DESIGN AND IMPLEMENTATION OF AN SCA CORE FRAMEWORK FOR A DSP PLATFORM

Carlos R. Aguayo Gonzalez (MPRG, Wireless@Virginia Tech, Blacksburg, VA, USA; caguayog@vt.edu);
Francisco Portelinha (Universidade Federal de Itajuba, Itajuba, MG, Brazil, portelinha@uti.psi.br); and
Jeff Reed (MPRG, Wireless@Virginia Tech, Blacksburg, VA, USA; reedjh@vt.edu).

ABSTRACT

The Software Commuications Architecture (SCA) was developed to improve software reuse and interoperability in Software Defined Radios (SDR). However, there have been performance concerns since its conception. Arguably, the majority of the problems and inneficiencies associated with the SCA can be attributed to the assumption of modular distributed platforms relying on General Purpose Processors (GPPs) to perform all signal processing. Significant improvements in cost and power consumption can be obtained by utilizing specialized, more efficient platforms. Digital Signal Processors (DSPs) present such a platform and have been widely used in the communications industry. Improvements in development tools and middleware technology opened the possibility of fully integrating DSPs into the SCA. This approach takes advantage of the exceptional power, cost, and performance characteristics of DSPs, while still enjoying the flexibility and portability of the SCA.

This paper presents the design and implementation of an SCA Core Framework (CF) for a TI TMS320C6416 DSP. The framework is deployed on a Lyrtech Quad-SignalMaster C6416 board. The SCA CF is implemented by leveraging OSSIE, an open-source implementation of the SCA, to support the DSP platform. Prismtech's e*ORB and DSP/BIOS are used as the middleware and operating system, respectively. A sample waveform was developed to demonstrate the framework's functionality. Benchmark results for the framework and sample applications are provided.

1. INTRODUCTION

The Software Communications Architecture (SCA) was developed by the Joint Tactical Radio System (JTRS) program of the US Department of Defense to standardize the development of Software Defined Radio (SDR) technology. The SCA was developed to enhance system flexibility and interoperability, while reducing development and deployment costs. Early implementations of SCA SDRs have struggled to meet performance, cost, size, and power requirements. Arguably, many of the these problems have their origin in the assumption of a modular, distributed platform based on General Purpose Processor (GPP) to perform all signal processing.

In order to overcome these problems, it is necessary to make better use of specialized hardware optimized for signal processing. Digital Signal Processors (DSP) are specialized microprocessors designed specifically for real-time digital signal processing. However, DSPs have been relegated as secondary elements in the SCA, requiring a Hardware Abstraction Layer (HAL) for connectivity. Ongoing improvements in development tools and middleware technology allow the implementation of SCA systems using only DSPs. By following this approach the flexibility and reusability brought by the SCA are complimented by the cost and power efficiency of DSPs. If taken to a logical extent, this approach could eliminate the need for a GPP on certain SDR implementations. In this paper we present the design and development of an SCA implementation for a homogeneous TI DSP platform.

2. SYSTEM ARCHITECTURE

The goal of this project is to study the repercussions of implementing the SCA in an optimized DSP platform. Therefore, we aim to minimize, or eliminate, the use of GPPs for this implementation. We leveraged the existing implementation of MPRG's Open Source SCA Implementation::Embedded (OSSIE) [2], by porting it to the C64 platform. The system implements the SCA version 2.2 in C++. Our development environment is TI Code Composer Studio running on a Windows PC. Most of the development is done using the Device Accurate simulator of the C6000. The final target platform is a Signal Master Quad from Lyrtech.

2.1. Software Architecture Elements

The general software structure can be seen in Fig. 1, showing the three different components of the SCA Operational Environment (OE): the Core Framework (CF), ORB, and operating system. In this project we used OSSIE as the CF, e*ORB from PrismTech as middleware, and DSP/BIOS as Real-Time OS. All of them are available commercially or as open source. Services (e.g. Log, Event, and Naming Services) are not considered in the initial implementation.

DSP/BIOS is a scalable real-time multitasking operating system designed specifically for the TMS320 family of DSPs [4]. It is developed and maintained by Texas Instruments. DSP/BIOS is built in modules which allows developers to reduce the footprint to a minimum by only integrating the features that are strictly necessary for operation. It supports preemptive multithreaded operations thanks to a real time scheduler and provides memory management modules for low overhead dynamic memory allocation. DSP/BIOS is not POSIX compliant, as required by the SCA, forcing a slight deviation from the specifications. The C6000 family of processors does not include a memory management unit.

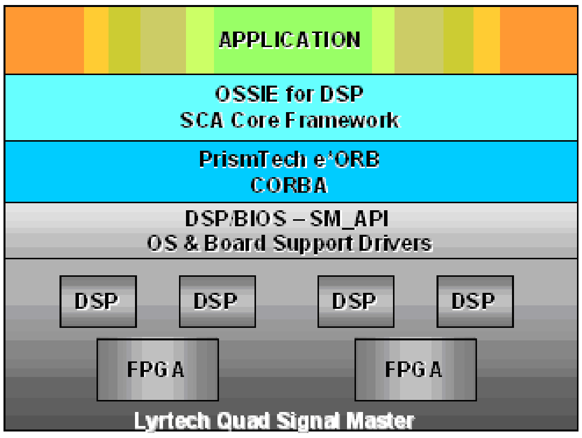

Fig. 1. Software Structure

The ORB used in this project is PrismTech's e*ORB SDR C++ version for DSP. This is a custom port of their commercially available C version, a very optimized, modular implementation of minimum-CORBA as standardized by the Object Management Group (OMG). However, Prismtech has released this version to the general public [3]. e*ORB supports the Extensible Transport Layer (ETF) which allows custom transport plug-ins.

2.2. Platform

The target platform for this project is the SignalMaster Quad C6416 from Lyrtech [1]. This high performance board contains four TI TMS320C6416T DSPs and two Xilinx Virtex II FPGAs divided into two clusters (one FPGA and two DSPs each). The system runs at 720 MHz and has 128Mbytes of SDRAM memory per DSP. There is a high speed bus between both FPGAs implemented using LVDS. The communication between two DSPs within the same cluster can be implemented using shared memory or FastBus, a proprietary protocol developed by Lyrtech. In this project, only DSPs are used for signal processing and framework functionality. FPGA operation is limited to inter-DSP communications.

3. REAL-TIME IMPLEMENTATION

The bulk of this project consists of porting the existing version of OSSIE to the C64 platform. The original OSSIE runs on an x86 platform running Linux with omniORB as middleware.

As with any other software project, development tools play a very important role. We use Code Composer Studio (CCS), an integrated development environment for TI DSPs, with version 5.1.0 of its Code Generation Tools. This particular version lacks the Standard Template Library (STL) and has limited support for C++ exceptions. The STL provides template classes such as Vector, widely used in the original OSSIE. In the absence of exception support, we use CORBA Environment variables coupled with a set of macros, distributed as part of e*ORB, for error handling. These characteristics forced significant changes in the original OSSIE source code.

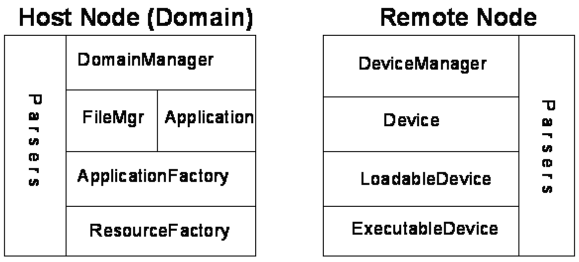

Fig. 2. Processing Node Deployment Scheme

An important aspect in the development of this project is the lack of a Memory Management Unit (MMU) in the C64. The MMU is responsible for handling memory access requests. It takes care of virtual memory management, paging, memory protection, and bus arbitration. Its job is to take pieces of dispersed physical memory and present them to the requesting process as a contiguous block. In porting OSSIE to the MMUless C64 platform all memory management is the responsibility of the developer and certain OS calls, such as fork(), are not supported.

Another important area in the development is the porting of all scheduling calls to the preemptive, multithreaded DSP/BIOS. The main difference from a traditional fair-share OS is that the active task with the highest priority will be scheduled for execution; no matter how many other tasks are waiting, or for how long. This characteristic allows deterministic execution, crucial in real-time systems, but makes the developer completely responsible for task scheduling and priority assingment.

The functionality of the Core Framework is split between Host and Remote nodes. The Host node includes an instance of DomainManager, while a remote node includes an instance of DeviceManager and other Devices. Fig. 2 shows the CF interfaces allocated to each node. There are other possible strategies, for example having a node host both DomainManager and DeviceManager, while the rest of the nodes in the platforms only host Devices. We decided on this approach to stress our implementation and evaluate the degree of flexibility delivered by it.

3.1. XML Parsing Strategy

The SCA specification requires the reading of the XML Domain Profile at runtime to obtain deployment and configuration information. For example, the ApplicationFactory interface must read a SAD file in order to know what components are included in a given waveform and their connections. Parsing an XML file is a complicated task for a DSP and there are not many tools available to perform this. In order to facilitate development, reduce memory requirements, and speed execution, we developed a two-step parsing scheme designed to facilitate Domain Profile parsing by the DSP. In this scheme, an offline translation of the XML files into a simplified format is performed. The simplified format only keeps the most important information from the profile files and stores it in a simple text file. The

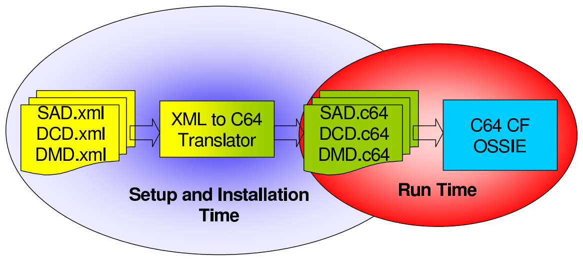

Fig. 3. Simplified Parsing Strategy

information kept includes all the data required for successful deployment and configuration of waveforms and components: UUIDs, descriptors locations, connections, etc. The information discarded represents information not indispensable for waveform deployment and operation: descriptions, headers, authors, etc. Even though the discarded information is important, and therefore must be provided when developing an SCA component, the main framework functionality does not require it for proper operation. A graphical representation of this approach is shown in Fig. 3. It can be argued that this approach is not SCA compliant. However, having this two-step parsing does not affect the design cycle of traditional SCA waveforms and only adds one extra step at installation time. The savings in time and complexity, along with the uncompromised portability of the resulting waveforms justify this decision.

We implemented the XML translator in C++ as a stand-alone application. It uses the Xerces parser and the parser library from the original OSSIE project. The translator parses an SCA compliant XML file, gathers the required information, and writes the translated file with a .c64 extension, preserving file names and directory structure. These simplified .c64 files are then parsed at real time by the framework running on the C64.

3.2. File System

Our hardware platform does not have long-term storage capability. Therefore, only a partial file system is implemented in this project. The host computer's hard drive and file system are used by the framework. This is accomplished by CCS I/O utilities and the JTAG interface.

To implement the file system interfaces we relied on IO functions from the TI run-time support library. However, the access allowed by this library is limited primarily in terms of directory manipulation. Therefore, functionality such as mkdir, rmdir, mount, and unmount is not implemented.

3.3. Software Component Deployment

The SCA specifies two equivalent mechanisms to launch software components. One is using ResourceFactory and the other using ExecutableDevice. The ExecutableDevice interface typically represents processors with a multithreaded operating system capable of launching software components. ExecutableDevice has access to the OS directives to schedule the component. ResourceFactory performs the exact same functionality and is used as a local tool to deploy components without a DeviceManager. In this project we use the ResourceFactory interface to deploy components in the host node and an ExecutableDevice for remote nodes.

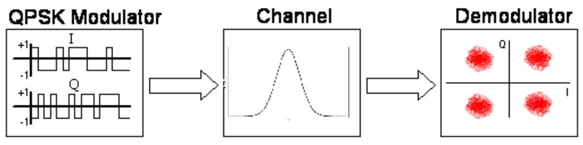

Fig. 4. Sample QPSK Application Waveform

The implementation of these interfaces uses DSP/BIOS task scheduler. Every time a new component instance is required, a new task is created and scheduled. The ResourceFactory and ExecutableDevice implementations are in charge of managing the new task's priority. Because of the lack of an MMU and long-term storage capability, it is necessary to have all the tasks loaded in program memory before they can be scheduled.

4. SAMPLE APPLICATION

In order to demonstrate the framework functionality, two sample applications are developed. These applications are intended for demonstration purposes and nothing else. No extensive signal processing is performed. The main goal for these applications is to verify the operation of the framework and to corroborate the feasibility of deploying SCA compliant waveforms onto the C64 platform.

The first application includes three simple components: BPSK Modulator, Channel, and Demodulator. The BPSK modulator generates a random stream of 1's and -1's. The stream is passed to the Channel component which adds Gaussian noise to the In-Phase and Quadrature components of the stream. The Demodulator only displays the constellation diagram of the signal. The second waveform includes a QPSK demodulator instead of BPSK. Fig. 4 shows a graphical representation of the second waveform. Both waveforms were successfully deployed on a single chip configuration using the ResourceFactory interface to launch the components.

5. RESULTS

In this section we present general profiling results for the implementation. The framework capabilities are demonstrated by switching back and forth between two waveforms. Code Composer Studio (CCS) is used to control the execution, display information and error messages, and enter selection values. Keep in mind that from the framework perspective there is no difference between deploying these simple waveforms and deploying more sophisticated ones. The source code for the framework, XML translator, and sample application is openly available and can be downloaded from [2].

5.1. Profiling

Profiling was performed on the framework and application using two different metrics: memory footprint and cycle count. The former represents the extra memory space necessary to suport the SCA framework. The latter represents the amount of overhead imposed by the framework in terms of processing power.

All results were obtained from a single-chip configuration. That is, all framework and waveform components were collocated within the same processor; they do not include a transport layer. **No optimizations were performed** in either the framework or the waveform components. All performance tests were carried out using the C6416 Device Cycle Accurate Simulator and the Code Composer Studio profiler. It is very important to emphasize that these results represent initial measurements and are subject to further investigation, validation, and optimization.

5.2. Memory Footprint

Memory allocation results are obtained from the .MAP file generated by CCS Code Generation Tools. This file contains a maping of all sections allocated in memory. It includes program memory and data memory. All dynamic memory allocation requests are served from a memory pool or heap, which is also included in the .MAP file. All profiling results are presented in 8-bit bytes. Note that each DSP on the SignalMaster Quad board has 128M of external memory (ERAM) besides the 2K Bytes of on-chip memory (ISRAM).

The total memory used by the system is shown in Table I. It represents a little more than 1% of the available memory per DSP in the platform. These results correspond to the single-chip implementation of both, BPSK and QPSK, sample waveforms. The footprint is directly related to the application's functionality and the number of components. Table II shows the memory breakdown by major components. The .ERAM$heap field represents the total heap available to serve dynamic memory allocation requests from the application. The footprint contribution from support libraries (e.g. Generic Runtime Library, Math Library, etc) is considered under the "Other" category. Fig. 5 shows a graphical representation of the main components' contribution to the total memory allocation.

TABLE I
TOTAL MEMORY ALLOCATION

Memory Type	Bytes
ISRAM	78096
ERAM	1383145
TOTAL Memory Usage	1461241
TOTAL Memory Available	120000000

Due to space limitations, we do not break down the memory footprint for each major component. Instead, we comment on some important aspects and state some qualifiers for these results.

In the break down of the memory requirements for the Core Framework (CF) we find that almost 70% of the total memory allocated for the CF comes from the C++ mapping of the SCA CF IDL interfaces. It is important to note that

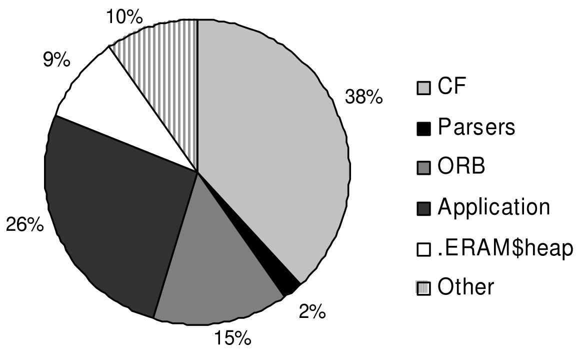

Fig. 5. Memory Footprint Summary

the CF IDL descriptions, cf.idl file, contain all the interfaces defined in the SCA CF, including some that are not used in single-processor operation (e.g. Device, DeviceManager). It is possible to optimize the C++ bindings of IDL interfaces by adding more control to the IDL compiler, enabling more selective code generation (e.g. for specific interfaces generate stub only, or skeletons only, or nothing). This approach opens the door for potentially large improvements depending on how much of the IDL interfaces are being used. This is a well understood approach, although it is not implemented in this project. Another important qualifier for these results is the absence of Device-related interfaces. No DeviceManager or Device interfaces were implemented. The methods in DomainManager relative to Device and service registration and unregistration are not implemented in this version as well.

The memory requirement results for the application include both BPSK and QPSK components, along with Channel, Demodulator, Resource Factory, Assembly Controller, and the user interface. The main waveform components have a very similar footprint as expected. However, the functionality of these components is extremely simple. More complex waveforms will require more memory.

The results correspondent to the ORB are from a preliminary release. Further development and optimization has been performed on e*ORB and these results may not accurately represent the memory footprint of the latest version.

TABLE II
MAJOR COMPONENTS CONTRIBUTION

Memory Type	Bytes
CF	556555
Parsers	31511
ORB	212412
Application	385624
Sub-Total	**1186102**
.ERAM$heap	131072
Other	144067
TOTAL Memory	**1461241**

5.3. Performance Profile

CPU Cycle requirements are collected for the most significant sections of the implementation. The sections profiled were domain initialization and waveform creation. The results are shown in Table III. Domain initialization is not application dependant and includes the instantiation of Domain Manager, ApplicationFactory, and ResourceFactory. Waveform creation represents the execution of ApplicationFactory's create(). It includes descriptor parsing, task scheduling and initialization, and component connection. Keep in mind that waveform creation is waveform specific and these results only apply to our test waveforms.

TABLE III
CF TASKS PERFORMANCE PROFILE

Task	Cycles	Time(sec)
Domain Initialization	2365664	3.286E-03
Create Application	10997946	1.527E-02

5.3.1. ORB Profiling: ORB performance has a great impact on the system because all inter-component communications are established using CORBA messages. Two specific scenarios are used for profiling:

- **Invocation:** roundtrip cycle count for a simple method invocation with no arguments.
- **Marshaling:** roundtrip cycle count for a simple method invocation with basic arguments.

Two different argument types were evaluated:

1) Single Data Type
2) Sequence (1024 elements)

In our version of e*ORB, even for interfaces with no arguments defined in their IDL definitions must receive a CORBA::Environment variable as an argument due to the lack of exception support. In both scenarios, Client and Server where launched as separate DSP/BIOS tasks with priorities 2 and 1, respectively. The e*ORB profiling result for different primitive data types are summarized in Tables IV & V.

An interesting point is that the very first time a client makes a request to a server, the execution takes longer than subsequent requests, as shown in Table IV. This extra delay is due to the binding of new connections, which is a one-off overhead. Subsequent calls are deterministic. There are possible optimizations for this delay and PrismTech is performing further investigation to remove any initial invocation problems if they exist. All results in Table V are from deterministic (after the initial) requests. A graphical representation of the profile is shown in Fig. 6. After observing these results, the need for block processing in an SCA system is evident. It takes 4,208 clock cycles to make a roundtrip marshalling call with a single float, while it takes 6,908 clock cycles to send a sequence with 1,024 floats. Averaging, it only takes 6.74 clock cycles to transfer each element in the sequence.

TABLE IV
E*ORB C++ FOR DSPS INVOCATION PROFILE (CPU CLOCK CYCLES)

Task	Clock Cycles
Initial Invocation	4184
Subsequent Invs.	4081

TABLE V
E*ORB C++ FOR DSPS MARSHALLING PROFILE (CPU CLOCK CYCLES)

	float	char	short	double
Basic Marsh.	4208	4127	4142	4124
Seq. Marsh.	6908	5732	6072	8342

5.4. Impact on Data Rate Performance

The framework overhead incurred during instantiation and waveform deployment can be arranged to happen off-line. The only aspect of the SCA that impacts the system throughput is the dependency on CORBA for intercomponent commnications. The maximum system data rate depends on many factors: algorithm processing delays, framework delays, analog to digital conversion rate, etc. In order to isolate the impact of the framework, we use the results shown in Table V to estimate an upper bound for the system data rate.

Ignoring processing delays, the maximum achievable data rate is given by $1/(T_{fr} + T_m + T_{tr})$ where T_{fr} is the delay due to interface adapters, T_m is the delay due to middleware processing, and T_{tr} is the delay due to transport mechanisms. In our system, we are only considering T_m because no interface adapters are required and no transport mechanisms have been developed at this time.

T_m is given by:

$$T_m = \frac{D_t \cdot S_s}{N_p \cdot n} \tag{1}$$

Where D_t is the measured transfer delay as shown in Table V. S_s is the number of samples per symbol. N_p is the packet size and n is the number of bits per symbol. To estimate the maximum data rate allowed by the framework, we assume $S_s = 8$ and $n = 1$. The clock speed in our system is 720e6. Substituting these values into equation (1) for a single float type transfer, that according to Table V takes 4,142 cycles, the maximum data rate achievable is *21,728* bits per second. However, if we consider sending a sequence of 1,024 floats, the transfer takes 6,072 clock cycles allowing a maximum data rate of *15,177,865* bits per second. This result highlights the need for block processing within the SCA, trading off latency and performance.

6. CONCLUSIONS

One of the main concerns of applying the SCA is the heavy infrastructure required to support it. In order to ease requirements in terms of performance, cost, and power consumption, we propose an implementation of the SCA Core Framework for a TI C64 DSP platform. This approach minimizes the

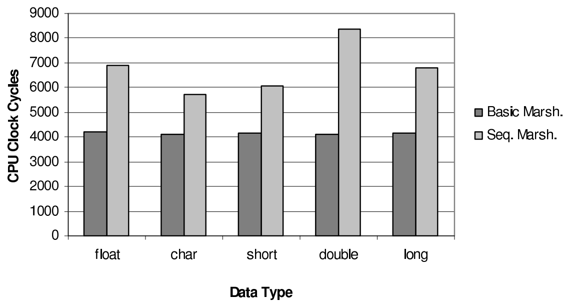

Fig. 6. e*ORB C++ for DSPs Marshalling Profile (CPU Clock Cycles)

use of GPPs and is demonstrated by deploying two sample waveforms. In this implementation, all the framework management tasks are performed by the DSP. Minor deviations from the specifications are required which do not affect the functionality or portability of either the framework or applications.

The total memory footprint of our complete implementation is about 1.5MB, which represents about 1% of the 128MB available per DSP in our platform. Performance benchmarks show that, although CORBA introduces some delays and overhead, the overall effect can be reduced by sending packets of data instead of single elements. The source code for the framework and sample waveforms is available open-source at [2].

ACKNOWLEGMENT

This work was supported by Texas Instrument, Prismtech, Mercury Computer, and Lyrtech.

References

[1] Lyrtech signal processing website. Available at: http://www.lyrtech.com.
[2] Open-source sca implementation::embedded. Available at: *http://ossie.mprg.org*.
[3] Prismtech website. Available at: *http://www.prismtech.com*.
[4] Texas instruments inc. website. Available at: *http://www.ti.com*.