LEVERAGING EMBEDDED HETEROGENEOUS PROCESSORS FOR SOFTWARE DEFINED RADIO APPLICATIONS

Almohanad S. Fayez, Qinqin Chen, Jeannette Nounagnon, Charles W. Bostian
Wireless@VT
Bradley Department of Electrical and Computer Engineering
Virginia Tech, Blacksburg, VA 24061
(afayez, chenq, jddjig01, bostian}@vt.edu

ABSTRACT

The paper focuses on utilizing embedded processors with integrated General Purpose Processor (GPP) and Digital Signal Processor (DSP) cores for SDR applications. With such processors, developers can leverage operating systems running on GPP cores and use DSP cores to accelerate signal processing tasks in building small form factor SDR solutions. The SDR community realizes the importance of heterogeneous computing; however, common software radio development tools like GNU Radio and OSSIE do not provide wide enough support for computing devices other than GPPs.

This paper discusses the toolset support for GPP/DSP based processors, and explores how the TI OMAP3530 processor can be used to add DSP support for SDR developments tools, specifically GNU Radio. It presents an example where the OMAP DSP core is used for filtering in a GNU Radio flow graph; the paper also discusses some integration aspects of which application developers should be aware.

1. INTRODUCTION

Looking at the diverse applications for SDR, ranging from mobile handheld to base-station level radios, it is evident that various classes of power utilization and computational density are needed to cover the wide spectrum of SDR needs. For such applications, three main computing devices are available: General-Purpose Processors (GPPs), Digital Signal Processors (DSPs), and Field-Programmable Gate Arrays (FPGAs). Many existing SDR platforms accommodate different combinations of computing devices and allow design partitioning amongst them[1], [2], [3], and [4]. Dealing with such computing heterogeneity is an important aspect of system design for application developers. It becomes important to abstract such computing heterogeneity in SDR platforms to focus on the overall application design without being encumbered by low-level system design issues.

In abstracting computing heterogeneity, the Software Communications Architecture (SCA) has introduced the use of CORBA as a middleware [5] between different computing devices and applications. While it is possible to abstract the underlying computing devices using such techniques, middleware interfaces can incur significant latency causing performance degradation in the overall system [6]. Some vendors provide proprietary FPGA/DSP interfaces and libraries for device communication such as the Lyrtech Lyrio interface [2]. However, such proprietary interfaces and libraries can be a hurdle for porting and developing applications outside the realm of the vendor's platforms and development environment.

This paper will discuss work done on the Texas Instruments (TI) OMAP3530 processor [7], an integrated GPP/DSP chip targeted for embedded DSP applications, installed on the Beagleboard [8]. The scope of the work includes writing a library to perform the inter-process communication between the GPP and DSP, integrating the DSP as a coprocessor for GNU Radio, and performance results for a DSP accelerated GNU Radio flow graph.

The paper is organized as follows: a summary of the hardware platform used, a description of the software environment used, current functionality available in the library, integration of the DSP in GNU Radio, and an example of using the library within GNU Radio.

2. HARDWARE PLATFORM

The Beagleboard is intended as a low cost (<$150) platform that makes the TI OMAP3530 processor accessible to the open source community [9]. The TI OMAP3530 is intended for mobile graphic applications [10]; it is an integrated GPP/DSP processor combining an Arm Cortex-A8 GPP and a C64x+ DSP core. The Cortex-A8 is composed of an Armv7 GPP with a NEON core and a multimedia acceleration SIMD coprocessor [11]. The C64x+ DSP is a 16-bit fixed-point VLIW DSP. The DSP contains eight independent functional units: six ALUs and two multipliers that support either four 16 x 16 bit multiplies or eight 8 x 8

bit multiplies [10]. The Beagleboard revision used in this work supports a host-side EHCI USB driver which is needed to interface with first generation USRPs [12].

3. SOFTWARE TOOLS

In targeting the GPP and DSP on the Beagleboard, a number of different software tools are used.

a- OpenEmbedded (OE) [13]: The general framework used to target the Beagleboard. It is used to compile the Linux kernel, *Angstrom* distribution, and to generate a file system for the board. OE is also used to compile the necessary drivers for the board and compile the various packages needed.

b- TI DSP/BIOS Link (DSPLink) [14]: This library is used for inter-processor communication between the GPP and DSP on the OMAP processor. The DSPLink library provides mechanisms for basic processor control, data transfer, memory sharing/synchronization, and messaging, among other inter-process functionalities.

c- C6x Compiler [15]: The TI DSP compiler used on a Linux machine. The TI integrated development environment, Code Composer Studio (CCS) v3.3, is also used to debug DSP code in combination with a JTAG. CCS is not necessary for compiling C64x+ DSP code since TI provides its compiler through their website at no cost. However, CCS is necessary to use a JTAG.

d- DSP BIOS [16]: DSP/BIOS is a lightweight real-time library which provides basic run-time functionalities on the DSP. Examples of such functionalities include scheduling threads, handling I/O, and capturing information in real-time.

e- GNU Radio [17]: GNU Radio is installed on the Beagleboard via the OE environment. The DSP is integrated as a functional block on a native GNU Radio installation. Using all of the previous tools, it is possible to target both the Arm GPP and the C64x+ DSP on the Beagleboard through GNU Radio.

To install the TI toolsets and build the necessary DSPLink drivers, we followed instruction available on [18].

4. GNU RADIO DSP EXTENSION

Our goal is to extend GNU Radio to provide support for DSP processors. While GNU Radio applications are written in Python, the actual signal processing is implemented in C++ and the Python/C++ interface is performed by the software toolset SWIG (Simplified Wrapper and Interface Generator). Using Python to construct flow graphs can be convenient since it's an interpreted language meaning it does not require source code compilation such as C++. However, compiled languages such as C++ are not encumbered by the overhead of runtime compilation associated with interpreted languages.

From a high abstraction perspective, Figure 1 demonstrates how the DSP looks to an application developer. The figure shows how the DSP can be instantiated to perform functions within GNU Radio flow graphs. In the figure, *GNU Radio block* is a generic term referring to existing GNU Radio functions. The code necessary to perform the inter-processor link using DSPLink and loading the necessary DSP program is encapsulated into a custom GNU Radio block so the underlying support code is transparent to users. The next sections in the paper will discuss the details of the work done to provide such abstraction for the DSP and DSPLink interface.

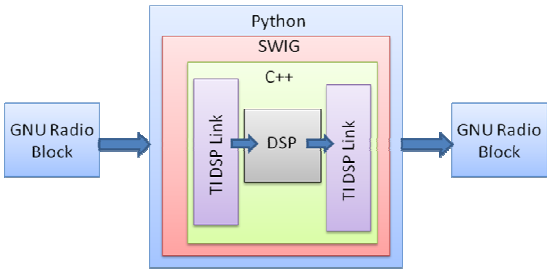

Figure 1. Application level abstraction of the DSP in the OMAP processor.

5. GPP AND DSP CODE DEVELOPMENT

GPP and DSP applications are written and compiled separately with the DSPLink library providing the necessary constructs for inter-processor communication. To better understand the program flow between GNU Radio and the DSP, Figure 2 shows the interaction between the different programming domains, with execution starting in the *GNU Radio Python* block which ultimately utilizes the *DSP Program block* to perform filtering. Looking at the figure, the DSP is invoked from a python file and through SWIG, GNU Radio invokes the necessary C++ calls; the latter Python and C++ code enables GNU Radio to invoke the DSP. The GPP library includes a set of functions necessary to communicate with the DSP over DSPLink.

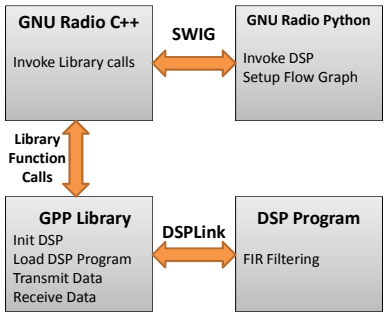

Figure 2. Interaction between the different programming domains.

The GPP library currently uses three components from DSPLink:

a- PROC: the processor component.
b- POOL: the shared memory configuration and management component.
c- CHNL: the channel component.

The PROC component is used to initialize the DSP processor, load DSP executables, and stop DSP execution; basically it is used to setup and load the DSP. The POOL component is used to configure and synchronize the shared memory regions between the GPP and DSP necessary for inner-processor data transfers. The CHNL component instantiates a logical data transfer link, or channel, between the GPP and DSP. While the CHNL component manages the logical data link between the DSP and GPP, the POOL component manages the physical memory used for the CHNL component including all the necessary memory allocation, synchronization, and address translation between the GPP and DSP. The GPP library essentially abstracts the DSPLink function calls necessary to initialize the DSP, instantiate the shared memory region, create the channel constructs, and the physical exchange of data back and forth with the DSP.

Data transfers types can either be real or complex (IQ) floating point data with all the necessary data type conversions, floating-point to fixed-point and vice-versa, taken care of by the application. The user needs only to supply a scaling value that is necessary for the data type conversion. Figure 3 shows an example of how the GPP library can perform the data type conversion between floating-point format on the GPP and fixed-point format on the DSP. The figure shows the GPP sending an array of three values to the DSP. The input data from the GPP are normalized floating-point values; the GPP scales the data by a factor of 2^14 and truncates any value after the decimal point to generate the fixed-point values for the DSP. For the purposes of this example, after receiving the GPP data the DSP sends the same received input back and the GPP scales the fixed-point values by a factor of 1/2^14 and generates floating-point value from the fixed point values. As we can see, the GPP does not receive the original values it sent to the DSP because of the quantization error in the floating/fixed point conversion.

The DSP program connects to the GPP data channels through DSP BIOS specific library calls. In this application, the stream I/O manager (SIO) is used to create and manage data streams needed to communicate with the data channels running on the GPP side. The DSP receives data streams from the GPP which includes the length of the data streams being passed to the DSP. The SIO is able to allocate data streams of various sizes dynamically; however, in the work presented, the DSP and GPP allocate streams/channels of a

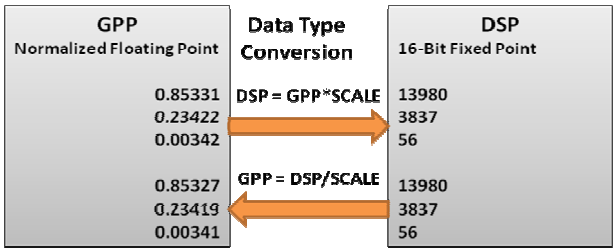

Figure 3. An example of floating/fixed point conversion where SCALE = 2^14.

static size, but they then fill the inter-processor buffers with as many data elements as needed by the application. The GPP passes the DSP information about the number of elements written in the buffer so the DSP does not have to read more elements from the buffer than necessary. By passing the buffer size with every transfer, the DSP reads only valid buffer elements and synchronizes the buffer without having the GPP clear the buffer before each transfer. To better understand the data buffer between the GPP and DSP, Figure 4 shows an example of two consecutive buffers being passed from the GPP to the DSP; the first buffer is of size 3 and the second is of size 1 while the size of the buffer = MAX is some larger number representing the maximum size, in this case 512, which is also the index of the last element in the buffer. The first buffer has only three data points; elements 4 through 512 (MAX) are empty while the second buffer has only one valid data point so elements 2 and 3 contain stale data while the rest of the buffer elements are empty.

To abstract the above discussed function calls, a library is compiled with an associated header file to be used by application developers to access the DSP. The primary library function calls include:

a- DSP and inter-processor buffer initialization.
b- GPP transmitting real or complex (IQ) data to the DSP.
c- GPP receiving real or complex (IQ) data from the DSP.

Please note that the DSP program is a separate executable which is loaded as part of the DSP initialization routine from the GPP side of the application.

0	1	2	3	4	…	MAX = 512
len= 3	data[0]	data[1]	data[2]	0	0	0

0	1	2	3	4	…	MAX = 512
len = 1	data[0]	OLD	OLD	0	0	0

Figure 4. Two example buffers to be transferred by the GPP to the DSP. The top buffer is of length = 3; the bottom is of length = 1, and the buffer size is a larger number = MAX in this case 512.

6. GNU RADIO DSP INTEGRATION

For the work presented, the base DSP and GPP code used written in C++ while hooks are also supported to enable DSP configuration and integration in Python. In integrating the DSP as a GNU Radio block, we used the GNU Radio template for writing a custom block "gr-howto-write-a-block-3.2.1" from the GNU Radio repository [17] and installed GNU Radio 3.2.1 on the Beagleboard using OE. The base template was used to create a set of *dsp* class functions. As a test of our framework, we use the TI DSP library "C64x+ DSP Library". Initially, we concentrate on providing filtering functionalities on the DSP; in this paper we characterize the performance of a complex FIR filter. The DSP FIR implementation is intended to replace its GNU Radio GPP-based counterpart (fir_filter_ccf), a FIR filter with complex input/output, real coefficients, and optional interpolation.

The DSP filter implementations are instantiated in the same way that GPP filters are instantiated in GNU Radio. The following is a sample instantiation of the gr_fir_filter_ccf on the DSP from a native GPP python file:

fir_ccf_dsp = dsp.fir_ccf_dsp(coeff, sf, interp)

Where *fir_ccf_dsp* is a handler to the DSP based filter, *coeff* is the set of filter coefficients, *sf* is the scaling factor and *interp* is the filter interpolation factor.

When the filters are instantiated during a GNU Radio flow graph, they essentially instantiate the associated DSP transmit/receive function calls for real or complex data. In compiling the DSP class functions for the Beagleboard, we use a modified OE *recipe* (compile instructions) provided by the authors of [12].

7. GNU RADIO TESTING

To compare the performance of a DSP based filter with its GPP based counterpart, we set up a GNU Radio flow graph comprised of a 300 tap complex filter with input data being read from an input file, fed to the complex filter, and the results saved to another file. Figure 5 displays the base flow graph used for both the GPP and DSP based filters. However, when the DSP filter is used, input data needs to be scaled, if the input data is in floating-point versus fixed-point format , and the data also needs to be physically transferred from the associated buffers on the GPP to the DSP and back to the GPP. Figure 6 demonstrates the DSP flow graph with the necessary extra processing overhead.

The taps and input data are normalized fixed point random data which are generated in Matlab with the input data sets ranging from 1600 IQ samples to 16*10^6 IQ samples. Flow graph execution time is measured for five

Figure 5. Base GNU Radio Flowgraph

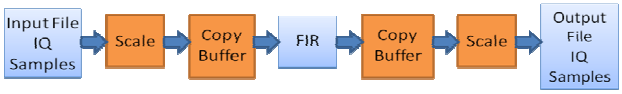

Figure 6. DSP FIR Flowgraph

different scenarios:

1- *GPP based FIR implementation:*
Measures execution time for GPP based FIR filter.
2- *DSP based FIR implementation:*
Measures execution time for DSP based FIR filter.
3- *Reading IQ input from file and IQ output sinking:*
Measures time needed to read input data from file and sinking, saving, it to file. This represents the setup time for both FIR implementations, flow graph setup, and data input/output.
4- *DSP loopback **without** data format conversion:*
Measures the time needed to copy fixed-point data from the GPP to the DSP.
5- *DSP loopback **with** data format conversion:*
Measures the time needed to copy floating-point data from the GPP to the DSP.

The GPP core clock frequency is 500 MHz and the DSP clock frequency is 360 MHz. The Linux *time* command is used in measuring the performance of each flow graph. We use the *time* utility output of total execution time instead of CPU execution time since it accounts for memory transfer time and DSP execution time versus CPU execution time which only measures execution while the GPP is utilized. We collect ten execution time measurements for the above scenarios and we use their average as the runtime for a specific input size. Table 1 summarizes the time measurements for the GPP/DSP based FIR filters and the DSP speedup factors and Table 2 summarizes the following:

- Time measurements for IQ data input/output, which is an overhead shared by both the GPP and DSP based flow graphs.
- DSP loopback **without** floating-point/fixed-point conversion.
- DSP loopback **with** floating-point/fixed-point conversion.

Looking at Table 1, we can see that the speed up factor increases as the input data size increases, basically the DSP

Input (IQ pairs)	GPP Execution (seconds)	DSP Execution (seconds)	Speedup factor
1.6*10^3	1.916	2.109	0.91
16*10^3	3.63	2.19	1.66
160*10^3	20.519	2.494	8.23
1.6*10^6	189.308	6.237	30.35
16*10^6	1876.937	46.837	40.07

Table 1. Summary of GPP/DSP based FIR filter execution times and DSP implementation speedup factor.

Input (IQ pairs)	Flowgraph setup IQ data I/O (seconds)	Loopback without conversion (seconds)	Loopback with Conversion (seconds)
1.6*10^3	1.718	2.11	2.128
16*10^3	1.713	2.153	2.151
160*10^3	1.782	2.361	2.487
1.6*10^6	3.159	5.085	5.735
16*10^6	22.987	37.919	42.11

Table 2. Summary of IQ data overhead in the flow graphs and DSP overhead for buffer copies and data format conversion.

implementation overhead consumes less time in comparison to the computation time necessary for the GPP implementation. When we observe the results in Table 2, we see that in the DSP-based filter flow graph, reading and saving IQ data into files consumes the majority of execution time, while for the GPP based solution the filter execution time consumes the majority of the execution time. Looking at Table 2 we can also note that the transfer between the GPP/DSP and data format conversion consumes a considerable amount of time.

We can see from the results, that DSP processors can accelerate the performance of SDR implementation in embedded platforms. In terms of overhead, buffer transfer between processors is an expensive operation execution-wise; therefore SDR implementations with multiple GPP/DSP transfers can offset the performance gain of using a DSP by long delays caused by memory transfers.

8. FUTURE WORK

We plan on supporting GPP/DSP buffer allocation so buffer pointers can be shared between GNU Radio and the DSP, essentially to pass pointers between them instead of direct memory copies. We also plan to perform over-the-air testing for transmitters and receivers utilizing DSP based filters and we will supplement the functionalities made available by the DSP to GNU Radio.

9. CONCLUSION

This paper presents work done to access and abstract the DSP processor on the Beagleboard. The paper describes the low level work done using DSPLink to support inter-processor communication between the GPP and DSP for the TI OMAP3530 processor and it also describes work done to support DSP side functionality using the TI DSP/BIOS real-time library. Testing for a mixed GPP/DSP computing scenario was done by linking the library to a GNU Radio install on a Beagleboard. The use of the DSP was demonstrated by allowing an existing GNU Radio flow graph to instantiate a DSP based filter versus the existing GPP based implementation and data was runtime measurements were collected to compare between the GPP and DSP based filter implementations.

10. ACKNOWLEDGMENT

This project is supported by Award 2009-SQ-B9-K011 awarded by the National Institute of Justice, Office of Justice Programs, U.S. Department of Justice. The opinions, findings, and conclusions or recommendations expressed are those of the authors and do not necessarily reflect the views of the Department of Justice.

10. REFERENCES

[1] [Online]: http://warp.rice.edu/trac/.
[2] [Online]: http://www.lyrtech.com/.
[3] [Online]: www.sundance.com.
[4] [Online]: http://www.ettus.com/.
[5] JTRS,"Specialized Hardware Supplement to the Software Communication Architecture (SCA) Specification".
[6] M. Carrick, S. Sayed, C. Dietrick, and J. Reed, "Integration of FPGAs into SDR Via Memory-Mapped I/O," in *SDR Technical Conference and Product Exposition* Washington, D.C., 2009.
[7] [Online]: http://www.ti.com.
[8] [Online]: http://beagleboard.org/.
[9] "Beagleboard System Reference Manual Rev C4," 2009.
[10] Texas Instruments,"OMAP3530/25 Applications Processors".
[11] [Online]: http://www.arm.com/products/processors/cortex-a/cortex-a8.php.
[12] C. R. Anderson, G. Schaertl, and P. Balister, "A Low-Cost Embedded SDR Solution for Prototyping and

Experimentation," in *SDR Technical Conference and Product Exposition* Washington, D.C., 2009.
[13] [Online]: http://wiki.openembedded.net/index.php/Main_Page.
[14] Texas Instruments,"DSP/BIOS Link User Guide".
[15] Texas Instruments,"C6000 Code Generation Tools".
[16] Texas Instruments,"TMS320C6000 DSP/BIOS User's Guide".
[17] [Online]: http://gnuradio.org/redmine/wiki/gnuradio.
[18] [Online]: http://ossie.wireless.vt.edu/trac/wiki/BeagleBoard.