Performance and Overhead Analysis of the ALOE Middleware for SDR

Ismael Gomez, Vuk Marojevic, Jordi Bracke, Antoni Gelonch
Dept. Signal Theory and Communications, Universitat Politècnica de Catalunya
Av. Canal Olímpic s/n, 08860 Castelldefels, Spain
ismael.gomez@tsc.upc.edu, vuk.marojevic@tsc.upc.edu,
jordi.bracke@tsc.upc.edu, antoni.gelonch@tsc.upc.edu

Abstract—**Current Software-Defined Radio applications (waveforms) are tailored to specific hardware. Processor vendors frequently adapt internal OS mechanisms for its specific architecture (e.g. scheduling and synchronization). The Abstraction Layer and Operating Environment (ALOE) is an open source SDR operating environment that isolates platform architecture from the application design. An integrated resource manager is capable of automatically mapping waveform components to a network of heterogeneous processors while meeting the waveform's real-time requirements. This paper analyzes the ALOE performance for x86 and ARM processors. It presents computing histograms of UTRAN transceiver components, the maximum achievable throughput of a simple BPSK modem, interface latencies, and overhead measurements of the ALOE background processes.**

SDR; middleware; resource overhead; software-defined radio performance; abstraction layer and operating environment (ALOE)

I. INTRODUCTION

Software Defined-Radio (SDR) is an emerging technology that is characterized by the implementation of signal processing chains in software rather than in dedicated hardware. As a result, programmable and reconfigurable devices, such as Instruction Set Processors (ISPs) and Field Programmable Gate Arrays (FPGAs), may be combined for building SDR mobile terminals and base stations. The increasing computational complexity with the spectral efficiency of modern Radio Access Technologies (RATs) promotes the utilization of heterogeneous Multi-Processor System-on-chip architectures [1]. Current SDR applications (waveforms) are tailored to a specific MP-SoC architecture. Processor vendors frequently customize a standard operating system (OS), adapting internal mechanisms for their specific MP-SoC architecture (e.g. scheduling and synchronization). Future waveform modifications —to accommodate new services or standard upgrades—are subject to hardware vendor upgrades. This lack of software and hardware reusability increases the development cost.

An operating environment or middleware with abstraction layers isolates the hardware from the software. Component-based waveforms promote the integration of components developed by third parties, moving capital risks to operational costs [2]. Furthermore, base stations infrastructure can be shared between numerous radio operators whereas commercial or tactical terminals can operate with multiple narrowband channels at the same time using the resources of an unused wideband channel [3]. Computing resource management is necessary to enable the shared resource paradigm.

The overhead introduced by SDR operating environments is of great interest. In general, it increases with the number of layers or flexibility introduced to the system. This paper presents a performance and overhead analysis for the Abstraction Layer and Operating Environment (ALOE). First, we briefly introduce the middleware and its measurement tools (section II). Section III describes the execution time profiles or histograms of the 384 kbps UTRAN transmitter and receiver components and the maximum achievable throughput of a BPSK modem. Section IV presents the interface latencies and the middleware's overall resource overhead.

II. ALOE MEASUREMENT MECHANISM

A. A Brief Introduction to ALOE

ALOE is an open-source SDR framework with real-time computing resource management capabilities [4] [5]. It is designed for digital signal processing applications, that is, data flow based processing. The middleware currently supports GPPs, Texas Instruments DSPs and FPGAs. The GPP version requires an underlying POSIX OS. It has been tested on x86, x86_64 and ARM5 architectures. It integrates a computing resource manager capable to automatically map waveform components into processing devices maintaining real-time deadlines [6]. The manager relies on an abstract metric for measuring the available and required computing resources. This characterization is currently done in equivalent multiply-accumulate operations (MACs). This representation enables predicting the resource consumption of a component on a processor given the measurements on another processor. The metric, hence, assumes equivalent architectures.

ALOE forces a deterministic time-driven scheduling of waveform components, rather than relying on the operating system scheduler. Processing time is divided in time slots. Each component is executed as a single process and instantiated once every time slot (periodically), one component after another. The execution order is determined by the resource manager. If the processor has several cores, the resource manager chooses where to execute each module (according to some metric) and ALOE binds the execution of the process to a single core.

Interfaces are asynchronous and message-based: like POSIX *read()* and *write()* functions, the transmitter calls an ALOE API function indicating a local buffer and the number of bytes to transmit; the receiver calls (asynchronously) another function indicating the number of bytes to receive from the buffer.

Communications between different processors (or cores) require an extra time slot delay, while removing precedence constraints and facilitating parallelism. In addition, ALOE has an option to add a delay for internal communications also, enabling parallel data transfers in bandwidth constrained devices.

B. Measuring the Execution Time

ALOE eases the measuring of the execution time of components: each component is executed once every time slot at highest priority, that is, it cannot be interrupted by other (operating system) processes; the difference between the start and finish times is then the component's execution time. The processor utilization is the component's execution time divided by the execution period (time slot duration). The start and finish times are measured in Linux with a timer of 1 ns resolution (*clock_gettime()* function with monotonic clock). However, the resolution is decreased to 1 μs for compatibility with other platforms (future releases will assume 1 ns). Note that since the scheduling is controlled by ALOE, we do not need non-intrusive measurement tools to measure the processor occupation.

The execution time data is collected by the background daemon EXEC, which is executed on each processor. It runs at the beginning of a time slot, collecting data produced in the previous slot. Each EXEC daemons sends the reported data to the SWMAN daemon, running in a separate processor. SWMAN saves the data from all nodes in the distributed processing network in text files for further processing.

III. Waveform Performance Analysis

This section analyzes the performance of ALOE waveforms. The first test runs a UTRAN bit-level transmitter and receiver for introducing the measurement process. The second test measures the maximum throughput achievable with a simple BPSK modem.

A. Platform Description

The platforms used in both tests are a dual-core Intel Centrino processor and a network of 2 ARM processors connected with TCP/IP. Another processor runs the management daemons (SWMAN, HWMAN, STATSMAN). It receives measurements through a TCP/IP interface and logs and visualizes them. The ALOE measures (at boot) the processor capacity in terms of the equivalent metric "*MAC operations"*.

Single-Core Platform (1x86): The first platform is a dual-core Intel Centrino running at 2.27 GHz with only one core activated. We measured a capacity of 600 Million MACs per second (MMAC/s).

Dual-Core Platform (2x86): The dual-core platform is the same platform as the single-core case, but with both cores activated. Hence, the resource manager will be able to map components to both cores. The platform has an equivalent capacity of 2·600 MMAC/s.

2 ARM Platform (2ARM): Two Marvell ARMv5TE 1.2 GHz processors are connected through a 100 Mbps Ethernet TCP/IP interface. This interface is not adequate for real-time communications as it consumes a lot of CPU time in retransmissions and protocol stack management. However, if the processor has enough resources to cope with both, the signal processing tasks and the TCP/IP stack management, real-time processing may be feasible. Each processor has an equivalent capacity of 200 MMAC/s.

B. Execution Time of UTRAN Components

The first test measures the execution time of the UTRAN components on the 2x86 and the 2ARM platforms. The 2x86 platform is capable of running the waveform at a periodicity of 10 ms. The source generates 3 840 information bits per time slot achieving a throughput of 384 kbps, hence being processed at real-time. The 2ARM platform, however, has not sufficient computing resources to achieve the necessary waveform throughput; the turbo decoder, in particular, consumes 1.15 MMAC per invocation (2 iterations each invocation), whereas each processor has a capacity of 2 MMAC per time slot. The 2ARM is theoretically capable of running the 384 kbps UTRAN in real-time, since the entire waveform needs 2.07 MMAC. The remaining capacity is, however, insufficient for coping with the TCP/IP overhead and, therefore, we need to set the ARM time slot to 20 ms, increasing the processing capacity to 4 MMAC per time slot.

Table I shows the measurements of the execution time for each UTRAN component. The measurements are averaged over 100 samples. They are obtained with the ALOE command *execinfo*. This command also shows the computing demands in the equivalent abstract measure. Although the MAC estimation outperforms a simple frequency scaling estimation ("Time" column in the table), some components still present large differences. Therefore, more appropriate metrics should be considered in future research.

TABLE I
UTRAN EXECUTION TIME FOR 384 KBPS DATA TRANSPORT CHANNEL

Component name	Mean Execution Time (µs)		Maximum kMAC		Estimation Error (%)	
	x86	ARM	x86	ARM	Time	MAC
CRC	104.0	308.2	69	65.4	38.0	5.5
TurboCoder	199.4	1015.4	126	205.8	63.9	38.7
RateMatch	189.9	2006.1	116.4	403.6	82.6	71.1
1st Interl	142.7	550.9	89.4	111.6	52.4	19.8
Segment.	23.4	71.6	15.6	15.2	39.5	2.6
Ch. Mux	112.2	388.3	72.6	78.8	46.9	7.8
2nd Interl.	147.1	787.6	93.6	313.4	65.7	70.1
2nd deInterl.	82.9	758.8	51.6	160.2	79.9	67.7
Ch. deMux	45.9	438.0	30	92.4	80.7	67.5
UnSegment.	36.5	193.1	28.2	42.6	65.3	33.8
1st deInterl	141.7	824.3	111	171.4	68.4	35.2
UnRateMch	96.1	1700.1	60.6	344	89.6	82.3
TurboDec	1827.8	5881.9	1144.2	1187.8	43.0	3.6
CRC Check	100.8	296.5	63.6	67.8	37.5	6.1
TOTAL	**3 250**	**15 221**	**2 071**	**3 260**	**60.8**	**36.5**

Fig. 1 shows the histograms of the turbo decoder and 1st interleaver components for the two processors. In this case 10 000 samples have been generated. We observe that the quantity of values far from the mean is lower for the ARM than for the x86. This is so because the ARM operating system has fewer active services than the x86, resulting in less variability in the execution time. In Section IV.A we show that this variability is also observed in interface latency measurements. We believe this is caused by cache misses or memory bus wait states and, since more background services are running in the x86 Linux system, the probability to experience a longer latency is larger. Note that this kind of variability has a great impact on the real-time.

We find this tool of great interest to identify the distribution of the execution time and its influence to real-time violations (causing missed frames in the digital converter). We ascertain that the main contribution to the variability of the execution time is given by the whole set of different program threads, which is Gaussian-distributed. However, there is another contribution which depends on the platform architecture and increases with the number of software layers in the system, e.g.: memory bus wait states, cache misses, operating system calls, math functions, etc. Therefore, hard real-time resource allocation is unfeasible if several layers of flexibility are added to the system.

C. *Maximum Waveform Throughput*

In this second test we will examine the maximum achievable throughput of a simple waveform.

The BPSK modem consists on 4 components: a bitstream source, a modulator, a demodulator and a sink. This waveform is very simple in terms of complexity. The source generates random bits, the modulator maps each bit to a 16-bit sample whereas the demodulator performs the inverse operation. The sink is a dummy module doing nothing. The signal processing task is very simple; thus, the throughput bottleneck comes from the middleware rather than from the application complexity. This test, hence, gives a figure of the maximum achievable throughput by any application executed on the given hardware platforms.

The throughput is a function of the packet length generated at the source and the execution frequency or times slot. Almost any middleware performs better with large packets, because the relative middleware overhead is lower per byte. Equivalently, low execution periods introduce a relatively higher overhead per-invocation which results in lower performance.

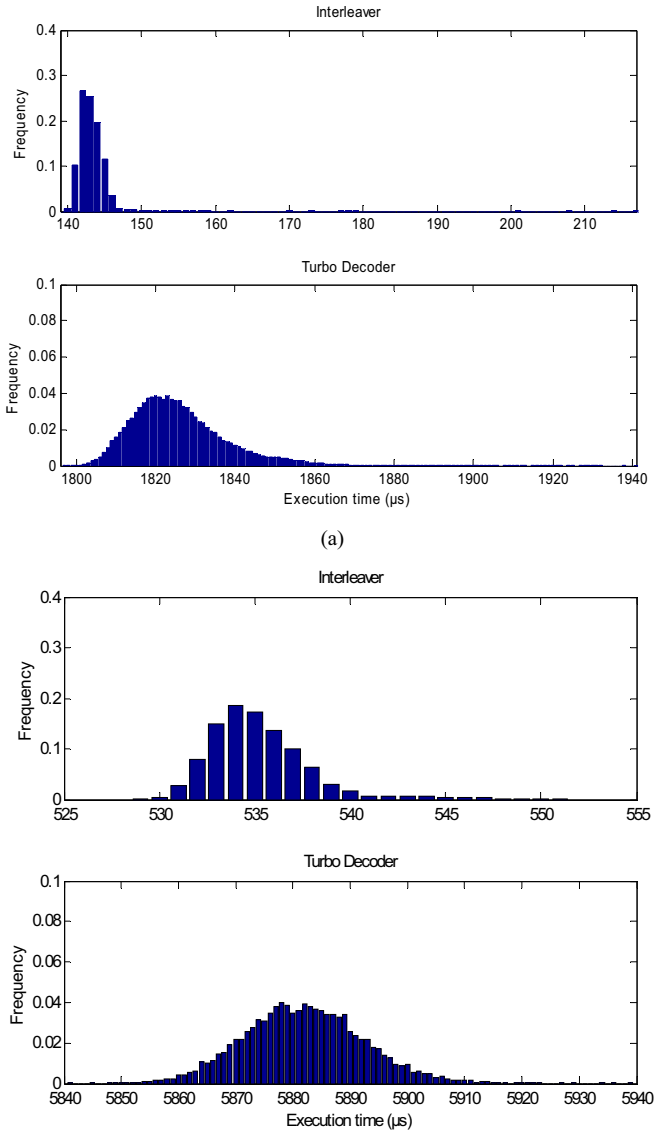

Fig. 1. Histograms of 10 000 measurements of the execution time of the 1st interleaver and the turbo decoder components in the x86 processor (a) and the ARM processor (b).

Fig. 2 shows the maximum packet length the system supports and its associated throughput as a function of the time slot duration. We observe that for short time slots, the performance is lower since the overhead is greater (section IV).

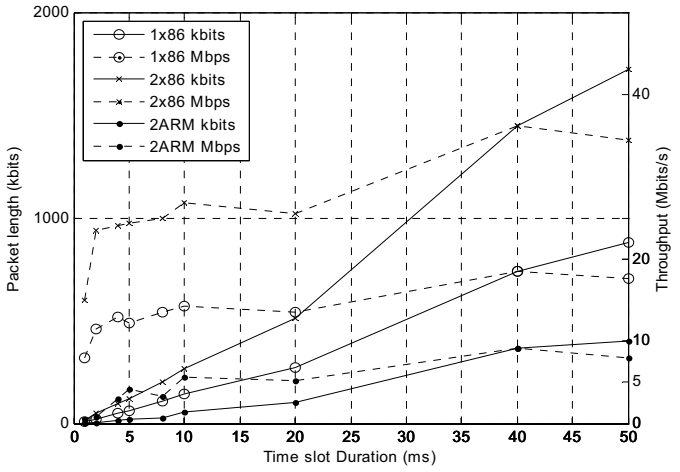

Fig. 2. Maximum achievable packet length (lines) and throughput (dashed-lines) for different time slot durations. Packet length is measured in the source although they are converted to 16-bit samples in the modulator.

The processor utilization in ALOE is rarely 100 % since, given the variability observed in the execution time, this situation would produce real-time failures. Our test consists on gradually incrementing the source packet length until a real-time failure occurs (detected by the EXEC daemon) and, thus, chose the latest sample as the "stable" one.

On the other hand, the speedup increases with parallelism: the throughput achieved with the x86 dual-core almost doubles the achievable throughput of the single-core. Since the resources consumed by the modulator and demodulator are similar, the resource manager maps the source and the modulator to a core whereas the demodulator and sink to the other. In a larger network of processors, with a larger number of components, the speed-up would further increase.

IV. ALOE Overhead Analysis

To be able to predict the real performance of a waveform in a platform, we must first analyse the behaviour of ALOE. Our analysis is similar to that in [7]. We characterize the interface latency and middleware overhead in terms of probability density functions.

A. Interface Latency

ALOE is capable of parallelizing data processing and data transmission. If the system has a direct memory access (DMA) controller, internal interface communications can be performed in parallel thanks to ALOE's pipelined architecture. Interprocessor communications can be parallelized as if an external device is capable of moving data without consuming processing time. Since a message produced by a component at time slot n is not available to the receiver component until time slot n+1, there is no need to wait until the transmission is finalized and DMA can be used (or equivalent mechanisms for external communications). This technique, however, introduces a latency of one time slot per component. Therefore, in order to meet the end-to-end application constraints, the time slot needs to be reduced. We have experienced that, basically in general-purpose processors, some platforms can not achieve low time slot durations and latency requirements cannot be met.

Another approach stops data processing during internal communications and adds an extra time-slot delay for inter-processor communications. Therefore, the end-to-end delay is equal or lower than twice (one for processing and one for transfers) the number of processors in the network. This approach is more appropriate for general-purpose processors since memory bandwidth is rarely a bottleneck.

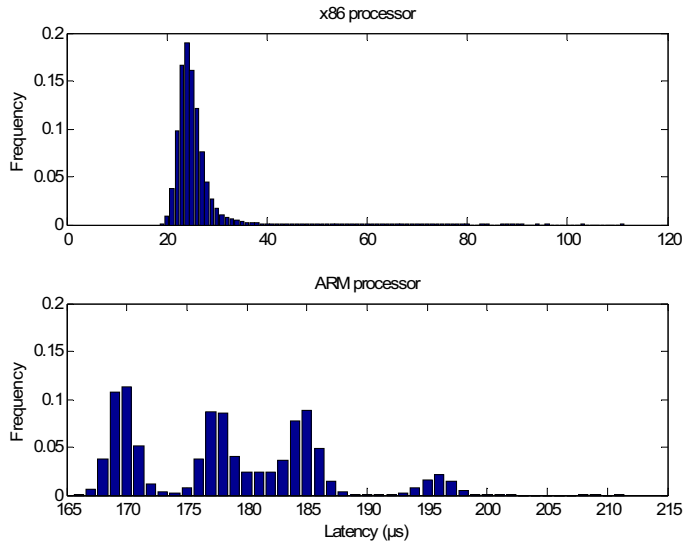

Fig. 3. Distribution of 100 000 measurements of the interface latency with packets of 100 kbits.

Since communications between processors (or cores) are realized in parallel, their latency is constrained by the duration of the time slot – they do not affect application performance. (A different situation is a complex protocol like TCP/IP that needs CPU time to perform computations, not communications). Therefore, the measurement of inter-processor latencies is out of the scope of this analysis.

Internal inter-component interface latencies are measured by ALOE. It measures, for every input and output interfaces, the time the processor is occupied performing transmissions. Since communications in ALOE are asynchronous, we have aggregated the time consumed by the *send* and *recv* functions. Fig. 3 shows the histogram of 100 000 measurements of the time consumed by these functions when transmitting a packet of 100 kbits. Note that although mean latency is smaller in the x86

processor, several values are almost 8 times larger than the mean. As explained before, missed cache accesses or memory bus occupations influence this variability.

These latency results are useful because if communications are not parallelized, the time the processor is transmitting must be subtracted from the total available processing capacity. In addition, internal bandwidth is a resource itself, controlled by the resource manager and thus must be characterized. Fig. 4 measures the interface latency for different packet lengths, averaged over 10 000 samples. As stated before, most middleware perform better with larger packet lengths. An average bandwidth of 2-5 Gbps in x86 and 0.4-0.6 Gbps in ARM is achievable with ALOE for standard packet lengths (Fig. 4).

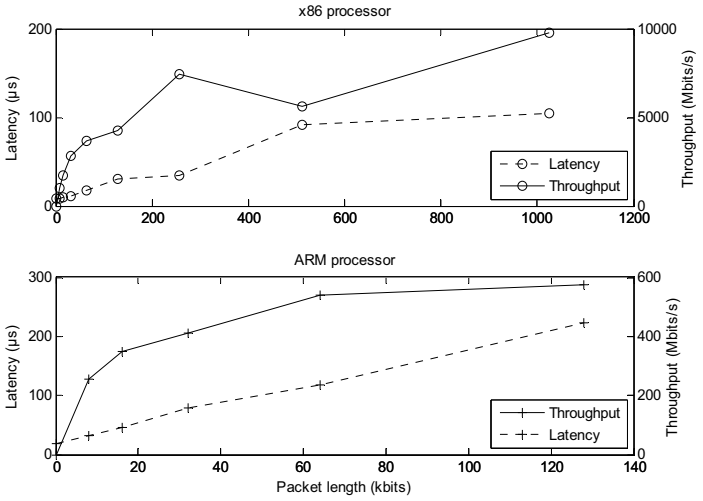

Fig. 4. Average interface latency and throughput for different packet lengths.

B. Time Overhead

This section characterizes the time the processor is occupied performing middleware control tasks. Again, this measurement needs to be characterized since the resource manager cannot allocate this time to the application.

In general, any middleware has a similar overhead model,

$$T_{overhead} = T_{base} + N \cdot T_{module},$$

where T_{base} is an overhead independent of the number of modules, T_{module} the time consumed by the middleware for each running module, and N the total number of modules.

Therefore, both figures (T_{base} and T_{module}) need to be properly characterized. Again, ALOE turns out to be useful for measuring the middleware time overhead since the measurements are available by default. The time overhead per module is measured by subtracting the time the component is running application code (including receiving and sending data) from the total time the component is awake during a time slot.

Fig. 5 shows a histogram of 100 000 T_{module} measurements. The measured average is 5.94 µs for the x86 processor and 24.97 µs for the ARM.

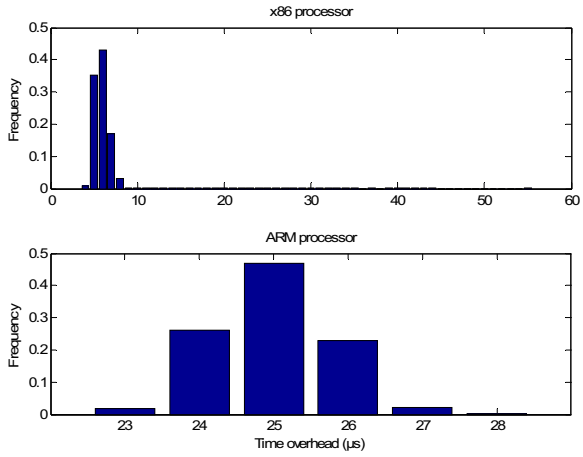

Fig. 5. Distribution of 100 000 measurements of the middleware overhead for every component.

TABLE II
ALOE BACKGROUND PROCESSES OVERHEAD

Daemon name	Mean execution time per time slot (µs)	
	x86	ARM
EXEC	87.12	98.48
STATS	42.38	50.32
SWLOAD	11.47	17.42
BRIDGE	10.71	14.42
FRONTEND	41.01	31.08
TOTAL	**186.69**	**211.72**
SWMAN (opt)	26.02	N/A
STATSMAN (opt)	28.81	N/A

Table II shows the mean time consumed by each ALOE background process. The first four processes must be running on each processor whereas the SWMAN and STATSMAN are optional. The EXEC measures the execution time and controls the real-time execution; the STATS manages and reports the application variable values; the SWLOAD only works during the application loading process; the BRIDGE sends and receives data to and from other processors (not applicable here); and the FRONTEND routes control packets to and from the rest of processors and monitors system status. The two optional daemons, SWMAN and STATSMAN, perform logging and management tasks of execution time measurements and application variables, respectively.

These values are absolute values and, thus, do not give a meaningful idea of how much processor time is dedicated to middleware tasks. The relative processor occupation depends in the invocation period. The more frequently the background and

the component processes are invoked the larger the relative overhead. Fig. 6 shows the relative middleware overhead as a function of the time slot for different number of objects. As expected, short periodicities are difficult to achieve in general-purpose processors without introducing significant overheads.

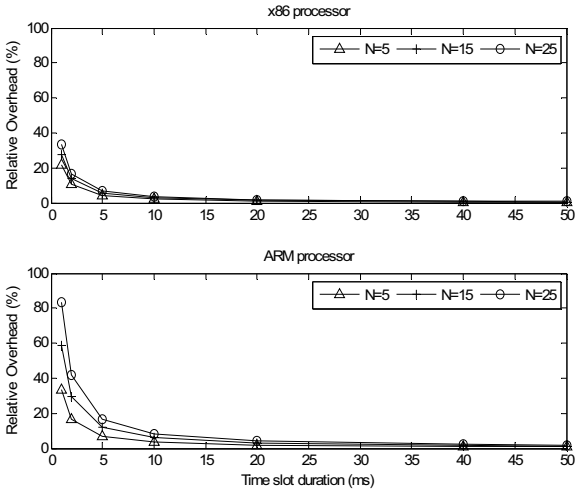

Fig. 6. Relative total overhead as a function of the time slot duration and the number of components.

C. Memory Overhead

The last figure to analyse is the memory overhead. Again, a base memory amount is dedicated to management tasks, whereas another amount is required by each component (the API).

TABLE III
ALOE MEMORY OCCUPATION

	Linux (in Kbytes)	TI DSP (in Kbytes)
EXEC	3 088	42
STATS	3 036	39
SWLOAD	3 856	49
BRIDGE	3 036	33
FRONTEND	3 040	45
TOTAL Base	**16 056**	**208**
ALOE API per comp.	260	21

The Linux implementation consumes a significant amount of memory. Nevertheless, this operating system typically runs on general-purpose processors where memory is not a constraint. The memory budged of digital signal processors is not as large and its utilization needs to be minimized. Table III shows the Linux memory footprint and the memory occupation of the corresponding DSP implementation. Since the implementation of EXEC, STATS, the API, and so forth are platform-independent, the code is the same for both platforms. The differences may then come from the standard C library implementation since the Texas Instruments DSP/BIOS API is much simpler than the Linux standard libraries.

V. CONCLUSIONS

This paper first shows that a pipelined time-driven scheduling has significant advantages in SDR applications: It (1) avoids end-to-end latency uncertainty, (2) relaxes inter-processor communications constraints while intra-processor communications can be easily realized in parallel, (3) enables to accurately measure and monitor the execution times of waveforms and components, (4) exploits parallelization facilitating scheduling and (5) facilitates distributed synchronization.

Second, we have shown that asynchronous message-based interfaces, as opposed to procedure calls, add lower latency with less uncertainty. On the other hand, synchronization must be realized through another process (e.g. time slot) and data serialization is not supported.

Several contributions have presented a similar analysis for the JTRS Software Communications Architecture [8] and GNU Radio [9]. Specifically, references [7], [10] and [11] perform similar tests on both architectures. Future SDR will need to integrate flexibility at all levels: from design to run time. This achievement will be an enabler to produce low cost truly reusable and flexible terminals. However, flexibility cannot be introduced at the expense of more power or CPU consumption if the cost of the overhead overcomes its benefits.

ACKNOWLEDGMENT

This work was supported by the European Commission in the framework of the FP7 Network of Excellence in Wireless COMmunications NEWCOM++ (contract n. 216715).

REFERENCES

[1] van Berkel, C.H.; , "Multi-core for mobile phones," Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE '09. , vol., no., pp.1260-1265, 20-24 April 2009

[2] M. Armbrust, et al. “Above the Clouds: A Berkeley View of Cloud Computing”, Tech. Rep. UCB/EECS-2009-28, EECS Department, University of California, Berkeley (February 2009).

[3] L. Pucker, “Implementation of a Shared. Resource Model in a Tactical Radio System”, IEEE MILCOM '04, no. 1, Oct 2004, pp. 734–34

[4] Gomez I., et al. “A lightweight operating environment for next generation Cognitive Radios,” DSD 11th Euromicro Conference, 2008

[5] ALOE Web Site, http://flexnets.upc.edu/trac

[6] Marojevic, V., et al. “A computing resource management framework for software-defined radios”, IEEE transactions on computers, October 2008, vol. 57, num. 10, p. 1399-1412.

[7] G. Abgrall, et al. "Latency estimation due to Middleware used in Software Defined Radio platforms", 6th Karlsruhe Workshop on Software Radios, WSR’10, pp. 105-112, 3-4 March 2010

[8] JTRS SCA v2.2.2 http://sca.jpeojtrs.mil/

[9] "GNU Radio – The GNU Software Radio", http://www.gnu.org/software/gnuradio

[10] A. Palomo, et al. “Software Defined Radio Architectures Evaluation”, SDR Technical Forum, Washington DC, October 2008

[11] F. Ge, et al. “Software Defined Radio Execution Latency”, SDR Technical Forum, Washington DC, October 2008