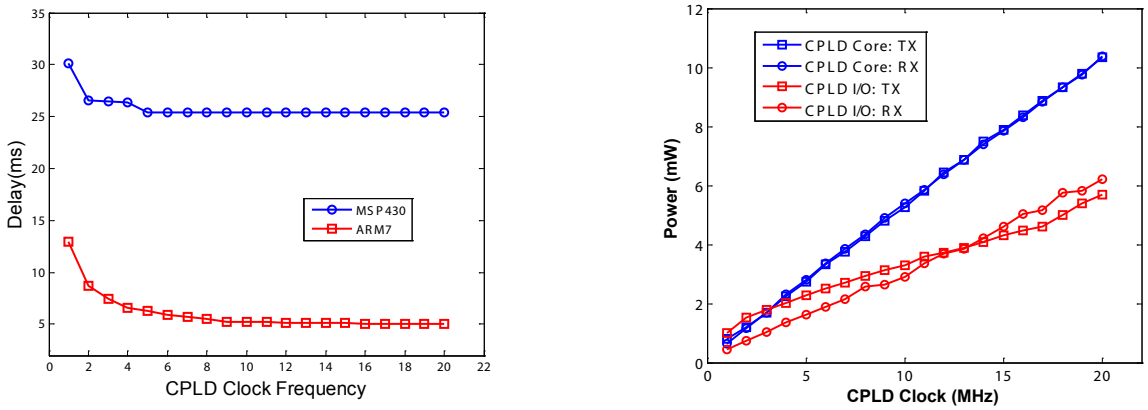
Figure 5(a) shows the results of these experiments. It is clear that in all cases the end-to-end delay is a linear function of the number of bytes transmitted. After applying a best linear fit to the data points shown in Figure 5(a) we derived the communication delay for all configurations as a function of the number of transmitted bytes N:

$$T_{I^2C}(ms) = 0.021 * N + 0.066 \quad (1)$$

$$T_{CPLD}^{MSP430}(ms) = 0.013 * N + 0.024 \quad (2)$$

$$T_{CPLD}^{ARM7}(ms) = 0.0025 * N + 0.0043 \quad (3)$$

According to (1), (2) and (3), the CPLD bus is always faster than the I^2C bus. For instance, using (1) and (2) we can compute the estimated end-to-end communication delay for both buses when 2048 bytes have to be exchanged between the two modules. The I^2C bus would require approximately $43ms$, while the CPLD bus would require $26.6ms$ (MSP430-based modules) and $5.12ms$(ARM-based modules), a reduction of $16.4ms$ and $37.88ms$ respectively. As it will be shown later, such a reduction in the communication delay could be critical in the case of real time applications.



(a) (b)

Figure 6: (a) The measured end-to-end communication delay for 1000 packets (2000 data bytes) as a function of the CPLD clock frequency. (b) Measured power consumption of the Xilinx $XC2C512$ CPLD on the ARM7 module in full transmit and receive modes.

Note, that this significant reduction in the end-to-end communication delay is due to the replacement of the $400KHz$ I^2C serial bus by an asynchronous 8-bit parallel bus between the processor and the CPLD that operates at the processor's maximum clock frequency. In that way, we are able to shift the bottleneck in the end-to-end communication delay from the I^2C low frequency clock to the actual processor's high speed clock.

Figure 5(a) also verifies that the CPLD-based communication channel fulfils two of the most basic design goals: resource efficiency and processor independence. Note that the end-to-end delay of ARM modules is approximately 5 times smaller than that of MSP430 modules; although, we used the same CPLD design in both cases. This is due to the asynchronous interface between the CPU and the CPLD. This interface allows the CPLD to communicate data over the bus as fast as the processor can deliver it without requiring to be aware of the exact processor that it is interfaced to.

Figure 5(b) provides more insight into the experimental results. When a small number of bytes (e.g. 2 or 4) is transmitted, the CPLD bus can be up to 52% (MSP430 modules) or 92% (ARM modules) faster than the I^2C. However, as the number of bytes transmitted increases the performance gap between the two buses narrows and eventually becomes stable after 160 bytes (MSP430 modules) or 80 bytes (ARM modules). When sending a large number of bytes, the CPLD bus is approximately 42% (MSP430 modules) or 88% (ARM modules) faster than the I^2C bus. This variation in the performance gap is due to the high overhead of the I^2C bus caused by the 8-bit destination address and the START and STOP conditions sent over the bus. When the number of transmitted data bytes is small, this overhead is relatively high leading to a larger performance gap between the I^2C and the CPLD buses. However, when the number of bytes increases, the impact of this overhead is minimized.

4.1.2 *Scalability Study*

One of the main design goals of the CPLD communication channel was to be able to scale well with the number of modules in the stack. In other words, the end-to-end delay performance of the TDMA-based scheme should not deteriorate significantly when the total number of slots increases.

In the experimental results shown in Figure 5 the minimum possible number of slots (*i.e*, 2) was used. When increasing the number of slots each data packet transmission might be delayed for an additional number of slots. Since the communication channel clock is the CPLD clock and each slot consists of 2 CPLD clock cycles we can compute the worst case overhead of end-to-end delay when the number of slots increases.

Number of slots	Number of data packets (Number of data bytes)					
	CPLD CLOCK: $32MHz$					
	MSP430($6MHz$)			ARM7($60MHz$)		
	40(80)	100(200)	400(800)	40(80)	100(200)	400(800)
3	0.0059%	0.0024%	0.0006%	0.0306%	0.0124%	0.0031%
4	0.0117%	0.0048%	0.0012%	0.0612%	0.0248%	0.0062%
5	0.0176%	0.0071%	0.0018%	0.0918%	0.0372%	0.0094%
6	0.1235%	0.0095%	0.0024%	0.1224%	0.0496%	0.0125%
7	0.0294%	0.0119%	0.0030%	0.1530%	0.0620%	0.0156%
8	0.0352%	0.0143%	0.0036%	0.1836%	0.0744%	0.0187%
9	0.0411%	0.0167%	0.0042%	0.2141%	0.0868%	0.0218%
10	0.0470%	0.0191%	0.0048%	0.2447%	0.0991%	0.0249%

Table 1: Worst case end-to-end delay overhead for different number of packets transmitted while varying the total number of slots in the system for both the MSP430 and the ARM7 processors.

Assume that the total number of slots increases from 2 to M. Then the *worst case overhead* on the end-to-end delay of transmitting a single packet over the CPLD bus (with respect to the delay measured when 2 slots are used) will be equal to:

$$T_{overhead}(\mu s) = (M-2) * 2 * CPLD\_CLK, M \geq 3 \quad (4)$$

where $CPLD\_CLK$ is the period of the CPLD clock expressed in μs. Equation (4) implies that every time we add a slot to the initial number of 2 slots the *worst case* end-to-end communication delay for a single data packet (2 data bytes) will increase by the duration of a single slot which is equal to 2 CPLD clock periods. Using this information it is possible to compute the overhead caused by the increased number of slots in the system for any arbitrary number of communicated data packets. Table 1 shows the percentage of increase in the end-to-end communication delay for different number of data packets when increasing the total number of slots from 2 up to 10. It is clear that the effect on the end-to-end delay communication delay is negligible as the number of slots increases and as the number of bytes sent also increases. This is because the CPLD can always operate at a higher clock speed than the processors it is interfaced to (in this case MSP430 and ARM7). As a result, the bottleneck in the end-to-end communication is the processor and not the CPLD. This provides enough time for the CPLD

to transparently time-share the data bus while communicating with the processor at a lower speed. Of course, when processors faster than the CPLD are used, the overhead of multiplexing the data bus becomes higher. However, the relatively high speed of the CPLD along with the inherent hardware parallelism and the inefficiency of general purpose processors (number of instructions per clock cycle etc.) ensure that the end-to-end delay performance will not be significantly affected by the number of time slots in the system even when high-end processors like the PXA CPU from Intel running from $100MHz$ to $200MHz$ are used.

Figure 6(a) sheds more light on the effect of CPLD's clock frequency on the performance of the data bus. The end-to-end delay for 1000 packets (2000 data bytes) as a function of the CPLD clock frequency is shown for both the MSP430 and the ARM7 processors (the total number of slots set to 2). It is clear that after a certain CPLD clock rate the end-to-end delay remains unchanged. This shows that the bottleneck becomes the CPU interface and the time required for the processor to read/write a packet to the CPLD. According to Figure 6(a), the minimum clock rate that allows the CPLD to transparently multiplex the data bus without increasing the end-to-end communication delay is $5MHz$ for the MSP430 processor, and $16MHz$ for the ARM7 processor.

Note, however, that even when the processor used is much faster than the CPLD (*e.g.*, in the case of an Intel PXA processor running at $400MHz$), the advantage of using the CPLD communication channel instead of traditional buses such as I^2C is still significant. The reason is that the bottleneck in the case of the CPLD bus would be the CPLD's speed which is equal to $68MHz$ while in the case of the I^2C bus the bottleneck is the actual I^2C clock which is $400KHz$, orders of magnitude less than the actual CPLD clock.

4.1.3 Power Consumption

The ability to transparently multiplex the communication channel across all the modules in the stack comes at the expense of increased power consumption. Figure 6(b) shows the core and IO power consumption of the Xilinx XC2C512 CPLD when it is driven by an ARM7 processor running at 60MHz. It is clear that the power consumption of the CPLD is a linear function of its clock frequency. At 16MHz, the minimum CPLD clock frequency when the ARM7-to-ARM7 communication is needed, the overall power consumption is approximately 13mW. This corresponds to less than 10% of the power consumption of the ARM7 processor [9]. At 5MHz, the minimum CPLD clock frequency when MSP430-to-MSP430 communication is needed, the overall power consumption is reduced down to 5mW. This is slightly larger than the power consumption of the MSP430 processor at full speed [13]. When the CPLD-based bus is not used by a board in the stack, the CPLD can be put into sleep mode consuming approximately $8.25\mu W$.

This shows that the CPLD communication channel achieves the performance and flexibility at a reasonable cost of power consumption. Besides, the mPlatform is designed to support rapid research prototyping of sensornet applications, rather than aiming for the absolute possible minimal power usage. Once debugged on the mPlatform, one can always re-implement the same communication architecture as the one shown in Figure 4 in a custom VLSI chip using say ASIC technology in order to minimize the power consumption.

4.2 Case Study

In this section we demonstrate how the mPlatform architecture can enable general purpose modular architectures to meet the real time processing requirements in real sensor network applications. We use sound source localization (SSL) as an example of a typical real time application. In the SSL application, an array of carefully spaced microphones is used to record sound. By measuring the time differences of arrival of the sound at the different microphones, SSL uses a combination of FFT, sound classification, and hypothesis testing and aggregation to determines the location of the sound source [20].

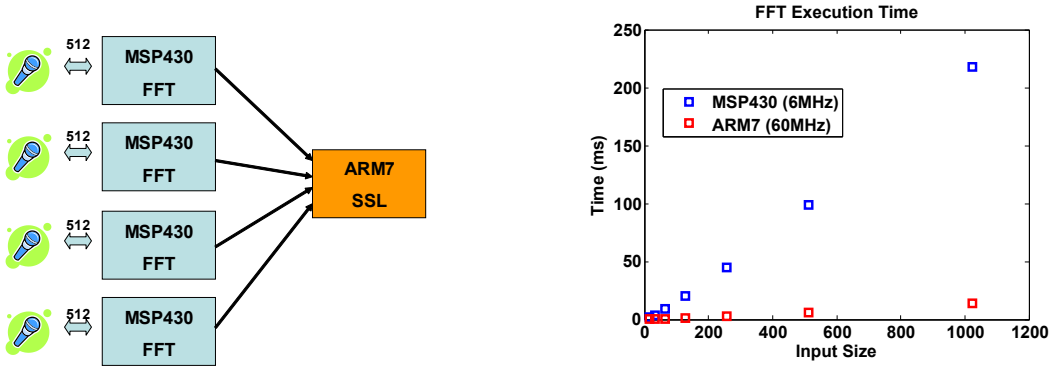
The configuration of the mPlatform architecture for supporting the SSL application, as shown in Figure 7(a), comprises 4 MSP430 based hardware modules and a single ARM7 based module. On each MSP430 module a microphone is interfaced to an embedded analog to digital converter (ADC) of the local CPU. The direct memory access (DMA) controller continuously captures blocks of 512 samples (each sample is a 16-bit value) allowing CPU clock cycles to be used for data processing simultaneously with data sampling. Every block of 512 audio samples is processed by a Fast Fourier Transform (FFT) software routine implemented on the local MSP430 microcontroller. The output of the FFT for every MSP430-based module has to be sent to the ARM7 module which is responsible for running the actual SSL algorithm. Note that 2048 bytes have to be communicated over the communication channel for every MSP430-based module. This is because the output of the FFT software routine has two parts: a real and an imaginary part. Each of these parts consists of 512 16-bit points that leads to a total number of 2048 bytes.

SSL requires that the mPlatform samples the sound data at a minimum of $4KHz$. In addition, the blocks of 512 samples provided by the MSP430-based boards to the ARM7 board have to correspond to exactly the same time period, otherwise the result of the SSL algorithm will be wrong.

To be able to verify if the mPlatform architecture would be able to support the SSL application we had to measure the execution time of the FFT software routine for different input sizes. Figure 7(b) shows the results of our measurements for both the MSP430 ($6MHz$) and the OKI ML67Q5003 ($60MHz$) CPUs. It turns out that the OKI processor is about 15 times faster than the MSP430. The execution time of the FFT software routine on the MSP430 for an input size of 512 points takes $99.2ms$ compared to the $6.32ms$ execution time on the OKI processor.

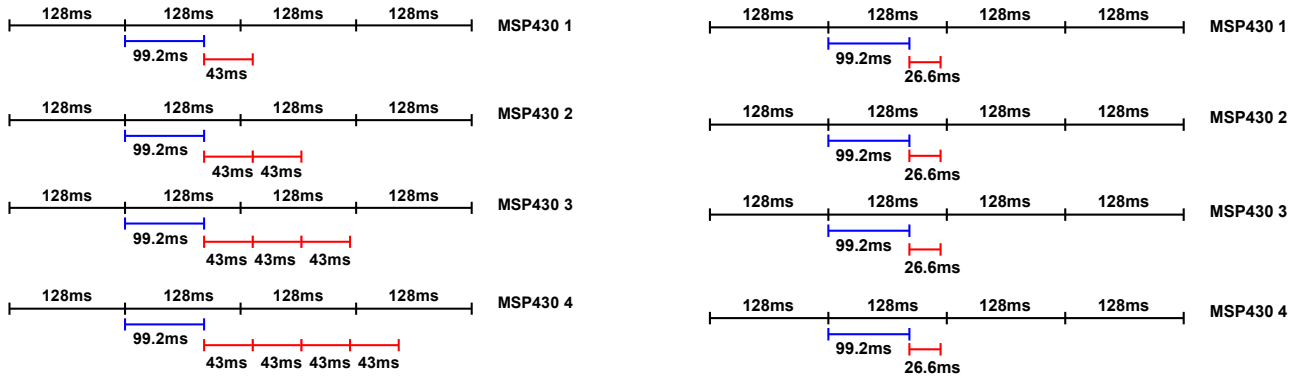
Having profiled the most important execution and communication components of the SSL application we can sketch its real execution sequence on the mPlatform architecture by using data from Figures 5 and 7. Figure 8 shows these execution sequences when two different communication channels are used: the I^2C bus (Figure 8(a)) and our CPLD-based data bus (Figure 8(b)). In both cases since we have to sample audio data at a frequency of at least $4KHz$, it takes $128ms$ to acquire 512 samples. Note, that data sampling and data processing can overlap since the sampling is handled automatically by the embedded DMA controller on the MSP430 processor.

As soon as the collection of 512 samples is completed, the FFT software routine has to be executed and 2048 bytes have to be sent over the communication channel. The FFT execution time, independent of the communication bus used, is $99.2ms$. In the case of the I^2C bus, however, it takes $43ms$ to send 2048 bytes according to eq. (1). Note that the total time of executing the FFT and communicating over the I^2C bus exceeds the data sampling time $128ms$. What is even worse is that in the case of the I^2C bus only one MSP430 board at a time can use the communication channel. As a result of this the total time for processing and sending the audio data on all four MSP430 boards is equal to: $99.2ms + 4 * 43ms = 271.2ms$, as shown in Figure 8(a), which is more than twice the audio data sampling time of $128ms$. In practice, this means that either the sampled data has to be buffered, re-



(a) (b)

Figure 7: (a) High Level Description of the Sound Source Localization (SSL) application. (b) Measured FFT execution time for different input sizes on both the MSP430 and the OKI MLQ675003 processors.



(a) (b)

Figure 8: Execution delay sequences for the sound source localization application when (a) the I^2C bus and (b) the CPLD bus is used

sulting in none real time SSL, or two blocks of data have to skipped for every block of data sampled, resulting in decreased SSL accuracy.

On the other hand, when the proposed CPLD-based data bus is used, the gain is twofold:

1. The communication delay for sending 2048 bytes is reduced from $43ms$ to $26.6ms$ according to eq. (2).

2. The data bus can now be shared due to the TDMA protocol implementation on the CPLD. This means that all four MSP430 processors deliver their 2048 bytes to the CPLDs that they are interfaced to at the same time. The TDMA protocol running on the CPLDs time-multiplex the data from all four processors at the expense of a negligible delay overhead according to Table 1.

As a result, as Figure 8(b) shows, the total time for processing and sending the audio data on all four MSP430 boards is now equal to $99.2ms + 26.6 = 125.8ms$. Note that this time is now less than the data sampling time of $128ms$, enabling mPlatform to meet the minimum real time requirements of the SSL application.

5. DISCUSSIONS

As our experiments have demonstrated, the implemented communication channel architecture provides up to 3 orders of magnitude higher throughput on the bus and up to 42% or 92% less end-to-end communication delay for MSP430 and ARM7 respectively, compared to the traditional I2C bus approach. The asynchronous interface between the bus controller and the local CPUs allows the communication channel to work with a range of processors at their peak performance. The TDMA protocol enables the efficient sharing of the bus with negligible delay overhead in the end-to-end communication, even when up to 10 time slots are used. The performance and flexibility of the communication architecture are obtained at a small increase in the overall power consumption by using low power Complex Programmable Logic Devices (CPLD).

5.1 Limitations and Future Work

The current CPLD bus implementation uses a continuous clock, so the TDMA protocol runs continuously even when there are no messages to be sent over the bus. This results in wasted energy, since, according to Figure 6(b), the TDMA protocol power consumption is almost independent of the bus activity. We can reduce the time when the TDMA protocol is active, hence the energy consumption, by running the TDMA protocol only when at least one CPLD has data to be transmitted. This *on demand* TDMA implementation will require modifications such as a clock that can be turned on and off instantaneously and an asynchronous FIFO implementation inside the CPLD.

The TDMA protocol implementation currently assigns a single time slot to each module. However, this can lead to inefficient use of resources if several high speed processors and slow processors share the same stack, since the current design allocates the bus equally across all the modules. We can mitigate this by extending

the current protocol to support multiple slot assignments per module. With this extension, we can allocate time slots among different modules based on their communication needs.

Even though the performance of our CPLD design allows the transparent multiplexing of the data bus, the CPU interface could be enhanced so that applications can further optimize the performance of the data bus. When, some of the modules in the stack do not need to use the communication channel, they could surrender their time slots. In practice, this would result in reducing the total number of slots used in the stack. The smaller the number of slots in the stack the less time a module has to wait for its slot to be activated.

The communication protocol that is currently used to transfer data across the CPLDs at different hardware modules uses 24-bit wide data packets where the first byte is always dedicated to addressing information. This creates a constant overhead of 33% for every communicated data packet. By changing the protocol specification to support start and stop packets, in the same sense that the start and stop conditions are used in the I^2C bus would allow us to transmit the addressing information only once for every burst of data. This would require a slightly more complex state machine implementation in the CPLD.

5.2 Related Work

A number of sensor node architectures have been developed over the last six years. The design goals behind each of these architectures are different. The Berkeley and Telos motes [12, 13], a widely used family of platforms, target small size and low power consumption. They are built around an 8-bit AVR or 16-bit MSP430 processor and have been extensively used in several different types of environmental monitoring applications. BTNodes [3] share the same design principles as the Berkeley motes while enabling Bluetooth based communication. Other platforms, such as imote2 [1] and XYZ [9], aim to provide ample processing and memory resources to facilitate prototyping and experimentation at the expense of increased power consumption. In another design paradigm, CPUs with different processing and power characteristics are co-located on the same sensor node [11, 18]. This architecture allows the design of sophisticated power management schemes that enable the execution of processing hungry applications on power limited devices.

As a modular platform, the mPlatform architecture is similar to the PASTA nodes [17], the MASS architecture [4], the sensor node stacks developed at MIT [2, 7] and the modular platform from Universidad Politecnica de Madrid [14]. The main differences of the mPlatform over the existing stack-based sensor nodes are twofold. First, mPlatform enables real-time event processing by incorporating a low power processor at each layer in the stack. Because of this, the sensors at different layers in the stack do not have to compete with all the other layers for the resources of a central processor. Second, an efficient, reconfigurable CPLD-based communication channel allows the different processors in the stack to share data almost simultaneously and at a significantly higher speed than the traditional communication channels already used.

6. REFERENCES

[1] R. Adler, M. Flanigan, J. Huang, R. Kling, N. Kushalnagar, L. Nachman, C. Y. Wan, and M. Yarvis. Intel mote 2: an advanced platform for demanding sensor network applications. In *SenSys 2005*, pages 298–298, New York, NY, USA, 2005. ACM Press.

[2] A. Y. Benbasat and J. Paradiso. A compact modular wireless sensor platform. In *IPSN, SPOTS track*, 2005.

[3] J. Beutel, O. Kasten, F. Mattern, K. Romer, F. Siegemund, and L. Thiele. Prototyping wireless sensor network applications with btnodes, 2004.

[4] N. Edmonds, D. Stark, and J. Davis. Mass: modular architecture for sensor systems. In *IPSN 2005*, page 53, Piscataway, NJ, USA, 2005. IEEE Press.

[5] C. Han, M. Goraczko, J. Helander, J. Liu, B. Priyantha, and F. Zhao. CoMOS: An Operating System for Heterogeneous Multi-Processor Sensor Devices. In *MSR-TR-2006-117*, 2006.

[6] J. King, R. Bose, S. Pickles, A. Hetal, S. Vanderploeg, and J. Russo. Atlas a service-oriented sensor platform. In *Proceedings of SenseApp*, 2006.

[7] M. Laibowitz and J. A. Paradiso. Parasitic mobility for pervasive sensor networks. In *Pervasive*, pages 255–278, 2005.

[8] I. Locher, S. Park, and A. S. M. B. Srivastava. System design of ibadge for smart kindergarten. In *Design Automation Conference(DAC)*, 2002.

[9] D. Lymberopoulos and A. Savvides. XYZ: A motion-enabled, power aware sensor node platform for distributed sensor network applications. In *IPSN, SPOTS track*, 2005.

[10] S. Matic, M. Goraczko, J. Liu, D. Lymberopoulos, B. Priyantha, and F. Zhao. Resource modeling and scheduling for extensible embedded platforms. In *MSR-TR-2006-176*, 2006.

[11] D. McIntire, K. Ho, B. Yip, A. Singh, W. Wu, and W. J. Kaiser. The low power energy aware processing (leap)embedded networked sensor system. In *IPSN 2006*, pages 449–457, New York, NY, USA, 2006. ACM Press.

[12] MICAZ. Wireless sensor node platfrom. http://www.xbow.com.

[13] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling ultra-low power wireless research. In *IPSN, SPOTS track*, 2005.

[14] J. Portilla, A. de Castro, E. de la Torre, and T. Riesgo. A modular architecture for nodes in wireless sensor networks, in journal of universal computer science, vol. 12, no. 3, 2006.

[15] K. Ryu, E. Shin, and V. Mooney. A comparison of five different multiprocessor soc bus architectures. In *EUROMICRO Symposium on Digital Systems Design, pp. 202-209.*, September 2001.

[16] A. Savvides and M. B. Srivastava. A distributed computation platform for wireless embedded sensing. In *ICCD, Germany*, 2002.

[17] B. Schott, M. Bajura, J. Czarnaski, J. Flidr, T. Tho, and L. Wang. A modular power-aware microsensor with $> 1000x$ dynamic power range. In *IPSN, SPOTS track*, 2005.

[18] Stargate. Wireless single board computer. http://www.xbow.com/products/xscale.htm.

[19] S. Yamashita, S. Takanori, K. Aiki, K. Ara, Y. Ogata, I. Simokawa, T. Tanaka, K. Shimada, and H. Kuriyama. A 15x15mm, 1ua, reliable sensor-net module: Enabling application-specific nodes. In *IPSN, SPOTS track*, 2006.

[20] C. Zhang, Z. Zhang, and D. Florncio. Maximum likelihood sound source localization for multiple directional microphones. In *ICASSP*, 2007.