LeonVM: Using Dynamic Translation for Developing High-Speed Space Processor Emulators

Paulo Marques[(1)], José Feiteirinha[(1)], Luís Pureza[(1)], Niklas Lindman[(2)], Mauro Pecchioli[(2)]

[(1)] CISUC, University of Coimbra
Dep. Eng. Informática, Pólo II – Universidade de Coimbra,
3030-290 Coimbra, Portugal
{pmarques, feiteira, pureza}@dei.uc.pt

[(2)] ESOC, European Space Agency
Robert-Bosch-Str. 5
64293 Darmstadt, Germany
{Niklas.Lindman, Mauro.Pecchioli}@esa.int

1 INTRODUCTION

A critical part of most space simulations is the actual emulation of the onboard processors of the spacecraft and of the onboard software being run on those processors. In the past, most of ESA missions used the MIL-1750 family of processors [1]. ESA is currently using the ERC32 processor [2,3,4] for most missions and upcoming missions. Nevertheless, in the long run, it is expected that the LEON family of processors will be used. LEON is a low cost SPARCv8-compatible processor which was initially developed by ESTEC. LEON is freely available as a synthesizable VHDL model that can be easily implemented on an FPGA board or as an ASIC. Currently, both Atmel and Saab-Ericsson Space have readily available implementations of LEON2. Almel's is commercialized under the designation AT697E, running at 100 MHz and offering a performance of 86 Dhrystone MIPS for space applications [5]; Saab-Ericsson's is named COLE and runs at 128 MHz having a reported performance of 90 MIPS [6].

One currently outstanding problem is that there is no available LEON2 emulator capable of achieving 90 MIPS running on existing machines. Even the emulation of an ERC32 running at 14 MHz, which is a simpler processor, achieves less than 30 MIPS using ESOC's family of emulators. This presents a problem because it is largely insufficient for doing real time simulation campaigns, which are critical for training spacecraft operators for upcoming missions. Even in the case of ERC32, the safety performance margin for doing these simulations is currently low.

The University of Coimbra, in collaboration with the European Space Operations Center/ESA, is currently researching techniques for doing fast emulation of space processors, especially in the context of spacecraft operator training. Typically, processor emulation is done by *interpretation*. Interpretation means that the emulator runs in a cycle where is *fetches* an instruction, *decodes* it and then *executes* it, starting all over again. We are currently investigating a different technique, called *dynamic code translation.* For that, we are prototyping a LEON2 emulator called LeonVM. A dynamic translator fetches a *basic block* of instructions, translates the whole block into native code, and then executes that native code. Further optimizations include caching previously translated blocks and combining heavily used blocks into single ones.

The results from our first prototype, which has been implemented over the last year, shows interesting performance figures. We are able to emulate the LEON2 with a performance of 146 Dhrystone MIPS on an Intel Core2 Duo E6600 machine running at 2.4GHz, and 102 MIPS on an AMD Athlon64 X2 4600+, also at 2.4GHz. These are relatively cheap off-the-shelf PCs, commonly found in the consumer market.

In this paper we present the architecture of the LeonVM emulator, the used techniques and performance so far attained. We believe that by using such techniques ESA can soon expect to have extremely performant emulators (e.g. for LEON2), especially for the spacecraft operator training domain, further enhancing the current software simulator infrastructure.

The rest of this paper is organized as follows: Section 2 presents an introduction to different processor emulation techniques; Section 3 presents the architecture of the LeonVM emulator, its performance and applicability; Section 4 presents the conclusions.

2 PROCESSOR EMULATION

In this section we present a brief introduction to machine and processor emulation. This subject is discussed in depth in "*Virtual Machines: Versatile Platforms for Systems and Processes*" [7], an essential reference on the subject.

2.1 Introduction

An *emulator* or *virtual machine* is a piece of software that allows a user to execute programs that were not compiled for its machine architecture. The two most well-known successes in this area was DEC's FX!32 emulation engine [8] and the current Java Virtual Machine (JVM) [9]. While the first allowed the owners of DEC Alpha workstations to transparently execute Intel x86 binaries on their computers, having appropriate performance; the second allows *bytecode* programs to execute on a variety of target platforms (e.g. x86 machines, SPARCs, PowerPCs, etc.). Note that in the second case, what happens is that programs that are written in Java are compiled for an abstract machine called Java Virtual Machine (JVM). The JVM is a 32-bit computer which has its particular ISA (a well-defined set of instructions), a certain memory model, standardized input/output devices and so on. In fact, this machine could and has been implemented in hardware. Nevertheless, nowadays, in most cases, people use a software implementation of this abstract machine, i.e. an emulator or virtual machine. For Java, many implementations exist, being the most notorious the ones from Sun [10] and IBM [11].

Fig. 1 shows the relationship between the several components of an emulation system. There is a target host platform where the user wants to run a certain executable of another platform. On top of the host platform sits a particular operating system that houses the emulator. The emulator provides a way of loading executables to be run. While the machine were the executable is going to be run is called *target host* or *target platform*, the original executable is called *guest executable*.

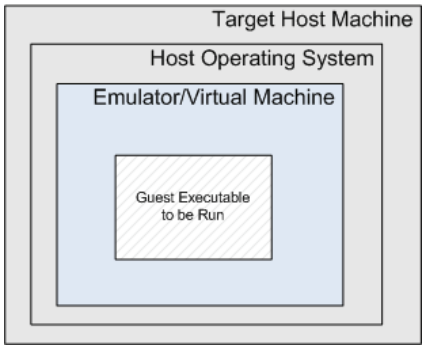

Fig. 1 – Relationship between the several components of an emulator system

When discussing emulation of target platforms, there are always two levels of compatibility that must be considered:

- Emulation of the ISA (Instruction Set Architecture) and existing peripherals.
- Conformance to one or more target ABIs (Application Binary Interface).

In the first place, an emulator can implement the full instruction set and environment of a target machine. This many include not only mimic the ISA but also all the peripheral devices and the way they are accessed. If an emulator goes to this extent, in theory, it will be able to execute any application that runs on the platform being emulated.

Nonetheless, in many cases, it is not wise to go to the full extent of emulating a complete guest machine. If an emulator supports the target platform instruction set it may be able to run existing programs without emulating the complete system. In many cases is possible to execute the instructions of a guest program "normally" and when certain low-level calls are necessary (e.g. for outputting a string to the standard output device), the emulator can simulate at high-level the target device and use its own hosting operating system to accomplish the task. This approach normally makes the emulation of programs much faster than just simulating the complete low-level details of the original system. For clarity, let's suppose that is necessary to emulate a disk access. It is much easier and faster to use the host Operating System (OS) file system services that emulate all the low-level details of a physical disk. This also applies to emulating high-level functionality, like an `open()` system call.

What makes the second case possible is that whenever a machine is developed, the manufacture publishes a specification called ABI (Application Binary Interface). The ABI specifies the contract between an executable and its surrounding environment. For instance, an ABI will say what is the binary format of an executable, what instructions it can contain and what are the standard services that the program can expect to be available (i.e. which system calls). It

also specifies things like the size of data types, how parameters are passed between function calls, stack organization and so on.

Thus, by having an emulator that conforms to a particular ABI and uses its operating system underlying resources, it is possible to run executables of target platforms without having to implement the whole architecture of a machine. This was the approach that was taken by the FX!32 emulation engine [8], by HP's PA-RISC to IA-64 engine [12], and the current Intel IA-32EL [13] engine which allows x86 applications to run on Itanium2 machines. It is also the approach implemented in current Java virtual machines, though in this case there is no clear separation between the machine architecture and the ABI – it is all one specification [9]. Normally, when a new compiler is written, one of the critical steps is incorporating the ABI specification of the target platform into the back-end of the compiler.

This means that, when we consider the emulator architecture presented in Fig. 1, either the emulator offers a way of loading executables that are ABI-compliant with the original platform or it simulates the complete system architecture, including boot-up sequence and physical peripherals. The first case is mostly used when the requirements ask for emulating the original platform running a certain operating system (e.g. running Apple's PowerPC MacOS-X executables in an Intel 4 machine with WindowsXP). The second case is most commonly used when developing embedded and real-time systems. Concerning the emulation of the LEON2 processor, it clearly follows under the second category. Software running LEON2 is typically seen as a whole binary image running either from an EPROM or downloaded through a serial line (UART). Although a real-time operating system can be used, LEON2 is in line with embedded software not with the generic OS emulation approach.

Over the years, the technologies used to allow programs to be executed on different machine architectures have been steadily evolving. Nowadays, two of the most common approaches are:

- Using a software interpreter
- Using a dynamic code translator

The next sections present them briefly.

2.2 Software Interpretation

Having a software interpreter is the oldest approach for cross-platform program execution. Basically, an interpreter is a monitor program that executes in a loop. On each iteration, it fetches a new instruction from the program, decodes it, and then executes it. The interpreter also maintains data structures which represent the memory image of the program and simulated internal state of the processor. This internal state includes the Program Counter (PC), the General Purpose Registers (GPR) and any special purpose registers. The Program Counter is updated on the execution of each instruction. Fig. 2 shows the architecture of such an emulator.

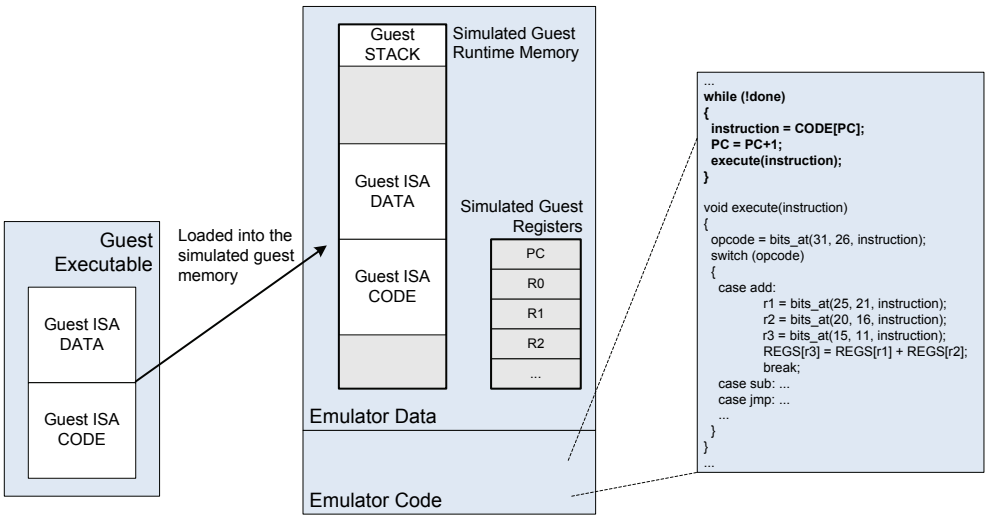

Fig. 2 – Architecture of a simple emulator

The biggest advantage of using an interpreter is that it is relatively simple to implement and is able to maintain a cycle-true simulation of the original platform. The biggest disadvantage is that it is immensely slow.

One of the main reasons why interpretation is so slow is that it requires too much work for each emulated instruction. The overhead *per* emulated instruction is too high. For emulating *each* instruction, is necessary to load registers with the appropriate values, perform the operation corresponding to the instruction, save the computed values, existing condition codes, and update the simulated program counter. In many cases this is done by calling different routines inside de emulator, which further slows things down. It is interesting to note that researchers (e.g. in [14]) report that context switches between SPARCv8 emulated instructions can, when optimized, be "as low as" 22 instructions when running on a SPARC host, and 10 instructions when running on an x86 machine. These values refer only to the overhead of saving and restoring context between emulated instructions. In [15], Cmelik reports an average of 10 SPARC instructions for each emulated MIPS instruction.

Thus, every emulated instruction implies executing many native machine instructions. This corresponds to the fetching of the instruction, opcode decoding and finally execution of the instruction. And while in the native guest platform many of the instructions actually are direct operations in registers, which is extremely fast, in the case of an interpreter, many of the replaced instructions require accesses to main memory, which is, at least, an order of magnitude slower.

But even worse than executing several instructions for each emulated instruction, there is the problem of "branching". The problem is that modern processors are heavily pipelined (e.g. Pentium-4 has a 20-stage pipeline). A failed branch represents an immense cost in terms of performance and throughput of instructions. The performance impact can be huge. Looking back at the example of Fig. 2, it is easy to see that for being able to execute each emulated instruction there are at least the following branches: the main loop branch, the `execute()` branch, the `switch()` branch and the three corresponding returns. Moreover, because the branches are correlated with the execution trace of the original guest program, this implies that in each iteration of the main loop, the interpreter typically branches to different locations. Thus, branch target predictions and similar mechanisms in modern processors are of little help. In fact, because of this architecture, the emulator program is essentially stalling the pipeline, almost making it execute one instruction at a time.

To summarize, the main reasons why interpretation is so slow are:

- For each instruction to emulate, it is necessary to execute a potentially large number of native instructions. (Potentially serious, but probably not the main cause of problems.)
- For each instruction to emulate, it is necessary to perform various memory accesses, especially if registers are mapped in main memory. (Serious and a great cause of performance loss.)
- For each instruction to emulate, it is necessary to perform a multitude of branches. Each branch seriously stalls the pipeline. Internal mechanisms of the host processor are of little help because jumps occur to many different places and, in general, in an unpredictable way. (Quite serious and a huge cause of performance loss; jumps should be avoided at all cost.)

2.3 Dynamic Code Translation

The idea behind a dynamic code translator is simple: instead of fetching, decoding and execution one instruction at a time, this can be done in blocks. Thus, a dynamic translator fetches a basic block of instructions[1], natively translates those instructions into binary code of the underlying platform, and they executes the result at full speed. The next figure illustrates the approach.

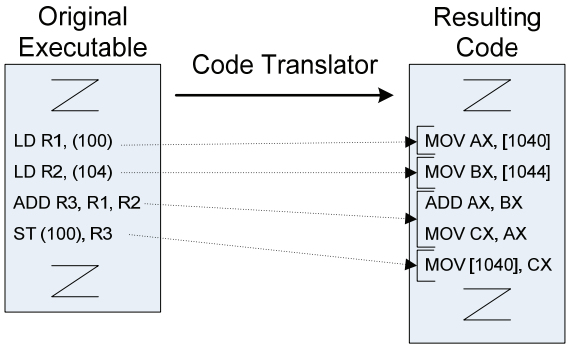

Fig. 3 – Native code translation

[1] A *basic block* is a straight line of instructions with no branches. This means that a basic block has only one entry point and one exit point.

The performance improvement can be huge:

- **The fetch-decode-execute loop is removed.** Instead of at runtime fetching and decoding single instructions, the main loop of the translator only has to be invoked between basic blocks. This means less branching and an improved usage of the underlying machine's pipeline.
- **Previously translated blocks can be cached.** If a certain block of code has already been translated, it can be saved for latter use without having to incur in the penalty of retranslating it again. Since programs exhibit locality of reference, this is the normal case and implies large performance gains.

It should be noted that the translation process occurs at runtime, *Just-In-Time* (JIT). Advanced translators also combine several previously translated blocks into one, further removing branches between them, leading to improved performance.

Fig. 4 shows the appearance of a virtual machine containing a JIT dynamic translator. Note that as program segments are translated, they are cached and put in a "staging area" memory. The virtual machine has to maintain a mapping between what program segments have already been translated and which ones have not.

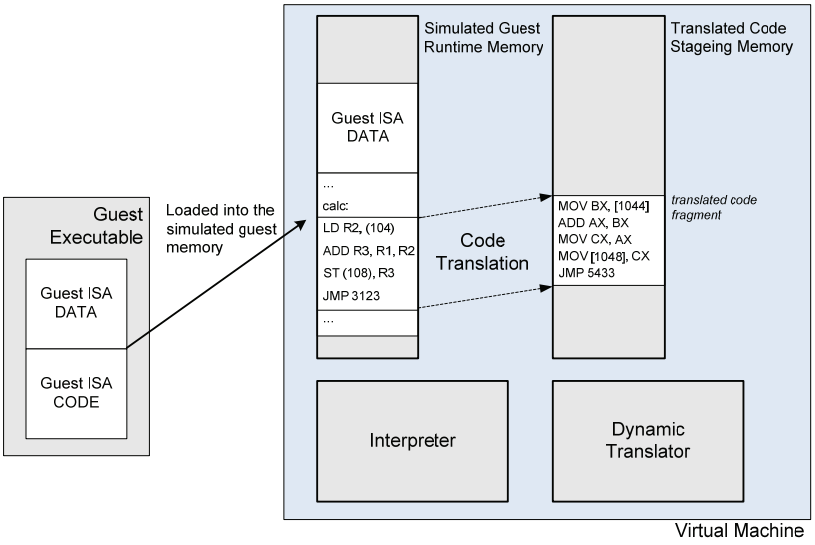

Fig. 4 – Appearance of a virtual machine with a just-in-time dynamic translator

One interesting point is that for some programs it might be possible to statically translate the entire executable before even actually executing it. Although in some cases it is so, for many programs, that is not possible. The main reasons preventing the use of direct static translators are:

- **Indirect jumps using register or memory contents** (i.e. the target locations for the jumps are only available at runtime). This problem is easily solved in a dynamic translator: the virtual machine only has to maintain an internal table mapping which pieces of code have already been translated. Thus, if an indirect jump occurs to a certain address and the target address has yet to be translated then the just-in-time translator is invoked.
- **Self-modifying code and self-generating code** (i.e. the program generates or modifies its code at runtime). Being a problem for static translators, in the case of dynamic translators, the problem is solved in the same way as before. If a program generates code at the execution time, it has to jump to the address of that code. Thus, it is an indirect jump. If the code modifies itself, it is slightly more complicated. In this case, the virtual machine has to keep track of where instructions are writing to. If they write to their code segment, then the corresponding basic blocks can re-translated or, eventually, marked with an "interpret only" flag.
- **Self-checking code** (i.e. code that computes checksums over itself). Self-checking code is solved in dynamic translators because the original code has to be maintained in memory while the program is executing. Thus, any calculation made over those memory locations is correct.
- **Code that loads dynamic libraries**. Being a problem for static translators, for dynamic translators it is not. The process of loading a dynamic library corresponds to reading a file into memory and performing indirect jumps to the recently loaded functions. Thus, the runtime is able to know that these routines are yet to be translated.

It is a hard truth that a dynamic translator between native ISAs (e.g. SPARCv8 to IA-32) will never be able to achieve the same level of performance than translators from bytecode (e.g. Java) into host platforms. Bytecode has been thought with translation in mind and designed to be fast to translate and execute. Nevertheless, it is still possible to achieve remarkable performances when translating between different low-level ISAs [8,12,13,14,15].

3 THE LEON VIRTUAL MACHINE – LEONVM

3.1 Overview

The LeonVM is a high-performance emulator for the LEON2 processor [16], developed at the University of Coimbra. This emulator is based on dynamic code translation, using the techniques explained in the previous section. Fig. 5 shows the internal architecture of the LEON2 processor family.

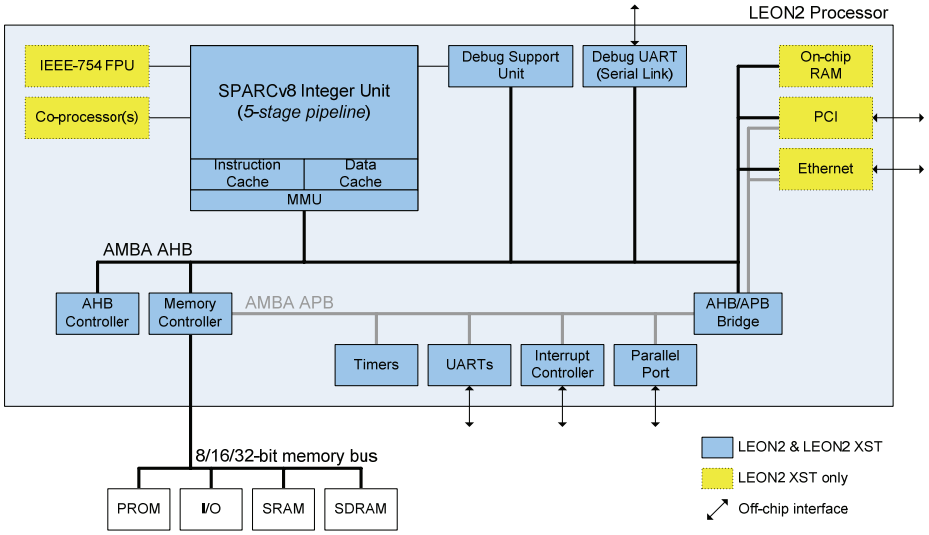

Fig. 5 – The LEON2 internal architecture

Currently, we emulate the following components: the SPARCv8 pipeline; the Memory Management Unit (MMU); an IEEE-754 compatible Floating Point Unit; and all major peripheral devices (Timers, UARTs, Interrupt Controller, Parallel Port). We do not emulate the special devices of LEON2 XST (Ethernet, PCI, on-chip RAM), nor the Debug Support Unit and Debug UART.

The next image shows a screenshot of the LeonVM while running "*SnapGear Embedded Linux*" with a 2.0.39 kernel (MMU disabled).

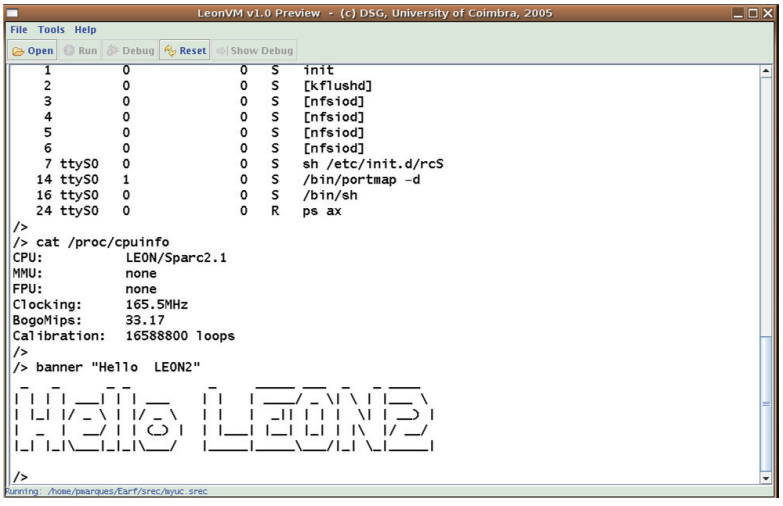

Fig. 6 – LeonVM running a 2.0.39 Linux kernel (MMU disabled)

While developing the emulator, we guarantee correctness by systematically running a set of applications as test harness. Typically, every time a bug is introduced, one of the applications starts giving different results or crashes. Actually, the Linux kernels are particularly susceptible to this, refusing to load, freezing or crashing whenever a bug is introduced. Our test harness currently consists in the following applications: *SnapGear Embedded Linux* (with and without MMU – kernels 2.6.11 and 2.0.39 respectively); *Dhrystone*; *Whetstone*; *Matrix Multiplication*; *N-Queens*; *Memory Test*; *Quick Sort*; *Primes*; *String Comparison*; *Pi Calculation*; *Fibonacci Numbers*; and *Hanoi Towers*.

3.2 Architecture

The next image shows the internal architecture of the LeonVM emulator.

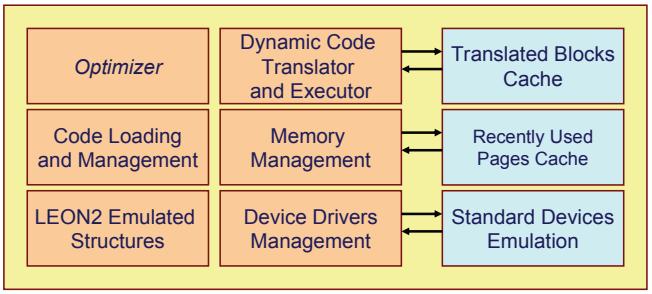

Fig. 7 – The internal modules of the LeonVM emulator

As it can been seen, it has six major modules:

- **LEON2 Emulated Structures**. This module contains all the data structures that must be maintained in order to properly emulate a LEON2 processor. It includes the mapping of the processor registers and condition codes, register windows, processor state and so on.
- **Code Loading and Management**. This module is responsible for loading an S-RECORD program image into memory and for interacting with the memory management module so that code can actually execute.
- **Dynamic Code Translator and Executer**. It corresponds to the most important part of the emulator. It basically consists in a cycle which fetches a basic block, translates it and then executes it. In reality, prior to this operation, it first checks a "previously translated blocks cache" against the current Program Counter. If the block has already been translated, being in cache, it is just executed. If it is not in cache, translation is performed, the block is put in cache, and finally run.
- **Optimizer**. The dynamic code translator only performs direct translation of instructions. The LEON2 registers used by those instructions are emulated as a data structure in main memory. This has performance implications since all operations that are actually performed in registers on the original processor now imply memory accesses on the emulator. The optimizer module detects whenever a basic block is being executed a large number of times. When that happens, it is probable that that block will continue to be heavily used. Thus, is makes sense to generate new code where a register mapping algorithm is run. This algorithm generates code that allocates native processor registers to LEON2 registers, inserting it as a preamble to the basic block. It also generates code that removes the register mapping, serializing the corresponding register values to main memory, inserting it as an epilogue to the basic block. Thus, in the middle, is then possible to employ high-speed versions of the translated instructions that only use physical hardware registers. The optimizer only runs when blocks execute above a certain threshold because register mapping and un-mapping is a time consuming operation, besides enlarging the generated code. It only makes sense to apply on heavily used blocks.
- **Memory Management**. For the execution of a LEON2 program, it is necessary to emulate the memory that the processor can access. This module is responsible for that task – translating LEON's addresses into a memory space that has been allocated in the heap. Also, if the LEON2 processor is being run with the MMU enabled, this module is responsible for simulating all the translation from virtual addresses to physical addresses of LEON, and the management of all data structures associated to the MMU (e.g. TLB, caches, etc.). For optimal performance, this module maintains a recently used pages cache.
- **Device Drivers Management**. In LEON2, devices are mapped in the memory address space. Thus, reads and writes to certain addresses must not go to memory but be diverted to these devices. In the LeonVM, devices are pieces of code that are dynamically loaded and can register a certain address range as its own. Thus, any reads and writes that are performed on these addresses result in a callback function invocation to the device

code. This allows for the implementation of the standard devices (e.g. UARTs) in a modular way. Also, special-purpose and debug devices can be easily included in the emulator. This module also provides support for the device modules to generate interruptions. Finally, it should be noted that certain devices had to be more closely integrated with the emulator than "ordinary devices". The timers are the most important example since they have to be constantly updated. It would not be possible to implement them with enough time granularity using the standard device mechanism.

For the most part, the emulator is written in C. Nevertheless, in some parts, assembly code is used for optimized performance. The most interesting part of the emulator is the Dynamic Code Translator and Executor Module. In reality, we coded all the SparcV8 instructions either in "normal" C or inline assembly inside C functions. Then, using GCC, their object versions are generated (i.e. the corresponding ".o" files). Finally, we have an external program, called DYN_GEN (of *dynamic generator*), which loads the object code files, and along with the core of dynamic code executer, (also in C), once again generates C code. This new file corresponds to the whole "Dynamic Code Translator and Executer" module. The next image illustrates the approach.

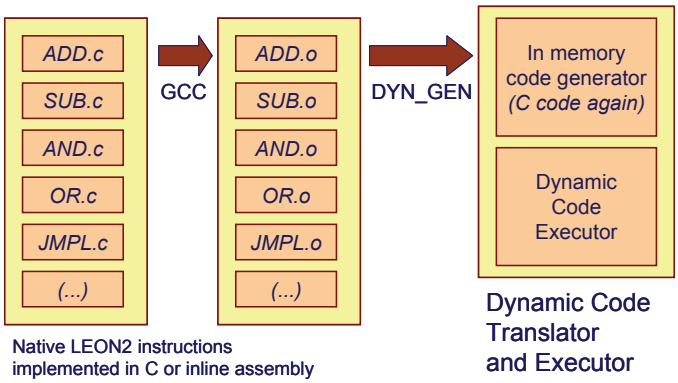

Fig. 8 – How the source code of the Dynamic Code Translator and executer is created

It may seem odd (or even incorrect) that DYN_GEN is using ".o" files and C code, generating C code. What happens is that for dynamically generating code at runtime it is necessary to have "object code templates" in memory. But, the direct C functions (e.g. "ADD") cannot be used since the intent is not to call a function but to concatenate at runtime the contents of several of these functions, representing a basic block. Thus, it is necessary to strip each one of the functions of its preamble and epilogue and save them as "templates". DYN_GEN does such task. Thus, DYN_GEN removes the unnecessary information from all functions and saves them is static arrays of bytes as "C code". This code is then combined with the rest of the dynamic translator code, written in C. This not only allow us to keep the changes of each function under control (e.g. for changing an ADD, it is only necessary to change one file), as it provides an effective mechanism for having a working dynamic translator. Also of importance, each instruction can either be coded in C, for portability and easy testing; or in assembly, for fast execution.

Throughout the project, many optimizations were implemented, which are not possible to discuss in detail here. Nevertheless, the most important optimizations used were:

- Use of "extended basic blocks". We consider a basic block a line of code with no conditional branches. I.e. if there is a branch instruction, as long as it is an unconditional branch, the code at the target location is considered part of the same block.
- Implementing a cache for previously translated blocks. Basically, it means that frequently used blocks are already translated and can be executed at full speed.
- On the implementation of the MMU, using a recently used pages cache. This means that pages that have been used recently are found quickly and memory accesses to these pages are fast.
- Implementing a "virtual register mapping mechanism". In the beginning of each basic block, code is introduced so that some of the machine's physical registers are assigned to LEON2 registers. Thus, all LEON2 instructions in a basic block that only use the assigned registers can actually use optimized versions of these instructions which operate exclusively on registers. At the end of the basic block, the physical registers are written back to memory, to the data structures corresponding to the processor state holding the registers. This allows for extreme performance improvements since operating on registers is two to three orders of magnitude faster than going to memory.

- Only generate optimized code based on profiling statistics: only the mostly used blocks are optimized. This was motivated by the observation that register mapping and un-mapping is a heavy operation. It makes little sense to optimize blocks that are not frequently used, both because of the time it takes to do it (which may be longer than the time it takes to execute them), and because the memory space they will eat up.
- Combining instructions that operated on condition codes for performing conditional branches as single optimized operations. It is quite common in the running code to have an instruction whose only propose it to set some condition code so that a following branch can be conditionally made (e.g. an `ORcc` followed by a `Bicc`). Since setting condition codes is an elaborate task and, for the most part, they are normally only used for conditionally jump, combining these instructions together as one yields good performance improvements.
- Implementing optimized versions of load and store operations. Due to the large number of loads and stores that a program makes, and due to the fact that programs exhibit temporal and spatial locality ("If I have used some data recently it is probable that I will use it again soon; If I have used a certain data, it is probable that I will use data near by."), optimizing those instructions (e.g. by using inline code) becomes critical.

Concerning the last optimization, two assumptions were made that, for the purpose at hand (fast emulation of LEON2 for spacecraft operator training scenarios), we believe are of little consequence. Nevertheless, they are the only points were our emulator departs from the strict implementation of the SPARCv8 ISA:

- We assume that memory accesses are always performed aligned. In the real processor, if a memory access is not aligned, it generates a trap. In our case, the memory access simply occurs. This allows for an overall simplification of the memory access code. We believe that this is of no consequence because for a misalign access to occur in reality it would imply that the compiler had generated incorrect code or that something seriously wrong had occurred in the processor. Thus, the real effect of the trap would be to probably kill the associated process (which is what happens in "normal" running environments), reset the machine or some action alike. In our case, the program simply accesses the memory location.
- We assume that if a basic block is executed a large number of times (currently 10.000), with all its memory accesses to RAM, it will not read or write from peripheral devices. Again, it would be quite strange for a piece of code which cannot contain any conditions, since it is in a basic block, and had executed thousands of times always accessing RAM, to start accessing devices. (It would imply that it had some kind of indirect pointer mostly pointing to RAM and after a very long time it would point to a memory address representing a peripheral device; all this without having any if-clause that would allow redirecting the pointer. That situation would be extremely odd if occurred in reality.)

As discussed in Section 3.1, our emulator currently runs with no observable problems on two Linux kernels, and with 11 different test applications, besides the normal Unix programs directly available on the SnapGear distribution.

3.3 Performance

The current performance of the LeonVM emulator is 146 Dhrystone MIPS on an Intel Core2 Duo E6600 machine running at 2.4GHz, and 102 MIPS on an AMD Athlon64 X2 4600+, also at 2.4GHz. The MIPS performance while running different applications varies, which is to be expected. The performance is shown in the graph bellow.

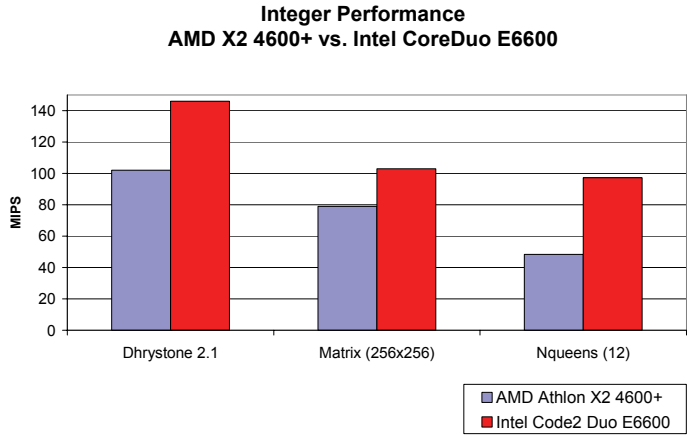

Fig. 9 – MIPS performance while running Dhrystone, Matrix Multiplication and N-Queens

It is also interesting to note that since the beginning of the project, from the days where only a simple emulator was implemented, the performance has been increasing steadily. The next graph shows the evolution over time, concerning only the AMD machine. Note that the result for January was obtained on an AMD Athlon64 3500+ at 2.2 GHz, since the AMD X2 4600+ processor only became available latter on. The results for Intel are not available since the processor is quite new.

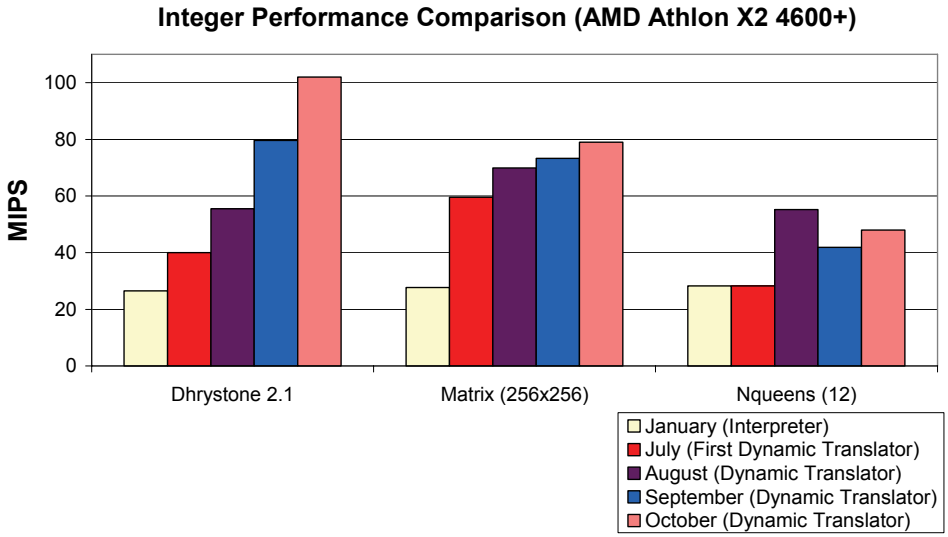

Fig. 10 – Performance evolution over time

In terms of floating point (FP) performance, currently, the LeonVM is weak. During the project no effort was made for improving FP performance since it was out of scope for us. Currently, all floating point operations are emulated in software using the freely available SoftFloat library [17]. The current performance is of 9.5 MFLOPS (Whetstone) on Intel and 8.3 MFLOPS on the AMD machine. The reported performance of Atmel's ASIC implementation is 23 MFLOPS. Nonetheless, it should be noted that the techniques used for implementing the integer unit should be equally applicable for the emulation of the instructions of the floating point unit.

4 CONCLUSIONS

In this paper we have presented the implementation of the LeonVM – a high performance emulator for the LEON2 processor based on dynamic code translation. Dynamic translation is a technique that yields very high performance gains in terms of processor emulation, focusing on fetching basic blocks, translating them natively into the host's instruction set architecture and executing them. When combined with other techniques like the use caches of previously translated blocks, profiling statistics, virtual register mapping and load/store caches, the performance can be quite good. In the context of this project, in less than a year's duration and limited resources, it was possible to implement a LEON2 emulator that runs at approximately 146 Dhrystone MIPS on an off-the-self standard PC.

In terms of difficulties, the downside of developing a dynamic translator is its complexity in terms of code and implementation. Creating a normal interpreter is not exactly easy; implementing a dynamic translator is much harder. Actually detecting the existence of bugs is simple by having a sufficiently strong test harness – things simply stop working, freeze or crash. Understanding the origin of the problems, offending instructions, and concrete bugs is a different story. The issue of locating bugs relates to the asynchronicity of the running code. The LEON processor has a number of timers and devices can generate interruptions. Thus, while running a multitasking kernel on top of the emulator, when a problem occurs, it can be extremely difficult to understand the execution trace that generated the problem.

Finally, in terms of applicability, dynamic translation is especially suited when only correctness from the point of view of the instruction set architecture is needed. I.e. the kernel and programs that run on top of the emulator see it as a normal LEON2 processor. It does not allow for cycle-true processor emulation, detailed and coherent timings, and internal (hidden) caches and buses simulations. Even so, it is well adapted for scenarios like spacecraft operator training, software testing and validation, and alike. Needless to say, for scenarios like embedded systems software development were correct timings are necessary, other approaches must be used.

REFERENCES

[1] "*Military Standard – Sixteen Bit Computer Instruction Set*", Ref. MIL-STD-1750A (USAF), United States Air Force, July 1980. http://www.cleanscape.net/stdprod/xtc1750a/resources/research.html

[2] "*TSC691E Integer Unit, User's Manual for Embedded Real Time 32-bit Computer (ERC32) for Space Applications*", Temic Semiconductors, Rev. G(10/09/96), September 1996.

[3] "*TSC692E Floating Point Unit, User's Manual for Embedded Real Time 32-bit Computer (ERC32) for Space Applications*", Temic Semiconductors, Rev. G(10/09/96), September 1996.

[4] "*MEC Rev. A Device Specification*" (also known as "*SPARC RT Memory Control Manual*"), Saab-Ericsson Space AB, Issue 4, April 1997.

[5] "*Red-Hard 32-bit SPARC V8 Embedded, AT697E Datasheet*", Atmel Corporation, Rev. 4226E-AERO-09/06, September 2006. http://www.atmel.com/dyn/resources/prod_documents/doc4226.pdf

[6] "*Panther Processor Board Product Information*", Saab-Ericsson Space, March 2005. http://www.space.se/NR/rdonlyres/3BC8C5BA-9003-4CE1-92E4-7304462C974E/1974/panther_processor_board.pdf

[7] Jim Smith and Ravi Nair, "*Virtual Machines: Versatile Platforms for Systems and Processes*", ISBN 1558609105, Morgan Kaufmann, June 2005.

[8] A. Chernoff, M. Herdeg, R. Hookway, et. al., "*FX!32 – A Profile-Directed Binary Translator*", in IEEE Micro, Vol. 18(2), IEEE Press, March/April 1998.

[9] T. Lindholm and F. Yellin, "*The Java Virtual Machine Specification*", 2nd Ed., Addison-Wesley Pub. Co., ISBN 0201432943, April 1999.

[10] "*Java Runtime Environment for the Java2 Platform*". http://www.javasoft.com/j2se

[11] "*IBM Developer Kits for Java*". http://www-106.ibm.com/developerworks/java/jdk/index.html

[12] C. Zheng and C. Thompson, "*PA-RISC to IA-64: Transparent Execution, No Recompilation*", in IEEE Computer, Vol. 33(3), IEEE Press, March 2000.

[13] L. Baraz, T. Devor, O. Etzion, *et. al.*, "*IA-32 Execution Layer: a Two Phase Dynamic Translator Designed to Support IA-32 Applications on Itanium-based Systems*", in Proc. 36th IEEE/ACM International Symposium on Microarchitecture (MICRO-36), IEEE Press, San Diego, USA, December 2003.

[14] K. Scott, N. Kumart, S. Velusamy, et. al., "*Retargetable and Reconfigurable Software Dynamic Translation*", in Proc. of the International Symposium on Code Generation and Optimization (GCO'03), IEEE Press, San Francisco, USA, March 2003.

[15] B. Cmelik and D. Keppel, "*Shade: A Fast Instruction-Set Simulator for Execution Profiling*", in Proc. of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, ACM Press, Nashville, USA, 1994.

[16] "*LEON2 Processor Manual, XST Edition, version 1.0.22*", Gaisler Research, May 2004.

[17] "*SoftFloat Library Homepage*". http://www.jhauser.us/arithmetic/SoftFloat.html