JIT Technology for SimLEON
Numerical Emulation of LEON2/LEON3 processors

Olivier MEURANT

Astrium Satellites
31 rue des Cosmonautes
Z.I. du Palays
31402 Toulouse Cedex 4
olivier.meurant@astrium.eads.net

INTRODUCTION

SimLEON is a LEON2/LEON3 processor emulator developed by Astrium Satellites as a core component of full software simulation facilities. It is used operationally on most in-house spacecraft projects as well as in some applications in launchers and aircraft fields.

LEON processors implement SPARC-V8 with cache and 7-stage pipeline designed to be clocked up to 200MHz, delivering computing power much higher than previous generations. Thus software simulation of such processors is very demanding to be able to meet real-time simulation (with hardware in the loop) or faster (full software simulation).

This paper presents JIT technology, successfully implemented into SimLEON and providing a breakthrough in emulation performance.

JIT TECHNOLOGY

JIT technology is based on two principles: cache and regroup operations.

A classical emulator is based on a loop: (See Fig 1)
1. Fetch: read next instruction to be executed
2. Decode: detect instruction type and parameters
3. Execute: execute instruction
4. Timing: compute time in cycle taken by the instruction

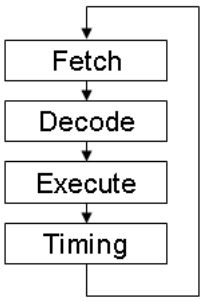

Fig. 1. Classical emulation loop

Emulated flight software is approximately constant: same set of operations is always performed for fetching and decoding. The first principle is then to cache operations in native code (PC-x86) for fetch and decode steps. Fig. 2 presents the new architecture of SimLEON.

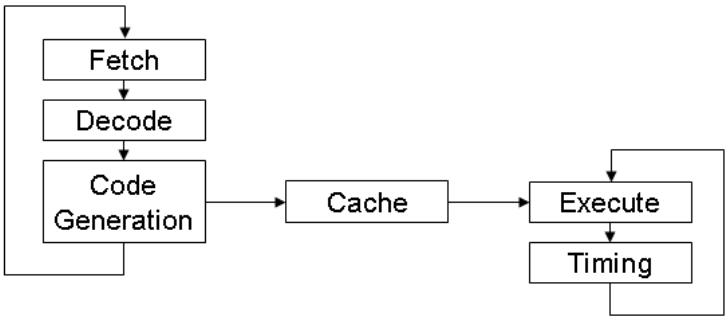

Fig. 2. Native code is stored in cache to be used later during execution

Flight software is composed of sparc assembly instruction. Sparc features a RISC architecture; each instruction encodes a very simple operation. A high level operation like a load from the memory is split between several instructions. Fig. 3 presents a load operation at address 0x80000304 (int value = (volatile unsigned int *)0x80000304;) :

1. "sethi" prepares the high part of the %l1 register.
2. "or" operation sets the low part of %l1, after this instruction %l1 register contains address 0x80000304.
3. "ld" operation reads memory at address specified in %l1 into register %l3.

Second step presents the execution of these 3 instructions. Third step presents the result of simplification of redundant operations.

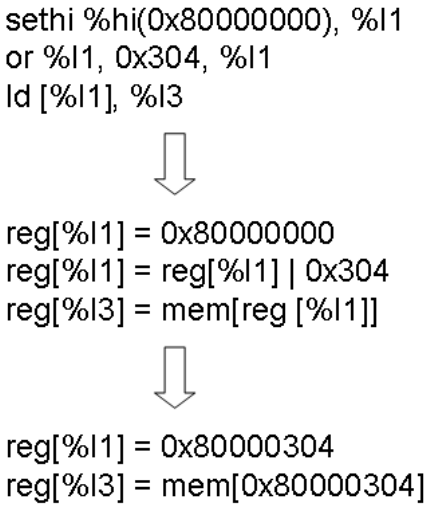

Fig. 3. A typical load operation in sparc assembly:
When grouping instructions, simplification appears on value and on operation.

Grouping assembly instructions into block of instructions leads to efficient simplification on value and on operation, reducing the needed computation power for emulation.

COMPILER TECHNOLOGY

Native code production requires a compiler technology. Four technologies have been assessed:
1. Hand coded: not easy to maintain, hard to optimize.
2. GNU-Lightning: quick code generation but no optimizer [1]
3. GNU-GCC compiler: Widely supported, good quality of generated code but monolithic design [2]
4. LLVM: open-source library, efficient, flexible and modular [3]

LLVM stands for Low Level Virtual Machine; it's a framework, a collection of c++ classes to build compiler-like tools. LLVM uses an open-source license (BSD), is multi-platform (linux, windows, macos), multi-cpu (x86, ARM...) and benefits from open-source community and other companies work (Apple, Google, Microsoft...)

LLVM IN SIMLEON: THE PIPELINE

LLVM defines an assembly language called LLVM-IR [4]. LLVM-IR aims to be a universal low-level intermediate representation; an example featuring a simple addition function can be seen on Fig. 4.
This language is in SSA form (Static Single Assignment) which leads to simpler operations on variables and thus easier optimisations.

```
define i32 @add(i32 %a, i32 %b) {
entry:
  %0 = add nsw i32 %b, %a
  ret i32 %0
}
```

Fig. 3. Addition in LLVM-IR

LLVM-IR is the entry point of SimLEON native code generation pipeline based on LLVM (see Fig 5). The pipeline is roughly split in 3 operations:
1. IR Optimisations: Optimize LLVM-IR with help of various optimisation passes (Constant propagation, instructions combining, simplifying control flow, GVN, DCE...), SimLEON provides its own custom passes to optimize specific parts (mainly on memory layout).
2. CodeGen and target specific optimisations: Transforms LLVM-IR in MC (Machine Code) intermediate representation which is tied to the target assembly language. This step provides "raw" native code.
3. Native code preparation: prepares the "raw" native code. It's mainly a linker task (symbol resolution, fix-up and relocations).

Fig. 5. Native code generation pipeline

SimLEON provides a configurable number of threads which act as pipeline and are concurrently able to generate native code.

FEEDING THE PIPELINE

SimLEON build system provides LLVM-IR for each instructions of sparc-v8 core including timing computation of the instruction. A code walker component is responsible for cutting flight software binary in blocks of instructions.
Fig. 6 presents a pseudo-listing of instructions; the code walker creates 4 blocks:

1. Block 1 begins at 0x00 and ends on branch instruction 0x08.
2. Block 2 is between 0x0c and 0x10
3. Block 3 between 0x14 and 0x20
4. Block 4 between 0x1c and 0x20.

Relationships between blocks are presented on Fig. 7.

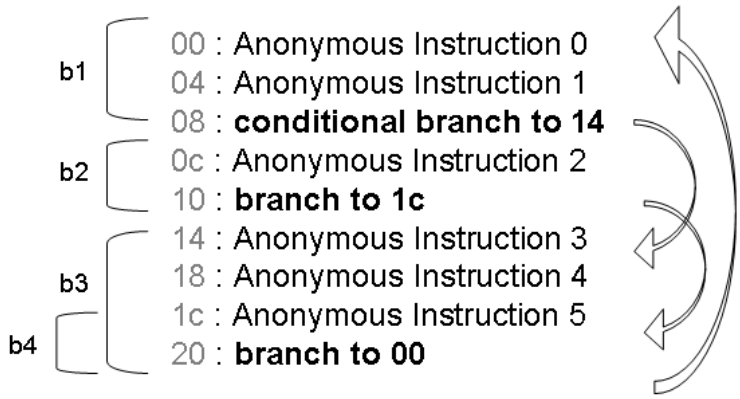

Fig. 6. Code walker example with 4 blocks

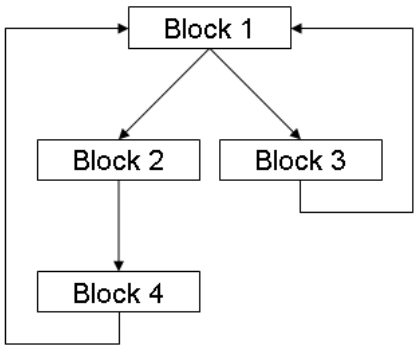

Fig. 7. Relationships between block of the example

LLVM-IR for each instructions of the block is glued together to produce a block of LLVM-IR. This block feeds one of the pipeline thread and production of native code allow the execution of operations needed to emulate the original block of sparc instructions.

EXECUTION

Execution loop of the emulator is modified to take advantage of the potential availability of an executable buffer, optimised for a specific block. Figure 8 presents the basic setup.

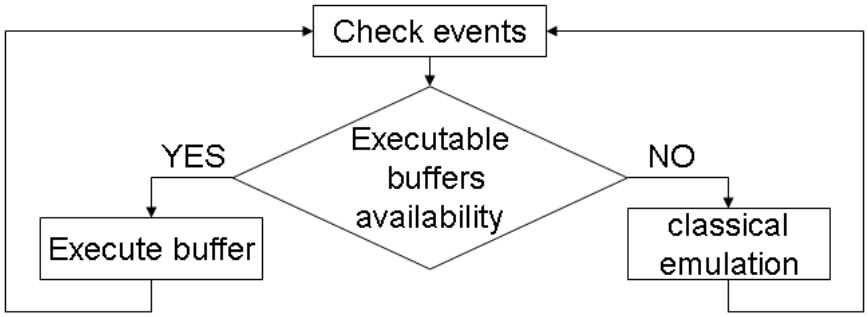

Fig. 8. Execution with optimised buffers

An optional mechanism is provided for the precision of the events. If an event is due to trig inside the execution of a block, the execution uses the classical emulation to guarantee the best precision available on event. Figure 9 explains this mechanism.

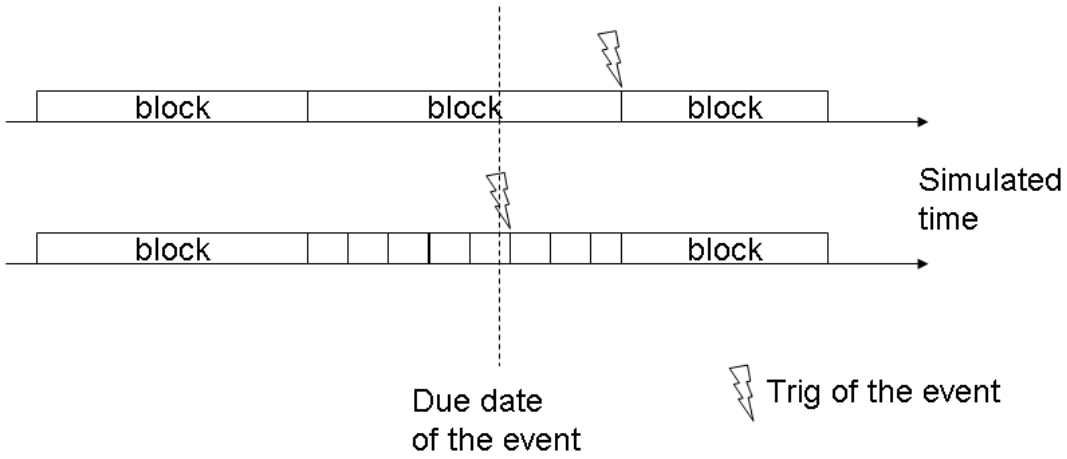

Fig. 9. Upper line presents the situation when the optional mechanism is off.
The trig of the event must wait the end of the current block to be trigged.
The lower line presents what happened when the mechanism is on.
A block is not executed; the event trigs with the best available precision.

An additional (and not optional !) mechanism ensures that if an executable buffer of native code exist for a modified area of the memory, the block is automatically discard to ensure correctness of the simulation.

STRATEGIES

Multiple strategies are available using this code generation technology:
- Static strategy: preparation of executable buffers is entirely done before launching simulation. This is suitable for operation simulator.
- Dynamic strategy: preparation of executable buffers is done during the run of the simulation. Selection of blocks to be prepared is based on heuristic: if an instruction is often executed, preparing a block containing this instruction allows a substantial increase of performances
- Mix of static and dynamic.

RESULTS AND CURRENT STATUS

SimLEON development has been completed mid-2010. Execution with a static strategy is fully supported. Precision in instruction timing modelisation (cache, pipeline…) has been verified, as well as the full set of services. (GDB, breakpoint…).

First test runs demonstrate a 4 to 5 times higher execution speed than previous SimLEON. This means a LEON3 clocked at 120 to150MHz can be emulated in real-time, or a LEON3 clocked at 30MHz can be emulated at x4 to x5 real-time.

On-going activities are currently focused on
- Optimisation to increase execution speed up to x10
- Implementation of dynamic strategy of execution

REFERENCES

[1] http://www.gnu.org/software/lightning
[2] http://gcc.gnu.org
[3] http://llvm.org
[4] http://llvm.org/docs/LangRef.html