Multiprocessor and Operating System Design for Signal Processing on an FPGA

Fernando Martinez Vallina, Nathan Jachimiec, and Jafar Saniie
Department of Electrical and Computer Engineering
Illinois Institute of Technology
Chicago, Illinois 60616

Abstract

Multiprocessor systems are a way to increment the flexibility and performance of embedded computing engines. These systems, which rely on application parallelization, require a different design paradigm than System-on-a-Chip (SoC), which does not scale to multicore systems. This paper presents a single FPGA multiprocessor development platform with network style interconnects for signal processing applications. Along with the computing platform, a custom built operating system is presented. The operating system presented in this work is called LightOS. LightOS is a lightweight operating system targeted at signal processing applications. The multiprocessor platform is functionally verified through the use of an FFT benchmark application. Furthermore, the study shows that the embedded platform combined with LightOS is suitable for signal processing algorithm development on an FPGA.

1. Introduction

Parallel computing systems have long been used as a means of accelerating program execution in the context of increasing problem size (i.e., data, computational complexity or both). These systems have been traditionally implemented either on high-end multiprocessor computing systems available from companies like IBM, HP and Sun, or on Linux clusters built from commodity of the shelf (COTS) computers. Both implementation styles for parallel computers have until recently been limited to a single processor per silicon die. New trends in integrated circuit fabrication are allowing for multiple processing cores to be implemented on the same die. This in turn is allowing for the characteristics of parallel computing systems to be ported into the embedded computing space [1][2].

This study presents the Embedded Multiprocessor Prototyping and Development (EMPD) Platform, which allows for full exploration of the system design space from both a hardware and software perspective. The EMPD system is based on a 32-bit RISC processor without the use of custom acceleration units typical of embedded platforms. The reason for this is to minimize the initial complexity while the EMPD is verified for functional correctness. EMPD system design is based on a software centric approach with minimal hardware design requirements. This allows the platform to attain flexibility and adaptability. Following EMPD design cyle, the FFT benchmark application is completely decomposed in software with floating point units being the only accelerators added to the system at this time.

In comparison to traditional Systems-on-a-Chip (SoCs), an EMPD based system has considerable software overhead. Although the EMPD has several advantages over a traditional SoC, this platform is heavily constrained in terms of physical resources. Among these resources, on-chip memory is the limiting factor for overall system flexibility and performance. Off-chip memory is available on the EMPD, but it is reserved exclusively for the main processor. The reason for this being, that the current EMPD has a single external memory access interface. If multiple processors access this memory for instruction/data, the overall net effect is that the system will be halted while the direct memory access controller resolves all service requests. As a result, all computation processors have their own private data/instruction memory. The current EMPD platform reserves 16KB of internal memory for each computation processor.

Currently, the application independent software on the EMPD platform consists of a minimal operating system with message passing capabilities on the computation processors and uCLinux [5] executing on the main processor. This two tier approach between computation and main processors allows the system to protect itself from data corruption as well as increasing the overall flexibility of the platform. Also, the user transparent interaction of the two operating systems allows for faster application development, system maintenance, and in field modification.

2. EMPD System

The EMPD experimental platform is built upon the Xilinx Virtex-IIPro FPGA [3] and the Xilinx MicroBlaze softcore processor [4]. This softcore allows for the EMPD to be tested under different resource allocation scenarios including the lack of and availability of hardware floating point support in each core. Although hardware support for floating point operation is always preferable in signal processing applications, it is still useful to analyze the

1-4244-0941-1/07/$25.00©2007 IEEE

impact of this unit in an embedded multiprocessor context where it directly affects the available physical resources of the system. Implementation results show that hardware floating point support at each core of the EMPD reduces the memory requirements for all computation processors by 20%. At the same time, the unused resources left on the FPGA are only sufficient for the implementation of very simple accelerators. Systems requiring complex accelerators and floating point support will have to reduce the number of computation processors.

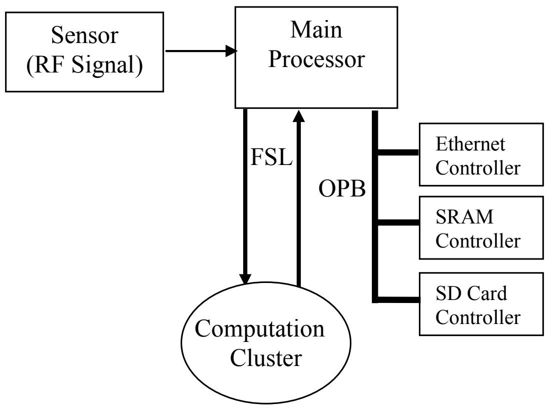

Figure 1. EMPD Architectural Overview

Figure 1 shows the architectural overview of the EMPD system. As previously mentioned, this is a tiered architecture in which only the main processor is readily visible to the programmer. The main and computation processors are all based on the MicroBlaze softcore processor. Both processor sets work in conjunction to carry out a user application. The responsibility of the main processor is to provide the overall input/output operation control, and to execute a complete multithreaded operating system capable of handling complex applications. The operating system used at this level of the EMPD is uCLinux [5]. The computational processors execute LightOS, which is a custom lightweight embedded operating system currently under development with the EMPD.

The interconnection of computation processors is predefined during system implementation. The EMPD supports computation processors connected in the star, mesh, and fully connected topologies as shown in Figures 2-4. Each of these topologies is created using the Xilinx point-to-point FSL link. Two of these links are used to create the bidirectional communication link between processors. For testing purposes of the EMPD, a broadband signal sampled at 100MHz is used.

3. µCLinux and LightOS

The two tier approach of EMPD requires a software platform capable of interfacing and handling the separate responsibilities of each tier. To accomplish this task, a dual operating system (uCLinux and LightOS) design was implemented.

3.1 µCLinux

An embedded distribution of Linux, uCLinux [5], is used in this project for the user environment. Specifically, EMPD uses the Microblaze uCLinux distribution developed at the University of Queensland [6][7]. In the EMPD platform, uCLinux serves the following functions:

- Provide file system support
- Provide network interfacing capabilities
- Provide scheduling and support services for user application
- Provide a common interface to the computation processors

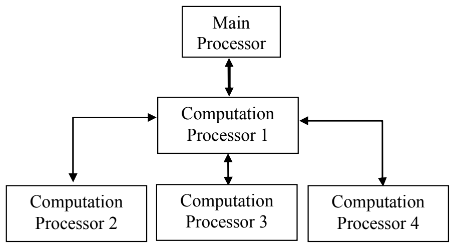

Figure 2. Star Interconnection

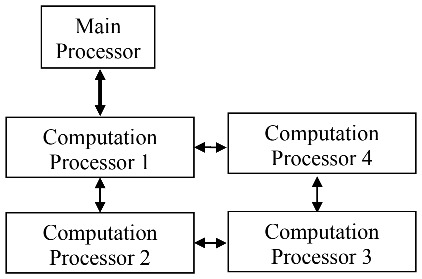

Figure 3. Mesh Interconnection

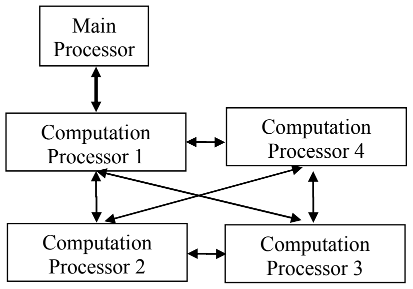

Figure 4. Fully Connected Interconnect

The communication between main and computation processors is based on a combination of message passing and shared memory. As shown in the interconnect figures 2-4, the main processor is only directly connected to computation processor 1. Therefore, computation processor 1 is the local gateway and coordinator for the computation cluster. In order to handle communication between the tiers, a special driver has been developed and added to the uCLinux kernel to facilitate application development. This driver views the computational cluster as both an I/O element and as a block of memory. The purpose of the dual mapping for the computational cluster is to accelerate data transfers between both tiers of the EMPD.

The I/O mapping of the computation cluster refers to the direct point-to-point communication between the main processor and computational processor 1. This link, which is implemented on two FSLs, is used for the transfer of commands, status messages and requests between tiers of the platform. Large data transfers are carried out through the use of a shared memory. The shared on-chip memory between tiers eliminates processor stalls during data transfers. Also, the main processor is freed to continue with other work until it is interrupted by the computation cluster.

3.2 LightOS

LightOS is a custom built operating system targeted at the computational cluster of the EMPD. The basis for this system is TinyOS [8][9], which has been used in distributed sensor networks and adhoc communication networks. As a result of the memory constraints on the EMPD platform, the current version of LightOS has the following basic characteristics:

- Programming threads
- Message passing between processors
- Recovery from a fault during data transmission

On top of these basic functions, there are two versions of LightOS executing on the EMPD. These versions are coordinator and worker. Both versions of LightOS support a computation thread to run user applications. At this time, the tasks of the computation thread are known at compile time and are not sent to each processor as in the traditional Message Passing Interface (MPI)[10].

The message passing support of LightOS is a small subset of the functions available in MPI. The basic send and receive functions are supported for the passing of integer and floating point data. More advanced data types, such as user defined structures, are currently not supported because of the memory constraints on the system. Unlike MPI, application code is precompiled and loaded into each processor along with LightOS. This helps conserve memory, but limits the runtime flexibility of the platform.

3.2.1 LightOS Coordinator

The LightOS Coordinator is executed on computation processor 1. Besides providing the basic services outlined above, the coordinator is responsible for communication to the main processor as well as distributing data to the computation processors. This version has additional thread support for handling data routing in the computation cluster and communication between the cluster and the main processor. A drawback of the coordinator role, is that processor 1 carries out almost no useful computational work for application acceleration. This means that the processor becomes an extra layer of overhead to the entire platform.

Although the actual interconnection between computation processors is fixed during system synthesis, LightOS has no notion of the exact topology before runtime. Among the tasks of the coordinator node is a heartbeat function. The heartbeat signal is sent from the coordinator out on all ports waiting for a response from the worker nodes. This allows for runtime determination of the interconnect topology, while making the software portable among EMPD realizations. Also, each returning heartbeat contains the identification of the worker processor, and the ID of all neighbor processors. After determination of the interconnect topology, the coordinator node (computation processor 1) will send a control message to each worker that will set its communicating behavior.

3.2.2 LightOS Worker

The LightOS Worker is a compressed version of the coordinator. Like the coordinator, the worker will send out a heartbeat signal during initialization to discover its neighbors. This signal is sent after the heartbeat of the coordinator is received. Once the communication port for the coordinator is established, the worker will probe all other ports for neighbors. After a time out, the worker will respond to the coordinator with its own ID and the ID of its neighbors, if any.

Neighbor data is important at the worker and coordinator levels for communication in topologies other than the star. For these other topologies, workers will act as pass through nodes for neighbor data not targeted for them.

The exchange of data is carried out with packets between all computation processors. Since all processors have the user application code, the messages only contain data. Each message is tagged with the ID of the originating processor, and is used for routing purposes. During the system initialization phase, it is the job of the coordinator node to inform all workers of their data source and data destination node IDs. Once the initialization phase is completed, worker processors are left to run autonomously based on the incoming data stream.

4. Benchmark Application

The Fourier Transform [11] is one of fundamental and most often used signal transformations for Digital Signal Processing (DSP) applications. The basic form of this transformation in discrete time for N data points is given by:

$$X[k]=\sum_{j=0}^{N-1}x[j]w^{jk} \qquad 0\le k\le N \tag{1}$$

$$w=e^{2\pi\sqrt{-1}/N} \tag{2}$$

For computational cost, equation 1 can be defined in terms of a matrix-vector product with an overall cost of O(N^2), where N is the number of points computed. This computation cost can be reduced to O($NlogN$) by means of the Fast Fourier Transform (FFT). As long as N is a power of two, the original N-point Discrete Fourier Transform (DFT) can be separated into two (N/2)-point DFT computations which are the basis for the FFT algorithm. This step is shown in equation 3.

$$X[k]=\sum_{j=0}^{(N/2)-1}x(2j)w^{2jk}+\sum_{j=0}^{(N/2)-1}x(2j+1)w^{(2j+1)k} \tag{3}$$

Following the expansion of Equation 1 into 3, it can be seen that each side of the DFT can be recursively expressed as a sequence of DFTs of smaller sizes. This in turn leads to an overall computation cost of O($NlogN$). This reduction in computational cost shows why the DFT is commonly implemented in terms of the FFT. Although there are many algorithms to carry out the FFT computation, the Cooley-Tukey algorithm [11] for the FFT provides a practical implementation that can be readily used on a sequential computer in either an iterative or recursive form. This FFT algorithm is common in software implementations on uniprocessor systems. For multiprocessing systems, the basic Cooley-Tukey algorithm has data dependencies which negatively impact the performance. For parallel implementation these dependencies are handled within the Binary Exchange (BE) Algorithm [11].

The BE Algorithm is a log(n) stage implementation of the FFT which requires data partitioning among available processors. The overall communication overhead depends on the level of data granularity after initial partitioning. This affects the performance of BE, which is a communication bound algorithm. A poor division of data will lead to most processors being idle and waiting for data from their neighbors to be able to fulfill the dependencies at each stage of the FFT computation.

In most cases, the partitioning of data is carried out by dividing the number of data points by the number of processors. For an N point computation on P processors, this will lead to N/P elements placed in the local memory of each processor. This coarse grain partitioning reduces the communication overhead of BE and helps to improve runtime performance. All data exchanges between processors are not eliminated, but are reduced to only the first log(P) stages. Where, each processor will receive the required data points from neighbor processors to satisfy the data dependencies of the stage. During the remaining log(N/P) stages of BE, there is no communication because all the required data is local to the each computational processor. Concurrent and parallel execution will occur between the processors until the final results are generated. At this point, there will be one more set of data transfers to place the results at their final destination.

The general characteristics of BE are summarized below:

- Number of execution stages = log(N)
- Number of communicating stages = log(P)
- Number of data points transferred = log(P)*log(N)
- Number of inter-processor messages = Plog(P)

5. Feasibility and Performance Evaluation

The purpose of the feasibility and performance benchmark study on the EMPD is to verify the functional correctness of the system for signal processing applications. For experimental evaluation, the EMPD is implemented on a Xilinx Virtex-2Pro FPGA with a clock frequency of 100MHz.

The results of the feasibility study show a functionally correct execution of the FFT algorithm with the platform average performance being given by the star interconnect topology. The results for this topology are shown in Tables 1 and 2. Table 1 shows the results of the EMPD without floating point support, and Table 2 shows the results with floating point support. Besides showing the impact of hardware acceleration on application performance, both of these tables show the impact of parallelization on performance as the amount of data to be processed grows.

A closer examination of Table 1, shows that for the case of no hardware support for floating point operations, the BE implementation of the FFT becomes a computationally bound application. For this type of application, parallelization results in a maximum speedup factor of approximately 3x in relation to a uniprocessor 128-point FFT. This speedup factor shows a sublinear speedup, which is typical in parallel applications as a result of synchronization and communication overheads. In terms of growing problem size from 16 points to 128 points, the parallelization of the FFT algorithm results in an approximately linear growth in execution time. This can be seen in Figure 5, which shows the execution time on a uniprocessor and a 4 processor system for different data sizes. This figure clearly shows that the uniprocessor system experiences a high rate of growth in terms of execution time as the number of data points computed increases. In contrast, the multiprocessor systems have a moderately slow execution time growth rate, which enables this system to efficiently scale for larger data sets.

Table 1. EMPD Star Interconnect, No Floating Point Support.

Computational Processors	Data Points (Execution Time in ms)			
	16	32	64	128
1	2.23	4.75	12.21	22.29
2	1.22	2.67	5.77	12.03
4	0.59	1.30	3.11	7.26

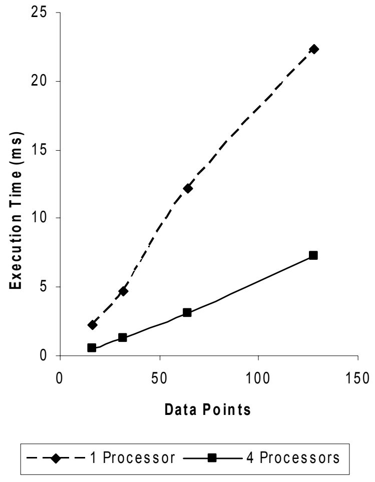

Figure 5. System Execution Times

Table 2 contains the runtime results for the test systems with hardware floating point support. A comparison to the values in Table 1, shows that adding this hardware accelerator increases the performance of all test systems. The removal of software emulation of floating point operations from the systems in Table 1 accounts for the increased performance seen in Table 2. Another consequence of the addition of a hardware floating point unit, is that there is a change on the performance bound for the FFT. The application executing on all systems shown in Table 1 was computationally bound as a result of floating point emulation and communication time was negligible on the multiprocessor systems. In contrast, the multiprocessor systems in Table 2 have become communication bound.

The performance numbers of Table 2 show improvements in execution time when the application is moved from the uniprocessor to the multiprocessor environment. The improvement factor is less than 1.5x as a result of the impact of communication cost on the overall application. For this case, the performance of the parallelized solution is dependent on the underlying communication links. The speedup factor for the 128-point computation from a uniprocessor to a 4 processor system is approximately 1.5. This indicates that the processors are mostly idle and waiting on data from their neighbors.

Table 2. EMPD Star Interconnect, Floating Point Support.

Computational Processors	Data Points (Execution Time in ms)			
	16	32	64	128
1	0.62	0.85	1.18	1.71
2	0.43	0.80	1.01	1.50
4	0.59	0.60	0.81	1.21

Experimental results of the other two interconnects showed a +/-10% difference with respect to the star. The mesh showed the worst performance as a result of the data pass through functions required of all worker nodes. The fully connected topology consistently provided better performance. In the case without floating point support, the fully connected has a 10% speedup over the star. This number increases to 20% once floating point support is added. The drawback of the fully connected topology is its high resource consumption.

6. Conclusion

This work presented the EMPD, which is a platform for the development and prototyping of embedded multiprocessor systems. EMPD was verified for functional correctness through the use of an FFT benchmark application. The results from this test application show that EMPD is a viable platform for multiprocessing embedded system research on an FPGA. Also during the development of this platform, a custom operating system called LightOS was created for interface and control of the computation cluster.

7. References

[1] Grama, A.; Gupta, A.; Karypis, G.; Kumar, V., *Introduction to Parallel Computing, 2nd ed.*, Pearson, Essex, England, 2003.
[2] Olikutun,K.; Nayfeh,B.A.; Hammond, L.; Wilson, K.; "The Case for a Single-Chip Multiprocessor, " *The 8th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IIV)*, pp.2-11, 1996.
[3] Xilinx Corporation, *Virtex-IIPro User Guide*, 2005.
[4] Xilinx Corporation, *Microblaze Processor Reference Guide*, 2005.
[5] Ludwick, S.; "Linux for Data Acquisistion," *Evaluation Engineering*, 2002.
[6] Williams, J.A.; :MicroBlaze uCLinux Project," University of Queensland, 2004.

[7] Williams, J.A.; Bergmann, N.W.; "Embedded Linux as a Platform for Dynamically Self-Reconfiguring Systems-on-Chip," *Engineering of Reconfigurable Systems and Algorithms (ERSA 2004)*, Las Vegas, Nevada, USA, June 2004.

[8] Probert, D.; Bruno, J.; Karzaorman, M.; "Space: A New Approach to Operating System Abstraction," *Proceedings of the International Workshop on Object Orientation in Operating Systems*, 1191. pp. 133-137.

[9] Morie, M.; et.al; "Using Meta-Interfaces to Support Secure Dynamic System Reconfiguration," *Proceedings of the 4th International Conference on Configurable Distributed Systems*, 1998.

[10] MPI Forum; "MPI: A Message Passing Interface," *Proceedings of 1993 Supercomputing Conference*, Portland, WA, 1993.

[11] Brigham, E.O.; *The Fast Fourier Transform and Its Applications, 2nd ed.*, Prentice Hall. Englewood Cliffs, N.J. 1988.