Tcl interpreter implementation for embedded systems based on LEON architectures

Rubén Marrero[1], Javier Miranda[1], Gustavo M. Callico[1], Jorge Amador[2], Roberto Sarmiento[1]

Abstract—**The European Space Agency (ESA) makes great efforts to harmonize the software technology that they use. In fact, many standards and concepts are defined by the Agency. In this context, the Agency has defined the concept called On-Board Control Procedures (OBCP) that consists on a powerful way to control spacecraft and to implement on-board functions.**

The main purpose of this work is to develop a Tcl language interpreter that could be functional in the LEON3 architecture that ESA uses in its missions. This development allows analyzing the possibility of using interpreted languages to perform the tasks assigned to the flight control software.

In order to develop this interpreter, we have started from an existing one, Jim, which has been rewritten and simplified to optimize the use of resources such as the decrease of memory and the increase of execution speed. After that, the interpreter was implemented in a LEON3 simulator and in a LEON3 emulator on a FPGA.

During this process, we have generated a set of tests in order to debug the developed system and we have concluded that the interpreter is functional for a selected Tcl language subset.

Keywords—**On-Board Control Procedures, Tcl, LEON3, Emulation, FPGA.**

I. Introduction

IN the last years, the aerospace sector has slowed down in adapting to the new communication systems and also in the user-machine interface, due to its traditional habit in using mature technology. Even though, trying to maintain this principle needs to incorporate a minimum flexibility to allow competing at an industrial level.

Facing this situation, the European Space Agency (ESA) inaugurated its own program in the development of microprocessors. Its objective is to facilitate the programmer tasks in the final steps of the product, independently of its final field of application: satellite, space shuttle, etc.[1].

In this context, specifically in the interface context, the On-Board Control Procedure (OBCP) concept arises. It consists on a set of programmed procedures on Earth, which gives a great autonomy to spacecrafts. It has been used ad-hoc during the last 25 years, but ESA intends to standardize it [2].

[1] Research Institute for Applied Microelectronics (IUMA), University of Las Palmas de Gran Canaria (ULPGC). E-mail: {rmarrero, jmiranda, gustavo, roberto}@iuma.ulpgc.es

[2] Head of GSTP Planning and Implementation Section, European Space Agency (ESA). E-mail: jorge.amador.monteverde@esa.int

II. Background and objectives

This work arises as a collaboration between ESA and the Integrated System Design Division (DSI) from the Research Institute for Applied Microelectronics (IUMA) of the University of Las Palmas de Gran Canaria (ULPGC), with the intention of creating an interpreter for a subset of the Tcl language (Tool Command Language) running on embedded systems based on LEON architectures that ESA is currently using.

The choice of Tcl is based on some of its features: the batch execution mode, the simplicity of its syntax, its ability to be easily extended and its interpreted code that can be created and modified dynamically.

The overall objective of this project is to analyze the possibility of using interpreted languages to ease communication and control of spacecrafts from ground.

The ultimate objective is summarized in the development of a Tcl interpreter, functional in the LEON3 architecture for a subset of the language proposed by the ESA and its subsequent validation in that architecture.

Given the characteristics of Tcl and given the requirements that are subject to space applications, it is necessary to establish a subset of the Tcl language, besides making the interpreter able to be executed on the LEON3 architecture. As an embedded application, it is necessary to optimize the resources used by the interpreter, such as the executable size and the dynamic memory consumption.

Furthermore, it is necessary to generate a test suite in order to debug and verify the interpreter during the development and implementation stages, and to validate the interpreter running on the LEON3 architecture.

III. LEON3 architecture and ESA software

The LEON project [3] arises in ESA as a successor to the ERC-32 architecture with the aim of developing a radiation-resistant processor, modular, easily portable, with standard interface to execute up to 100 MIPS.. In addition to this, the new design, described in VHDL, would be licensed under General Public License (GPL), allowing System-on-Chip (SoC) integration in a simplified way. This new architecture was based on SPARC v8, defined through an IEEE standard. Furthermore, the connection of additional modules will be possible through an AMBA bus system.

At first, it was necessary a previous development where it could demonstrate its functionality by implementing a minimum number of interfaces and functions. This first prototype was called LEON. Once this was verified, the development of a new and more complete processor began, trying to add new features as a floating point unit and new interfaces, leading to LEON2 and LEON3.

LEON3 is the one which will be used finally in the ESA missions, with an architecture based on the SPARC V8 instruction set and whose main innovations are [4]:

- Multiway cache.
- PROM/SRAM/SDRAM controller.
- AMBA buses (AHB and APB).
- Advanced debugging on chip system.
- Power down mode.
- SPARC v8e extensions.
- 7-stage pipeline: Fetch, Decode, Register Access, Execute, Memory, Exception, Write.

On the other hand, ESA is making great efforts to harmonize its software technology defining many standards and concepts [5]. The application interfaces in space systems, which is developed to support daily operations after the deployment of such systems, is one of the main objectives. In this context, the On-Board Control Procedure (OBCP) concept is defined by ESA [2], and it is the main reason of this project. ESA has published many drafts which aim to study the convenience of Java programming language in order to substitute languages such as C and ADA, but some features are strongly disputed, because they are not needed in aerospace context: class loader, just-in-time compilation, etc.

Thus, this paper will attempt to demonstrate the applicability of Tcl as on-Board control software, in a glue-language manner.

IV. TCL & JIM

Tcl [6], is an interpreted language created by John Ousterhout in 1988. The main design goal was to create an extremely simple syntax language, in order to facilitate its learning, without losing functionality and expressiveness. At the same time it could be integrated in other applications and could be easily extensible in a shell scripting way. Tcl scripts can be more compact and readable than other scripting languages, so the code is able to be maintained easily over time.

Tcl is modular, scalable, cross-platform and can be used as an interactive shell or shell script. The language is in continuous evolution due to a large community of developers, highlighting the Tcl Core Team in charge of the interpreter core. This has been possible largely because it is a free software project, although it also has some commercial support packages.

Its scalability allows adding new commands written in C, C + + or Java to the standard interpreter.

Among the several virtues that Tcl holds as a general purpose interpreter it is worth to mention that there is a wide variety of systems that can run the interpreter, it is included in real-time operating systems such as *VxWorks*; and the fact that it allows much of the modern forms of programming: modularization like subroutines or libraries, standard control structures, exception handling and various types of variables like associative arrays.

On the other hand, Jim is a small open source re-implementation of the Tcl interpreter, very stable and modular, with a wide range of extensions [7]. It supports a large subset of the built-in commands of Tcl interpreter and it is cross platform suitable, POSIX compliant and able to run on various operating systems like Linux, FreeBSD, QNX, eCos or Windows.

It combines features of smaller and earlier versions of Tcl (6.x) and modern features of the latest versions (7.x and 8.x), while adding its own characteristics.

There exists another Tcl interpreters for embedded purpose such as TinyTcl, but it is not being maintained and documented as well as Jim.

V. DEVELOPMENT TOOLS

In this section it is introduced some of the tools that have been used during the development process.

A. *BCC Compiler*

Bare-C Cross-Compiler is a cross-compiler for LEON2 and LEON3 that includes a wide range of tools based on GNU ones [8]. It allows compiling multithreading applications.

B. *TSIM Simulator*

TSIM is an instruction level simulator capable of emulating both ERC32 and LEON based systems. The tool started out as free, but over time, Gaisler Research decided to remove the old versions and start selling new ones [8]. As simulator, it provides high accuracy and a level emulation cycle time, giving a performance of up to 45 MIPS.

In LEON mode, TSIM simulates the full functionality of the microprocessor, including caches, on-chip peripherals and a memory controller. TSIM is currently distributed for Linux, Solaris and Windows.

C. *GRMON Monitor*

GRMON is a monitor for the LEON processors debugger [9]. Once the system is dumped to an FPGA or an ASIC connected to a PC, GRMON provides a graphical interface in which it is possible to: (1) watch every read-write access to all registers and memory; (2) set breakpoints and watchpoints in the code; (3) connect with GDB; (4) analyze applications performance; (5) downloading and execution of applications.

GRMON, as TSIM, is a propietary software distributed for Linux, Solaris and Windows.

D. *Valgrind*

Valgrind is a set of free tools to debug the memory management of any application [10]. The main tool, at least in this work, is *memcheck*, which acts as an intermediary for all memory requests to the system and can detect: (1) not allowed memory access; (2) use of uninitialized variables; (3) memory leaks; (4) illegal frees; (5) overlapping source and destination blocks.

VI. DEVELOPED INTERPRETER: TOBI

Tobi (Tcl On-Board-Interpreter) is the interpreter of a subset of the Tcl programming language, designed in this work to meet the ESA requirements for on-board systems. This subset of the language is a set of commands for list handling (*concat*, *lappend*, *lindex*, *linsert*, *list* and *llength*), mathematical expressions (*expr* and *incr*), control structures (*eval, for, foreach, if, return,* *switch* and *while*), I/O (*open, close, gets, puts* and *flush*) and variable & procedure Management (*global, proc* and *set*).

The main objective is that Tobi could be validated to be used in aerospace applications and thus be a candidate to the OBCP core currently claiming by ESA for their LEON3 architecture.

The main criteria for the development were:

- To limit its functionality to the selected tcl subset.
- To reduce its memory consumption.
- To respect the design rules followed in *tclsh* and Jim cores (name and type of data structures, functions, etc..).
- To make it compilable for the LEON architecture while delivering high performance.

After analyzing the *tclsh* and Jim implementations, it was decided to develop the prototype from Jim v0.64 sources.

A Venn diagram is shown in Fig. 1, representing the sizes and functionalities of each interpreter.

Tclsh is the Tcl interpreter that consumes more memory and also provides more functionality. Jim, as a reimplementation of *tclsh*, combines its own specific functions and those inherited from *tclsh*. Tobi, as a reimplementation of Jim, provides a subset of Jim functionality (including those inherited from *tclsh*) and also includes its own functions such as I/O commands that have being reimplemented and even its capacity of being compiled on LEON architecture. As shown in Fig. 1, Tobi is the smallest Tcl interpreter in functionality and resource consumption, making it a very suitable option for embedded applications.

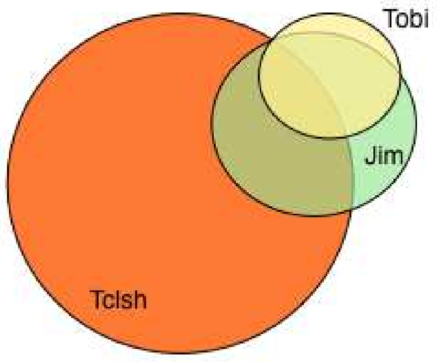

Fig. 1. Venn diagram representing the size and functionality of tclsh, Jim and Tobi.

VII. PERFORMANCE ANALYSIS

This section verifies compliance with the objectives through a comprehensive comparison of Tobi against Jim and *tclsh* v8.5.

The executable sizes, dynamic memory consumptions and speed will be compared. This comparison is carried out in the development platform (Linux Ubuntu 10.10) because *tclsh* and Jim interpreters cannot be compiled into the LEON architecture. The compiler used was gcc (Ubuntu / Linaro 4.4.4-14ubuntu5) 4.4.5. Furthermore, this comparison is linked to compiler options and different TOBI configuration parameters at compile time by using preprocessor directives that enable or disable certain parts of the code, such as optimizations that improve expressions evaluation, substitutions and loops through prediction routines.

A. *Executable sizes*

On the Linux Ubuntu platform, the resulting sizes (KB) for Jim and Tobi with and without optimizations, but with the same compiler options (-O0, -O1, -O2, …), are shown in Fig. 2.

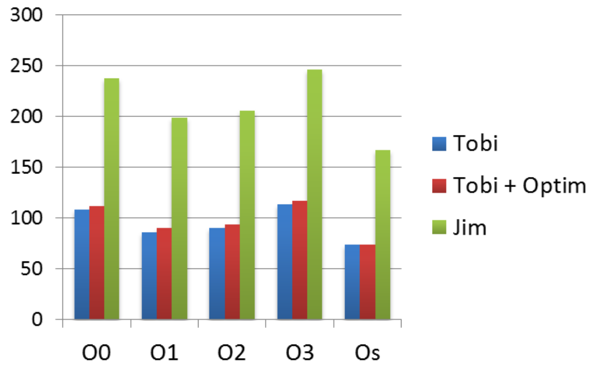

Fig. 2. Size (KB) on Linux Ubuntu 10.10.

The difference in size between Tobi, with and without optimizations, is of 4KB, except when the compiler option is –Os, in that case sizes are equal.

However, Tobi size is around 100KB, 0.44 times the Jim size, reaching a minimum of 74 KB with the –Os compiler option. As reference, *Tclsh* v8.5 installation occupies 4.5 MB on Linux Ubuntu platform.

The difference of sizes between Tobi and Jim are due to the reduction of superfluous functionality that implies an inferior amount of code lines to compile in order to support a reduced set of commands.

Moreover, in Fig. 3 is shown the obtained TOBI sizes using the cross-compiler *Sparc-elf-gcc* (BCC 4.4.2 release 1.0.36b) 4.4.2 for the LEON architecture, as well as the obtained sizes if it is used the *mkprom* tool included in the BCC package, that consists in the compressed boot image of the interpreter, that will be decompressed in RAM memory once it is booted on the final system.

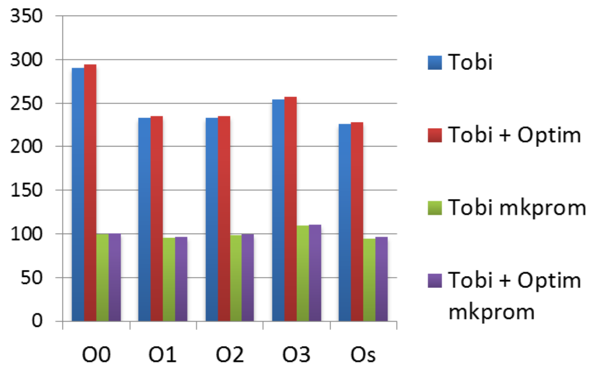

Fig. 3. Size (KB) on LEON

The tendency of sizes showed in Fig. 3 is the expected one. The smallest size is obtained using the –Os option, and the difference of sizes using or not optimizations is around 2-4 KB. Furthermore, the boot images are compressed in a factor that varies from 2.4 to almost 3.

However, the space occupied by these executables is greater than the space occupied in the development platform, due to the cross-compiler that includes in the executable some static libraries that could need the application; in the development platform these libraries are dynamically loaded by the operating system.

B. Dynamic memory consumption

Decreasing the executable size could not be an objective if the amount of dynamic memory consumed by the application at runtime is not reduced.

This section makes a comparison of dynamic memory usage during the execution of some of the scripts that Jim uses to check the performance of the interpreter. Such scripts can be found on the Jim developer's website [7]. This analysis was conducted through the application Valgrind (Valgrind-3.6.0.SVN-Debian), which also allows checking that there is no memory leaks in the interpreter, at the same time validating the memory management. The comparison was made with the-O2 compile option for Tobi and Jim.

Table I shows the dynamic memory consumption for the most significant Jim scripts. An empty script ("empty.tcl") has been added in order to check how much memory is used passively by the interpreter itself.

TABLE I

DYNAMIC MEMORY CONSUMPTION (KIB)

	Tobi	Tobi + Opt	Jim	Tclsh8.5
empty.tcl	8	8	52	332
whilebusyloop.tcl	9	9	53	332
miniloops.tcl	11	11	54	332
use_repeat.tcl	12	12	54	332
fibonacci.tcl	5231	5231	5273	332
expand.tcl	11327	11327	11370	9546

It can be seen the memory savings achieved by Tobi over Jim, especially in iterative algorithms ("whilebusyloop.tcl", "miniloops.tcl" and "use_repeat.tcl"). The differences are primarily due to the greater functionality offered by Jim that needs a greater number of structures in memory.

On the other hand, if the execution is recursive ("fibonacci.tcl" and "expand.tcl") differences between Tobi and Jim are no remarkable. However, this is not the kind of application that it is expected to find in a flight control software.

C. Execution speed

In order to determine and compare the execution speed, many intensive tests offered by Jim's developer [7] were executed, intended to check how fast the different interpreters under study for the various compilation options were.

In Fig.4 and Fig.5 are shown the most representative results when running the iterative test "whilebusyloop.tcl" and the recursive test "fibonacci.tcl", which are quite intensive in computation, although our application is not intended to execute this kind of applications because it will be more dedicated to control than computation.

Tobi without optimizations is always slower than Tobi with optimizations and Tobi with optimizations is faster than Jim, except in -O1 and -Os cases. *Tclsh* is the fastest as it was expected.

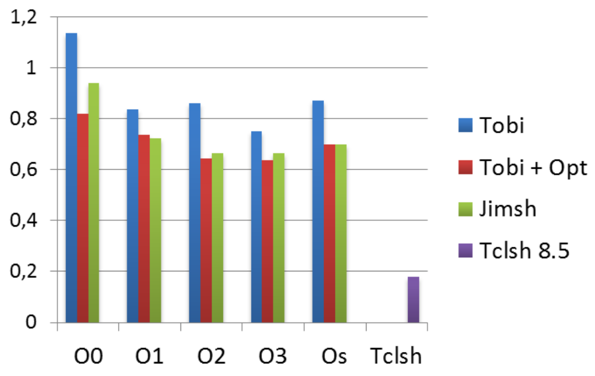

Fig. 4. Time (seconds) – iterative script "whilebusyloop.tcl"

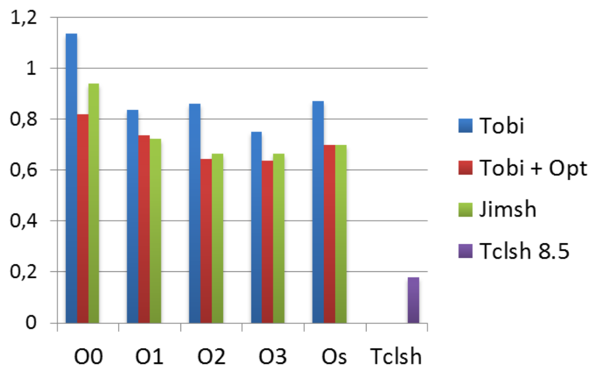

Fig. 5. Time (seconds) – recursive script "fibonacci.tcl"

VIII. FUNCTIONAL TESTS

Tobi uses a different method than the generic one (Fig. 6) used for verification during the development stage, relying on other Tcl interpreter (Fig. 7). In this way, any issue in the test infrastructure that could hide functional misses is avoided. Furthermore, this infrastructure could need some commands that are not included in TOBI. The used tests were included in Jim v0.64 and Tclsh v6.7 collections.

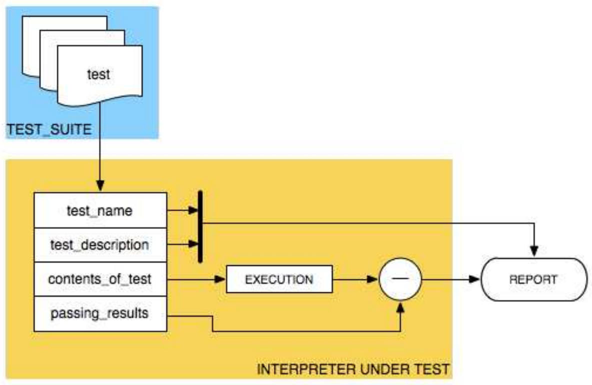

Fig. 6. Generic test scheme

However, when Tobi is tested on the final system, it is used the generic method (Fig. 6) with tests included in Jim v0.64 collection, because there is no other reliable Tcl interpreter that could be executed on LEON3 architecture.

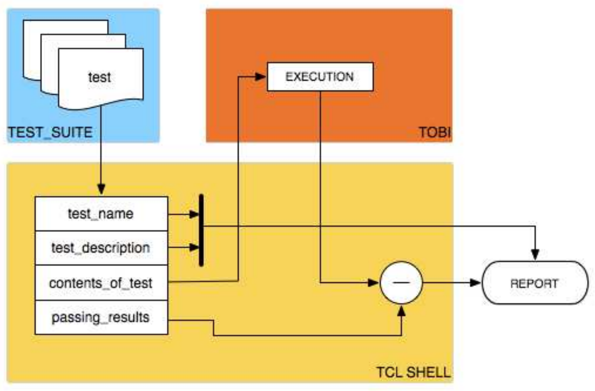

Fig. 7. Modified test scheme for development stage

Thereby, the obtained results were those expected, i.e., any test that needs commands that Tobi does not support fails. In that way, 76.17% of Jim test collection and 42.11% of Tclsh test collection were passed.

IX. Simulation & emulation on LEON3

Tobi has been executed and tested over an operating system in a FPGA that emulates LEON3 architecture. It has been done in a 4-stage process:

1) Tobi verification on a simulator.
2) LEON3 integration in the FPGA.
3) Tobi emulation in the FPGA without operating system.
4) Tobi emulation in the FPGA with operating system.

A. Tobi verification on a simulator

In order to perform a first test to verify Tobi, the commercial emulator TSIM2 developed by Gaisler Research for LEON and ERC32 architectures was used.

Thereby, Tobi was adapted in order to execute some tests from the Jim package without operating system, but using the generic method shown in Fig. 6, including tests in the Tobi binary file. The results were successful.

B. LEON3 integration in the FPGA

The LEON3 VHDL sources and libraries were adapted and integrated in a ML507 [11] board that includes a Virtex-5 FPGA. The editing and adaptation of these sources is carried out in the Xilinx ISE 2.1 tool, that has also been used for the simulation phase (integrating simulator Modelsim SE 6.2), synthesis, mapping and routing. Finally, using this tool the FPGA was programmed successfully.

The design occupies 58% of the FPGA slices, so it still has enough room to add new modules. The percentage of I/O pins used is 47% and, from the total available memory on the FPGA, it has taken only 13%, which corresponds to a total of 720Kbyte; the rest of memory is taken from the external RAM.

C. Tobi emulation in the FPGA without operating system

After completing the integration process, the application was introduced into the microprocessor. The usage of GRMON debug monitor, via JTAG, made it possible to access to read-write cycles on the chip bus allowing verifying the proper behavior. The emulation was successful.

D. Tobi emulation in the FPGA with operating system

The last step in this verification process was checking the application on a operating system. In order to perform this task, TOBI and operating system sources were compiled together, obtaining a single image to dump on the microprocessor.

The operating system used was SnapGear Embedded Linux that is based on a set of source packages that contains the Linux kernel (2.6), libraries and some applications to develop Linux-based embedded systems.

The tests performed in the FPGA using the operating system were also correct.

X. Conclusion

The main objective of this work was to develop a prototype of a Tcl interpreter, functional on LEON3 architecture for a subset of Tcl language, and its subsequent validation in an FPGA.

Adapting the code to make it compilable for LEON3 architecture was achieved by emulating the interpreter on the TSIM simulator, which simulates the architecture. Then the interpreter was implemented in a LEON3 FPGA-based architecture with no operating system, and then with Snapgear Embedded Linux operating system.

Finally, we conclude that the main objectives of this project have been achieved since the desired performance of this interpreter have been reached, and even we have demonstrated that this methodology might be extrapolated to another interpreters for embedded purposes.

References

[1] Gaisler, J. "*The LEON Processor User's Manual*" v2.3.5, Gaisler Research.
[2] ECSS Space Engineering "*Spacecraft On-Board Control Procedures*" ECSS-E-ST-70-01C, 2010.
[3] Clarke, P. "*European Space Agency launches free Sparc-like core*". EETimes. 2000.
[4] J. Gaisler, S. Habinc, et al. "*GRLIB IP CORE User's manual*" v1.0.19. 2008.
[5] ESA "*European Space Technology Harmonisation Technical Dossier*". 2010.
[6] *Tcl Developer Xchange:* http://www.tcl.tk
[7] *The Jim Interpreter: A small footprint implementation of the Tcl programming language :*http://jim.tcl.tk
[8] Gaisler, J. "*BCC – Bare-C Cross-CompilerUser's Manual*" v1.0.29, Gaisler Research.
[9] Gaisler, J. "*GRMON User's Manual*" v1.1.32, Gaisler Research.
[10] *Valgrind* http://valgrind.org.
[11] Xilinx Inc. "*ML505/ML506/ML507 Evaluation Platform: User Guide. XILINX VIRTEX-5*" 2009.