Multi-Core and Real-Time - Making It Work

Glenn Steiner
Sr. Manager, Xilinx, Inc.

Endric Schubert, Ph.D.
General Manager, Missing Link Electronics

ABSTRACT

The design of multi-channel real-time control systems is a challenging combination of dealing with high software complexity and strict requirements for timing behavior. We demonstrate how such systems can benefit from advanced hardware / software partitioning which maps certain real-time functionality into dedicated hardware for parallel processing while utilizing multi-core CPUs for the software. This paper also discusses hardware / software co-design options to take advantage of a new class of compute devices which integrate Application Specific Standard Processors (ASSP) within programmable logic devices.

1. INTRODUCTION

The definition of Real-Time means that a system must respond to an event within a fixed, predetermined time. This becomes difficult when the response time is short and safety relevant, for example in anti-lock braking systems, and/or multiple events must be processed leading to multiple responses, each with a different timing requirement. Because these so-called Real-Time Control Systems (RTC) must integrate computational and physical processes [2] their design complexity is dominated by managing time and concurrency in the computational part [3]. Examples are new automotive applications such as telematic systems for cars which bridge into the safety relevant domain, or next-generation industrial control systems which integrate the control loop with motor control and safety supervision.

Nowadays reasonably priced, high performance multi-core processors are becoming more common. However, multi-core processors, especially when running a high-level operating systems such as Linux tend to be ineffective at providing real-time performance. What makes an applications-class multi-core processor great for performance makes it poor for real time applications, in general: Multi-stage pipelines take extra time to flush; memory-management units (MMUs) and paging operating systems require Table-Walks, these tables may not be residing in cache which increases the response time; not all caches are lockable and locking wastes precious cache space and decreases system performance.

Interrupt service architectures are optimized for multi-tasking but not real-time behavior, vectorizing interrupt controllers may not always be available. Additionally, Interrupt Service Routines can fall out of cache. In short, while multi-core processors come with the ability to save large amounts of state information this takes compute time.

Field-Programmable Gate-Arrays (FPGA) can offer very interesting design choices for such RTC: Dedicated hardware co-processors for Asymmetric Multiprocessing (AMP) architectures combined with programmable logic are enablers for providing real-time performance with multi-core processing systems. With a new class of devices which integrate Application-Specific Standard Processors into the FPGA fabric not only a wide range of I/O standards are directly supported, but they can also be used to implement the demanding compute and real-time requirements.

However, to the system designer who is used to a very software centric view (with limited concurrency to handle and very different timing aspects to deal with) the architectural choices can be very overwhelming.

Therefore, we will examine techniques for increasing real-time capability in multi-core systems including the use of coprocessors, Asymmetric Multiprocessing (AMP) in FPGA programmable logic. This paper first presents the basics of RTC including multi-channel Real-Time Control Systems. We will then discuss methods for migrating certain real-time functionality into dedicated co-processing hardware blocks. We finally demonstrate how a new class of devices, so-called extensible processing platforms, can offer efficient hardware / software co-design options.

2. BACKGROUND

Figure 1 shows an example of the building blocks of a basic control system. The aspect of the physical system to be controlled, which is a state variable like rotational speed, temperature, flow, etc. can be seen on the right hand side. This physical value is either directly measured by sensors or, if this is not possible for some reason, calculated from other sensors data, and thus obtained indirectly.

Once the value of the physical variable is known to the system, it is compared to an externally given setpoint, which is

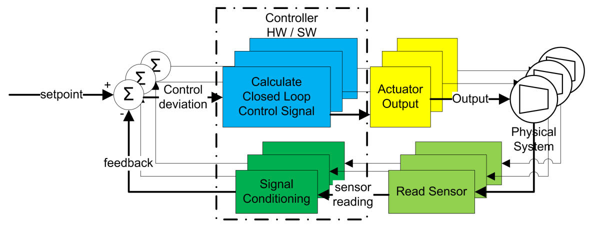

Figure 1: Basic Structure of a Multichannel Control System

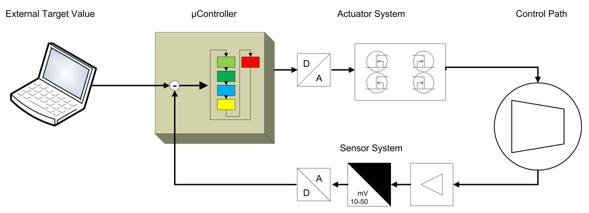

Figure 2: Architecture of CPU-Based Control System

the target value for this variable. This comparison is done by computing the difference, the so called control deviation, between setpoint and current value of the state variable. The control deviation is forwarded to a controller, which then adjusts its outputs according to its inherent control strategy to bring the system more closely to the target setpoint. This output controls an actuator which influences the physical behavior of the system under control. In most systems the setpoint is not fixed, but can be influenced by the user of the system or some control software.

Also there may be multiple channels with different physical values and different dynamic behavior to be controlled. This is indicated by the stacked layers in Figure 1.

One challenge in designing a closed-loop control system is the selection of the right control algorithm for the application including the right set of parameters for this algorithm. An extensive design exploration trying out different control algorithms with different parameter sets and different speeds of execution is crucial for finding the optimum solution.

2.1 Traditional Approach

Let's first have a look on how to implement such a system using a microcontroller to implement a closed-loop Real-Time Control System together with the necessary user application in software. The basic layout of such a system is shown in Figure 2.

As the microcontroller has to cope with both, the continuous control and the user interaction, the software has to implement both parts. Being only capable of executing operations sequentially, a microcontroller can only work on one task at a time. Hence the inherently parallel problem — maintaining certain target values while processing a user application — has to be serialized first. This can be implemented by a program structure as shown in Figure 3.

The calculation of the controller output has to happen quasi-

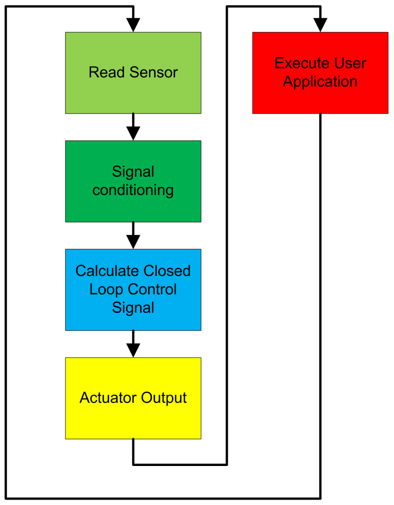

Figure 3: Flowchart of a CPU-Based Control System

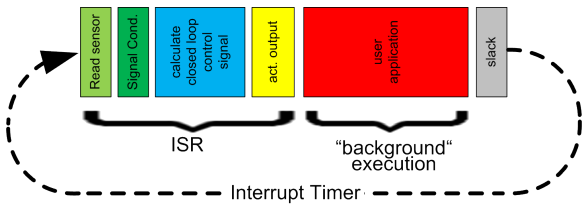

Figure 4: Timing in an interrupt driven control application

continuously, or at least on a very regular deterministic basis in respect to timing. To achieve this, the complete control algorithm including data acquisition and data output is done in an interrupt service routine that is called regularly using a timer interrupt. This timer interrupt divides the program execution into time slices. In every time slice the control algorithm has to be performed to acquire sensor measurements, compute the next control value and propagate this value to the output stage. When these tasks are finished, the user application may use the processor for the rest of the time slice. This concept is similar to time-triggered architectures [1] and enables the design of deterministic, distributed systems.

In Figure 4 a visualization is given on how the flowchart of Figure 3 is handled in the microcontroller using interrupt service routines (ISR) and "normal" program execution.

The time slices must be large enough such that the entire control algorithm is guaranteed to be completed within one slice leaving enough computing time to execute the "background" user application. Between the execution of controller code and the user application there is some time needed for the context switch. As the different portions of the computations within each time slice are not exactly the same in each cycle, additionally there has to be some slack for worst case execution times to allow the algorithm to "breathe". Depending on the chosen input and output devices (mostly analog-to-digital and digital-to-analog con-

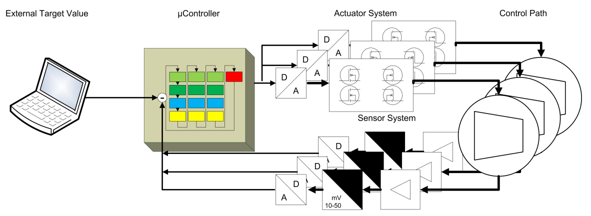

Figure 5: CPU-Based Multichannel Control System

verters) there may be some additional latency for data acquisition and data output.

As accurate timing is crucial for the system to work, the first step in the software development process is to decide on a specific control algorithm and then to validate the implementation with respect to timing. Furthermore, some interfaces (universal asynchronous receiver/transmitter (UART), Serial Peripheral Interface (SPI), Inter-Integrated Circuit Bus (IIC), etc. may be supported directly by the microcontroller, while others may need to be implemented via software routines. Typically, this depends on the microcontroller used and can add to the design complexity.

Once the definition and implementation of the control algorithms and the interface routines are done, it is necessary to estimate the maximum worst-case execution time and, thereby, to define the minimum cycle time. This cycle time must be sufficiently long (including slack), as some background processing time must be available to run the user application. Even though this user application is not as time critical as the control algorithm, there must be enough slack in processing time to achieve sufficient responsiveness to the users input.

To the novice control system designer it may sound strange to run the actual user application in the background while having the timer interrupt routine doing a major part of the work. Nevertheless one gets used to write software code in such manner and – after some hands-on experience – most likely gets such code to work after all.

However, it remains difficult to write such code and a lot of caution is necessary to keep oversight over all the different timing and execution layers. One popular pitfall, for example, is the handling of nested interrupts which is necessary as the software is processing the timing interrupt handler for quite a bit of time. Even with a carefully designed system and a powerful microcontroller it is sometimes not possible to deliver a repetition rate for the closed-loop control faster than in the low kHz range. This rate falls approximately proportionally with the number of channels to be controlled.

2.2 Multichannel Control Systems

Very often, multiple channels must be controlled concurrently by one single control unit. Examples are multiple motors or certain lighting applications synchronized to multi-axis motion control, etc. Such a multichannel closed-loop Real-Time Control System is outlined in Figure 5.

Under normal circumstances it is difficult enough to serialize a closed-loop control system for a single channel onto a sequential microcontroller and maintain all the real-time constraints. But having multiple channels with different time domains can make things really complicated. For example, because of the different closed-loop control paths running at different speeds, the time slices for the interrupts have to be determined similar to a least common denominator in terms of timing.

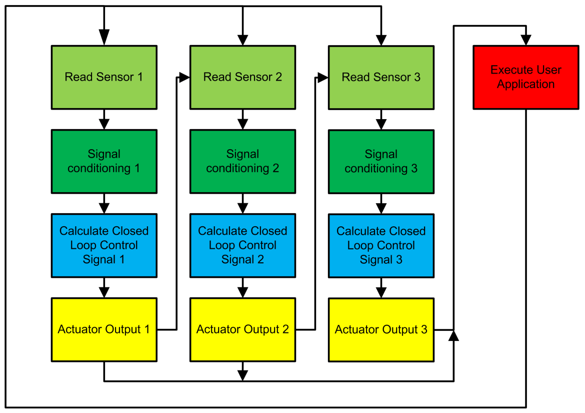

Figure 6: Concurrency in a Multichannel CPU-Based Control System

Shown in Figure 6 there is no linear straight forward software flow anymore. Quite contrary, in every interrupt cycle a decision has to be made which task needs to be computed next and which task won't need attention for the moment. The calculation of the worst case execution time ends up in a quite complex design challenge [3], e. g. using queuing theory to be handled correctly. In addition to the increased complexity in software development, this implementation inevitably introduces a considerable amount of jitter into each of the individual closed-loop control channels.

3. PARALLELIZED COMPUTATION

Instead of serializing the entire control system, one can decouple and segregate the different tasks of the closed-loop control system and run each task independent from each other, concurrently in modular hardware. This is an obvious approach to get out of timing trouble.

In a true parallel system, the user application and the control loop computations can be independent from each other. They may have to exchange data for current setpoints and current system states, for example, but this does not require any tight coupling. Instead it can happen asynchronously through a defined interface such as message passing or shared memory.

To the contrary, in the microcontroller-based approach these two independent tasks become very tightly coupled and influence each other very strongly, because computation is done sequentially with one task possibly blocking the resources of the other task.

One solution to achieve such parallelism is to use Field Programmable Gate-Arrays (FPGA) and to implement each different task of the control system using a separate hardware module which all run concurrently.

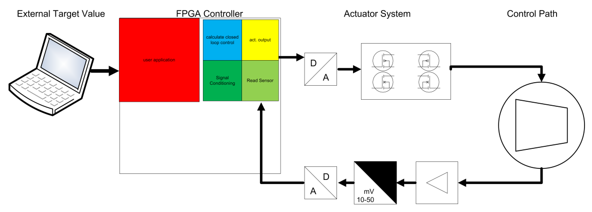

Figure 7: Architecture of an FPGA-Based Control System

FPGAs can be seen as flexible hardware processing devices that may be configured to execute almost any digital computation. The possibilities range from simple glue logic to advanced microprocessor designs that can be implemented in FPGA logic. Due to their programmability and the flexible interconnection inside it is easy to maintain the inherent parallelism of concurrent tasks. One application very well suited for FPGA logic is to run finite state machines as they are fundamental to many of the control problems described above. Additionally, with the possibility to embed one (or more!) microprocessors into an FPGA, sequential software applications can also be handled efficiently by FPGA devices.

During the design phase of a Real-Time Control System processing can be divided into several tasks with well defined interfaces to each other. In our example, the tasks of a closed-loop Real-Time Control System (as shown in Figure 2 and Figure 3) are: data acquisition by reading the sensors, signal conditioning, computation of the output control signal, data output to the actuator under control, application software for user interaction.

The first four tasks can be implemented via independent hardware modules realized directly in configurable FPGA logic, while the application software stays on a sequential processor which is now exclusively available to the user application.

As you can see in Figure 7, four different tasks are now implemented in hardware modules that run independently and parallel within the FPGA. Even though there is of course some synchronization and communication between the hardware modules, this is done inside FPGA logic and not by a sequential program as in Figure 2. As a result, each individual module can now be modified, copied or replaced individually, without influencing the timing behavior of the other modules.

Modularization also enhances design efficiency as it enables concurrent engineering and early testing of the system. We will now have a closer look at the different hardware modules to build such a FPGA-based Real-Time Control System.

3.1 Data Acquisition Module

The Data Acquisition Module together with the connected sensors is responsible for measuring the values of the system's physical state in each control cycle. It can be realized as a simple state machine that acquires data from an external sensor and stores it into a register over and over again. The acquisition speed can be provided externally to the module to synchronize the complete system.

Depending on the sensors in use it may be necessary to have a certain protocol to access the sensor. For example, it is very common to connect sensors via analog-to-digital converters (ADC) which again support connectivity via SPI or IIC interfaces. Optionally, very fast ADCs can be connected via the FPGA's LVDS interfaces. Sometimes special protocols are required to interface to the sensors. Such protocols can also be implemented directly within the FPGA logic, so that the entire data acquisition is completely contained within the Data Acquisition Module. The advantage is that changing sensors has little or no impact on the rest of the system. This significantly reduces the design risks for late changes, for example, and is one of the key benefits of using FPGAs for Real-Time Control System.

3.2 Signal Conditioning Module

The Signal Conditioning Module converts the acquired data into an internal data format suitable for further processing. Data formats can in many cases be based upon a numerical fixed point representation which is very suitable for FPGA logic. Depending on the measured physical values it may also be necessary to perform certain data pre-processing before passing the value to the control algorithm module. Such digital signal processing operations may include the computation of a derivative, for example to transform a rotational speed of a wheel into a unidirectional speed over ground. Other examples of data pre-processing include low-pass filtering of the signal to cancel out distortions.

As we will explain in the next section, for most of those signal conditioning algorithms there exist pre-designed signal processing hardware blocks that can be combined to quickly build very complex pre-processing. Using those hardware blocks, for each sensor input (or more precisely, for each Data Acquisition Module) there can be a corresponding, independent Signal Conditioning Module.

3.3 Control Algorithm Module

A key portion is the implementation of a closed-loop control algorithm. The control algorithm has to compute the next output value for the controlled system depending on the current sensor feedback and must be appropriate for the given control system application behind.

The careful selection and design of the control algorithm is critical to the quality of results and the robustness of the Real-Time Control System. Now, using FPGA technology, the different tasks are decoupled, independently implemented in hardware modules and it becomes much easier to explore different control algorithm strategies using different configurations without interfering with the overall system's timing behavior.

There exist plenty of different control principles that are more or less suited to a concrete problem and, normally, it is left to the designer to deliver the best suited algorithm for the specific problem. Even after deciding on an appropriate control algorithm, the design exploration is not finished, yet. For most of the algorithms there exist different hardware implementation alternatives. Depending on the Real-Time Control System's cycle time, and utilizing the

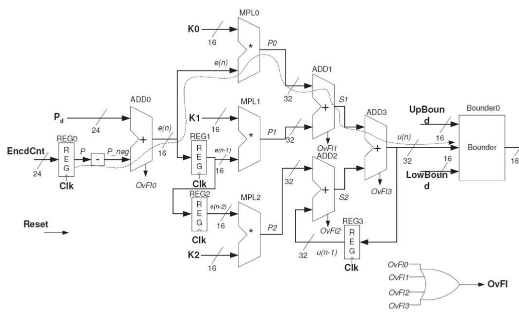

Figure 8: Speed-Optimized Control Algorithm Module for PID Control

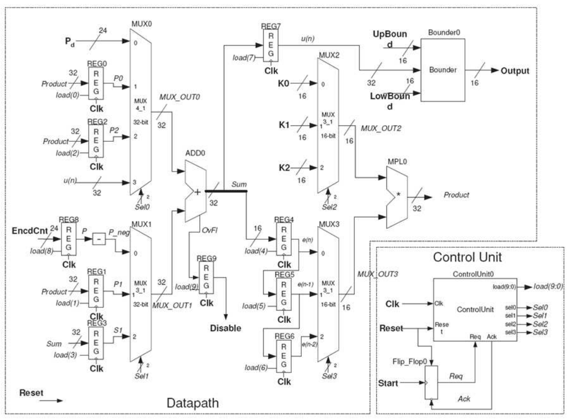

Figure 9: Area-Optimized Control Algorithm Module for PID Control

ability to trade-off between area and speed in an FPGA implementation, an algorithm can be implemented either in a parallel manner to achieve maximum computation speed or in a more sequential fashion to save FPGA resources. As an example, a comparison of different hardware implementations of the popular PID algorithm can be found in [6].

One - fast - implementation shown there is given in Figure 8 which needs only one single clock cycle to calculate the next control output from the measured input. Compared to a microprocessor-based implementation this is an incredibly fast computation and, hence, can lead to incredibly short cycle times. This aspect highlights yet another benefit of FPGA-based RTC implementations: The control cycle times can be sped-up by orders of magnitude.

Another – slower but area-optimized – implementation of the same PID controller is shown in Figure 9.

3.4 Data Output Module

In its' structure the Data Output Module is very similar to the Data Acquisition Module, except that the Data Output Module takes the output of the Control Algorithm Module, possibly does data pre-processing and then drives the actuators. Most often this is implemented as a direct pulse-width-modulation or as a sigma-delta converter feeding a simple power output stage like a full- or half-bridge driver. In other system configurations, the data output may drive an external digital-to-analog converter (DAC), connected via SPI or similar protocols. Data pre-processing may be as simple as clipping, to limit the output to a certain valid range. Or, it may involve sophisticated signal processing implemented in FPGA logic. Because of an FPGA's superior processing power digital signal processing algorithms which can be implemented as a DSP program also fit most FPGA devices.

3.5 User Application Software

The last task to be implemented is the user application program which interacts with the user and propagates high level parameters from and to the control system. This portion is best suited to be run as a software program on a microprocessor. In contrast to the traditional microprocessor-based approach, almost all time-critical and performance hungry control and interface tasks are offloaded from the microprocessor into dedicated hardware modules. Now, the microprocessor is available almost exclusively to the user application software.

The effect of such a system partitioning is that the closed-loop control part and the real-world interface with hard real-time requirements is now completely decoupled from the user software which normally has far less stringent requirements – at least as far as real-time behavior is concerned. The embedded software programmer can now focus on the functionality of the software instead of struggling with complex interrupt timings to force an inherently parallel problem into a sequential compute scheme.

3.6 Scalability Aspects

Once a control algorithm is implemented for one single channel, it is easy to scale the Real-Time Control System to multiple channels. This is as easy as instantiating another set of hardware modules for the Data Acquisition Module, the Signal Conditioning Module, the Control Algorithm Module and the Data Output Module. Assuming sufficient logic resources in the FPGA, there is no impact on timing and controller cycle time, regardless whether a single or several channels must be controlled. As an example, a three-channel FPGA-based Real-Time Control System is shown in Figure 10.

Often, a multichannel Real-Time Control System has very different real-time requirements for each different channel. Sometimes, the real-time constraints of each different channel are orders of magnitude apart from each other. In traditional microprocessor-based Real-Time Control System design this imposes a serious challenge as it either complicates the interrupt service routines or requires that slower control channels must use the cycle time of the fastest channel. However, in FPGA-based Real-Time Control System each channel operates in parallel and, thus, can be optimized independently from the other channels.

Therefore, it is possible to implement the most timing constrained channel using a speed-optimized control loop while less demanding channels use more resource optimized control loops.

Again, these – and other – optimizations become possible because of the parallel processing capabilities of FPGA technology which allows to decouple each task of a closed-loop

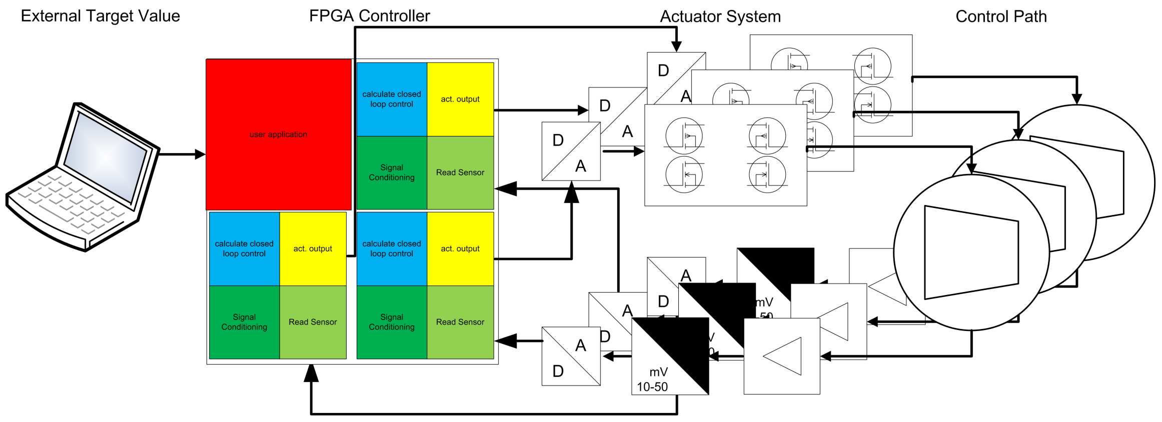

Figure 10: FPGA-Based Multichannel Control System Using Hardware Replication

Real-Time Control System and to implement each task in an independent hardware module.

4. EXTENSIBLE REAL-TIME PROCESSING

Processing solutions continue to converge: Because of Moore's Law it became feasible to integrate more and more functionality into a digital circuit. As a result, specialized co-processing and dedicated I/O was added to General Purpose Processors (GPU) effectively creating the so-called Application Specific Standard Processors (ASSP).

FPGAs have long been used as a companion for a GPU, to implement special purpose processing and/or dedicated I/O connectivity. With more FPGA resources available, eventually GPU were embedded inside the FPGA - initially as resource efficient hard blocks, later as more flexible soft core implementations. Not being limited by chip-to-chip interfaces between FPGA and GPU, this enabled a more flexible and much tighter integration of GPUs with programmable logic, introducing new co-processing concepts for hardware / software partitioning [4].

Recent advancements are a logical next step: Embedding ASSPs into FPGAs. Integrating an ASSP as a hard block into an FPGA has some significant effects: From a circuit resource point-of-view it becomes very advantageous to integrate powerful multi-core CPUs, together with integrated memory interfaces to support a variety of memory solutions: DDR-2, DDR-3, LPDDR2, etc. Rich I/O interfaces can be made available which normally would have to be implemented in programmable logic. For example, specilized I/O such as SPI, I2C, CAN, UART, GPIO, SDIO, USB, GigE are readily available. These modern device concepts even support Analog-to-Digital conversion, such as the Agile Mixed Signal I/O [5].

The result is a new class of processing devices which on one hand offer a complete processing system with dedicated peripherals, defined memory map and which can boot software without programmable logic involvement. Thereby, these device support industry standard development tools and processes and can run industry standard operating systems and application software.

On the other hand, these device provide a more flexible and much tighter link between the software - running on the ASSP - and the hardware - inside the programmable logic - offering higher bandwidth and lower latency for hardware-to-software and software-to-hardware communication. This means that processing systems can easily be extended with dedicated I/O and application specific co-processing.

5. CONCLUSION

We have shown techniques for migrating certain real-time processing functionality from software to customized hardware blocks. Many of those intensive data processing tasks with tight timing constraints can run much better in the parallel operating hardware domain than in the sequentially running software domain. This puts the burden of processing where it belongs and results in a more scalable implementation.

We have also shown how a new class of processing devices supports this design methodology. Because these devices integrate ASSPs inside FPGAs a more flexible and much tighter communication between the hardware and the software domain is possible, effectively meeting the requirements for implementing Real-Time Control System. Architectures which utilize the afore mentioned techniques for real-time processing in customized hardware blocks become feasible and cost efficient.

6. REFERENCES

[1] H. K. et al. The time-triggered architecture. January 2003.

[2] E. A. Lee. Computing foundations and practice for cyber-physical systems: A preliminary report. *UCB/EECS-2007-72, UC Berkeley*, May 2007.

[3] P. Marwedel. Embedded and cyber-physical systems in a nutshell. *DAC.COM Knowledge Center Article*, 2010.

[4] E. Schubert. Extend the powerpc instruction set for complex-number arithmetic. *Xilinx XCELL*, 66, 2008.

[5] Xilinx, Inc. *White Paper (WP369): Extensible Processing Platform.*

[6] W. Zhao, B. H. Kim, A. C. Larson, and R. M. Voyles. Fpga implementation of closed-loop control system for small-scale robot. In *Proceedings of the 2005 Int. Conference on Advanced Robotics*, July 2005.