Performance Improvements in the ESOC Emulator

Jose E. Marchesi

Terma GmbH, Europaplatz 5, 64293 Darmstadt, Germany
jco@terma.com
jemarch@gnu.org

Abstract **– While the ESOC Emulator is the best performing ERC32 emulator currently available, we can identify several reasons why performance improvements would be of benefit to ESOC and other users. This paper explores some possibilities on improving the performance of the ESOC Emulator, and other improvements that could be useful in the future.**

INTRODUCTION

We can identify at least three reasons why performance improvements would be of benefit to the users of the ESOC emulator:

- Current and future missions at ESOC are requiring higher and more demanding Emulation power in terms of more powerful processors, and more processors, to emulate i.e. Constellation simulators.

- There is an increasing demand in operational simulators to emulate more software running on processors rather than writing functional models, for example Mass Memory and Payload sub systems, as well as parallel emulation of AOCS and Basic OBSW.

- A Leon emulator could be based in the current ESOC Emulator, expanding the current code base to cover the SPARC v8 instruction set.

The first part of this paper briefly introduces the ESOC emulator, a fascinating piece of software that has been used for years to emulate the Operational Simulators at ESOC, and that is full of potential. A brief introduction of the effort to increase its performance: the relocation of the Emulator to a real 64-bit RISC hardware platform – SPARC v9. And finally, the handling of mapped-memory input-output operations and its impact on the performance of the simulations is discussed.

Please note that in diagrams featuring Ada packages a dashed line denotes the instantiation of some generic package by some other package. The direction of the relationship should be evident.

OVERVIEW OF THE ESOC EMULATOR

The ESOC Emulator (also known as SM_EMU_ERC32) is a software emulator of the ERC32 family of processors and some of its accompanying devices, including the emulation of co-processor and floating-point instructions. It is used to drive the operation of Operational Simulators, running the on-board software of the simulated spacecrafts.

The ESOC Emulator is composed by two main components: an emulator **core** and an emulator **shell**. The emulator core implements the main functionality of the emulator i.e. the fetching, decoding and emulation of the instructions, management of interrupts and timers, etc. The emulator shell is the interface between the user of the emulator and the emulator core. It provides a set of services in the form of procedure calls. Those services include the execution of a given number of instructions, access to the emulated CPU registers, a debugging interface, warm and hard resets, the raising of interrupts and traps, scheduling of events, etc.

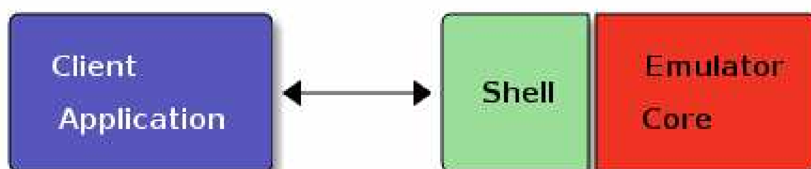

Fig. 1. Client apps communicate with the emulator using the emulator shell.

The emulator core is written in assembly language due to performance reasons (described in the next section). However, assembly language is non portable, and the relocation of the emulator to new machine architectures (or new versions of some existing hardware architecture) would require tremendous amounts of work. For that reason the emulator core is written in a RISC like abstract instruction set, or AIS. The EMMA macro assembler (shipped with the emulator) can translate AIS programs into very efficient native assembly programs provided that the target physical machine implements a typical RISC architecture. That includes:

- Plenty of general-purpose registers to store both integer values and floating-point values. At least 32 integer registers of 64 bits are required.
- Register –to-register and immediate-to-register three operands instructions.
- Load/store instructions for half words, words and double words.

In addition to provide machine-independence, the EMMA macro assembler eases the development of assembly language programs by providing facilities such as the allocation of registers, management of labels, program sections, etc. The emulator core makes an intensive use of those facilities.

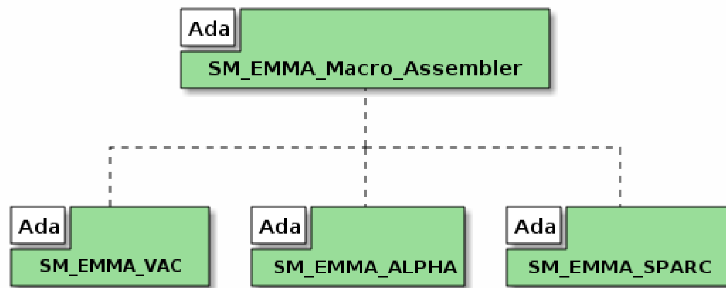

Fig. 1. EMMA macro assembler packages.

The EMMA macro assembler is available in the form of several Ada packages. (see Figure 1). The SM_EMMA_Macro_Assembler package provides generic facilities and internal services used by the backends. Each backend implements a set of generic procedures corresponding to the instructions defined in the AIS. The execution of those procedures will generate different code, depending on the specific backend used.

The ERC32 emulator generator is a generic Ada package that makes use of the EMMA AIS. The user can (and should) customize many aspects of the emulated machine by tailoring the generic package. Those customizations are mainly hooks that will get executed when some event arises, such as the reading or writing of some memory areas. Once tailored, the resulting package can be compiled along with EMMA to build an Emulator Generator binary. See Figure 2.

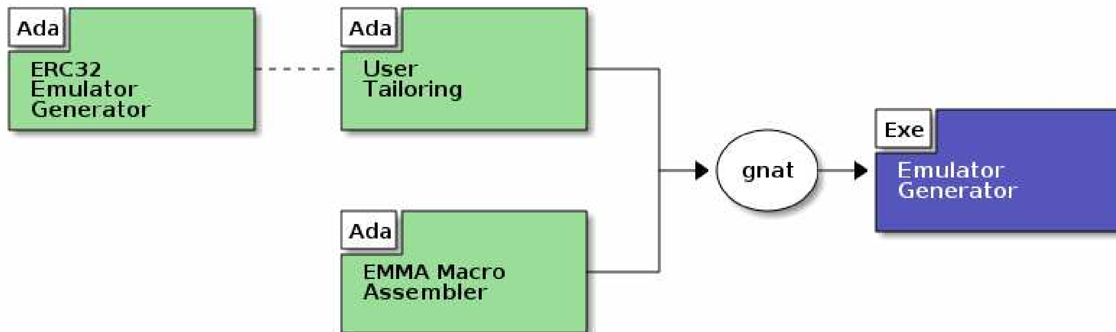

Fig. 2 Building the Emulator Generator

The emulator shell is a generic Ada package providing the services described above. It must be tailored by the user. The combination of an emulator core (generated by the Emulator Generator) and a tailored emulator shell conforms a complete Emulator that can be integrated in an operational simulator. The resulting Emulator library is usually a static archive containing position independent code. See Figure 3.

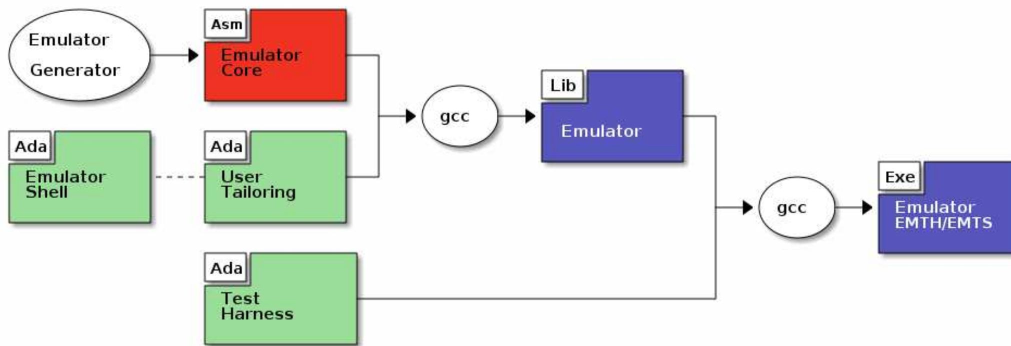

Fig. 3 Build of the emulator library and the test harness.

A test harness command line application is also included with the ESOC Emulator suite. It implements a fairly complete scripting language.

RELOCATING THE ESOC EMULATOR TO SPARC

As we mentioned in the previous section the core of the Emulator is in charge of implementing operations that are quite intensive in terms of CPU usage. Those operations include:

- Fetching, decoding and execution of the emulated instructions.
- Management of the address space of the emulated processes, including memory attributes and the handling of memory-mapped input-output.
- Emulation of interrupts and traps.
- Scheduled events and timers.

Given the CPU requirements of the emulator core it was decided to write it in assembly code instead of using a high level language such as Ada or C. Even if it is usually a real challenge for a human to compete with modern optimizing compilers such as gcc in terms of the performance of the generated code, CPU emulation is one of the few fields where a significant advantage can be achieved by a carefully hand crafted implementation.

Early versions of the ESOC emulator used to generate emulator cores written in the Alpha assembly language. Those cores could run on AD164 co-processor cards attached to personal computers. Each AD164 card featured an EV5 Alpha processor. The performance achieved by those systems was quite good, reaching mips/Ghz ratios of 25, meaning that a 1Ghz host machine would be needed to emulate 25 million of instructions per second.

Unfortunately, at some point it was decided to port the emulator to IA32 and AMD64 systems. Those systems are CISC architectures and thus are not suitable for a native EMMA backend. A backend generating portable C called VAC (Virtual Alpha in C) was written and added to the EMMA distribution. VAC is the backend used today in operational simulators. The native back ends, generating Alpha and MIPS-64 code, are no longer used.

There is a performance penalty in the usage of the VAC backend. In the words of Alan Dartnell: "It should be noted that while an emulator written using the EMMA AIS may be very efficient when the AIS is translated into 64 bit RISC instructions, this is not the case when the AIS for VAC is used as in this case the emulator code in effect is modeling an Alpha like processor emulating the emulated processor" [2]. How big is that performance penalty? Based on measures taken using the AD164 cards and given the similarities between the 64 bit RISC architectures, it has been estimated that the performance of a single processor RISC system clocked at 1.5 GHz would be in the order of 100 mips [1].

The previous estimation was made in the context of the Herschel-Planck operational simulator. At that time the idea of using a native AIS implementation targeted at a modern RISC machine was considered, but then discarded because the simulation infrastructure (namely SIMSAT) was not able to run in modern RISC architectures such as SPARC or MIPS.

Some months ago we considered at Terma the possibility of writing a native AIS implementation for a modern RISC architecture. The idea was to generate an emulator core in native assembly and measure the achieved performance in a modern machine. After considering the different possibilities we choose the SPARC architecture. The reasons for choosing SPARC include:

- The availability of very powerful SPARC machines, like the medium and high-end servers sold by Sun/Oracle.
- It is a stable architecture in terms of market. The sad fate of the Alpha architecture, that was bought and then killed by Intel in favor of the Itanium, is unlikely going to happen to SPARC.
- The Open SPARC initiative may promote the development of third party SPARC-based systems.
- The other widely used RISC architecture, MIPS, is used in low consumption and portable devices. Those devices are not appropriate for running CPU intensive software like the ERC32 Emulator.

The SPARC native AIS backend was completed in two months, and the process of designing and writing it was quite an interesting, instructive and entertaining activity. Packages were developed to cover the full range of supported AIS instructions, including routine calling operations, branch instructions, byte operations, float functions, integer operations, loads and stores, logical operations, shift operations, conditional integer moves, constant pools, data directives, scaled integer operations and synchronization operations.

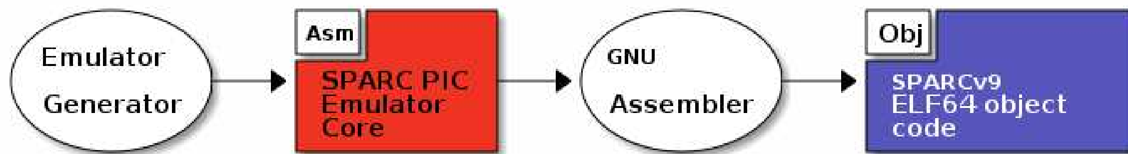

Fig. 4 Generation of an emulator core for SPARC.

The superb EMMA design helped a lot, and made it possible to port a complex assembly application like the Emulator core in a relatively small time. The availability of the existing Alpha and MIPS back ends was also quite useful: problems like the endianess and the delay slots were already handled there and we were able to reuse the same solution. However we had to face some interesting challenges, many of them derived from peculiarities of the SPARC architecture. A brief list follows:

- **Windowed registers architecture.** SPARC systems, unlike other RISC architectures like Alpha and MIPS, organize most of their general purpose registers in overlapping windows. That makes it possible to slice the window every time a routine is called, avoiding the need of saving the value of the registers. The size of the window is enough to provide 32 integer registers at any moment, a few of them reserved for special purposes (like the stack pointer and the stack frame).
- **Limited range of values in immediate operands**. Most of the SPARC immediate-to-register operations accept 13 bit signed integers. That makes it necessary to use a temporary register to hold 32 bit constants. On the other hand branch instructions use signed 22 bit signed immediate values to hold the destination addresses, so additional temporary registers are needed there. The consequence is that a one-to-one instruction translation can't be used for most of the branch operations. The same applies to the load/store operations.
- **The stack bias**. The SPARC v9 Sys V ABI mandates the usage of a stack bias to differentiate processes running in 32 bit mode from processes running in 64 bit mode. That had to be taken into account in the implementation of the routine calling conventions and the registers saving operations.
- **Rigid stack management**. The SPARC architecture is designed to promote the usage of registers to hold the intermediate results of computations, as opposed to save those values in the stack. As a consequence the design of the stack frame in SPARC ABIs is quite rigid: at any time there should be enough space allocated in the top of the stack to hold the values of the registers. In the occurrence of a window overflow or some other trap, the current values of the registers are copied there before to jump to the trap management routine.
- **Position independent code (PIC)**. The emulator requires the usage of position independent code, since the object code of the emulator core will usually be included in a shared library. That complicates the code generation.

The EMMA SPARC backend generates position independent code SPARC v9 assembly code that is compatible with the System V application binary interface. The GNU Assembler gas is then used to assemble the object files. The Ada parts of the Emulator are compiled with the GNU compiler.

One of the main conclusions reached by this activity is the confirmation of the portability of the ESOC Emulator, as well as the convenience of its design. The Ada components will run in any platform supported by the GNU compiler (a language is as portable as its compiler/interpreter) and new EMMA back ends can be added in a relatively simple way. All combined, this provides excellent performance and good portability: a luxury that can be easily underestimated.

At the time of writing this paper we are still assessing the performance improvements achieved by the new backend. Preliminary measures show promising results.

HANDLING OF MEMORY-MAPPED INPUT/OUTPUT

A known performance issue in the operation of spacecraft operational simulators is the management of the memory mapped Input/Output, or MMIO. The emulator shell provides services to mark blocks in the address space of the emulated process as MMIO served memory locations. Whenever a memory location pertaining to those areas is accessed by a load or store operation, the code generated by a user-provided tailoring procedure is executed and the emulation continues.

The management of a MMIO read or write operation usually requires two steps:

- **The decoding of the requested address** to determine the I/O device that shall attend the I/O operation. The user-provided tailoring procedure is given the memory address and the size of the data to load or to store.
- **The modelling of the I/O device.** As soon as we identify which I/O device shall process the access to the MMIO address, the proper model is invoked to emulate the operation.

Some of the standard I/O devices that can be found in an ERC32 board are modelled by the Emulator core. The user tailoring can invoke those internal models after the decoding is done. On the contrary, spacecraft-specific I/O devices are usually modelled out of the emulator core, in some SMP2 model. That means that the user-provided tailoring shall call those external models by using a call back. Unfortunately, the classic way to manage those situations in Operational Simulators involves the usage of a single call back mechanism that handles the requests for all the I/O devices. The situation is depicted in the Figure 5.

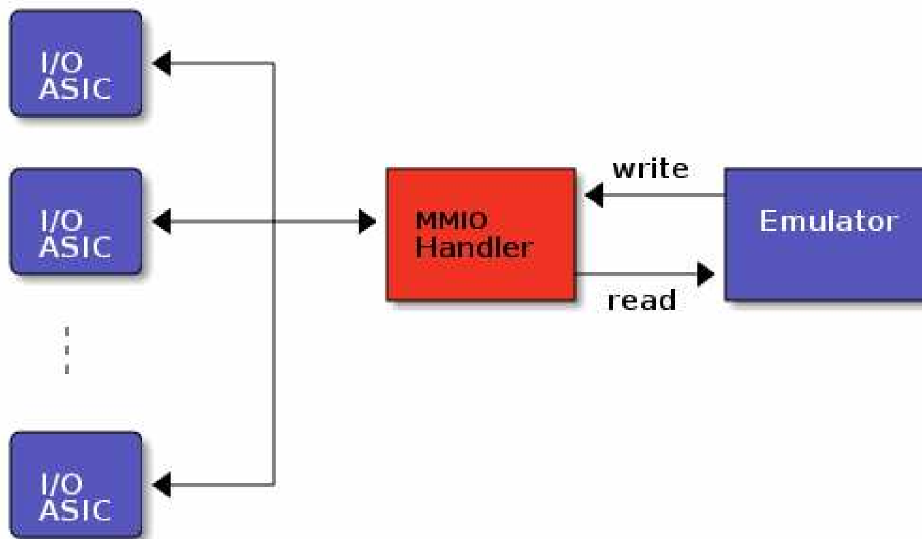

Fig. 5 Emulator interacting with several I/O ASIC models.

The usage of a single call back to manage the MMIO requests for the externally modelled I/O devices has several problems:

- **The frequency of read/write operations** directed to externally modelled MMIO devices is in the order of thousands of accesses per second. That means that the efficiency of the MMIO handler is critical and can easily become a bottleneck in the overall performance of the emulator, and thus of the simulation.
- **The decoding of the I/O address is performed twice.** Since the user-tailored procedure implementing the MMIO calls has to decide whether to call to an internal IO model or to invoke the single call back for the external devices, some of the decoding of the address is always performed in the emulator core. That is usually done using efficient assembly operations such as the application of logic masks and bit rotations. But then the external handler has to decode the external I/O device. That second-stage decoding is usually done with inefficient C++ data structures from the standard templates library such as lists and vectors.

It seems that the usage of a single call back mechanism is the consequence of a long standing requirement in Operational simulators for ESOC: compatibility with the TSIM ERC32 emulator. If that requirement is not present more efficient solutions can be put in place by using the customization facilities provided by the ESOC emulator. Fortunately, recent operational simulators introduced some interesting performance improvements in this area.

SWARM tries to avoid the usage of the single call back mechanism for one of its mostly accessed models by directly accessing to the memory managed by the Link Memory model, for both reading and writing operations. A pointer to the buffer managed by the I/O model is passed in the emulator arguments. The same optimization could be applied for the writing operation in the Exchange Memory. For other devices the single call back is still used.

Herschel-Planck makes use of an extremely nice Ada written model for the COCOS device. COCOS stands for "Computer Core Support" and is a sort of multi-function hub that acts as an interface between the ERC32 processor and the several buses and devices[3]. Most of those devices are mapping registers and memory areas in the address space of the processor. The COCOS model used in Herschel-Planck, along with the tailoring of the emulator, provides a direct access to a memory buffer for simple register read and write operations. It still resorts to an external MMIO handler in complex read and write operations where a side effect has to be simulated by the model. The MMIO tailoring of the emulator used in Herschel-Planck can be configured to not use the direct access to the memory buffer via an emulator shell service and a test harness command. This is quite useful when debugging.

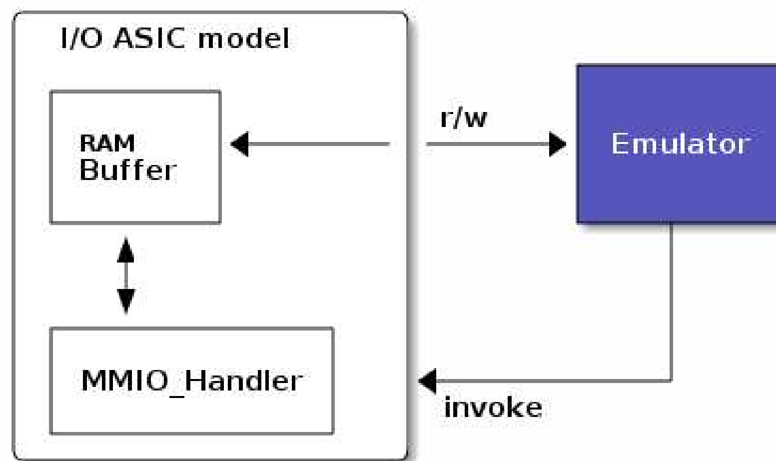

Fig. 6 External I/O device and the Emulator.

The conclusions we derive from those use cases are:

- **Direct access to memory buffers** provided by the external I/O devices shall be used whenever possible. The user shall be able to force the usage of call backs for debugging purposes.
- **The usage of a single MMIO handler routine shall be avoided.** Instead, every external I/O device shall provide a MMIO handler. In that way we avoid effort duplication, since all the decoding could be performed in the tailored emulator core. See Figure 6.
- **Moving some of the I/O models into the emulator core** is an interesting possibility for devices usually used in spacecrafts.

REFERENCES

[1] A. Dartnell. "Herschel-Planck Simulator Software Budget Report".
[2] A. Dartnell. "Extensible Meta-Macro Assembler (EMMA) User's Manual". Issue 2 Revision 0. 24 October 2001.
[3] Catovic, Edvin, Hedberg, Daniel. "High Level Description of an ASIC Implementing CPU Support and I/O Control". Chalmers University of Technology. Goteborg 2002.