Emulator Performance Study

Dr. Mattias Holm
Terma
Shuttersveld 9
2316 XG Leiden
Netherlands
maho@terma.com

March 2, 2015

Abstract

This paper describes an internal study into emulator performance, focusing on the *ESOC Emulator Suite 2.0*, *TSIM* and *T-EMU 2.0*. The study was done by applying both theory to estimate emulator performance based on different emulation methods and experiments that where executed to validate the model and to provide data for development of next generation Terma Emulator, T-EMU 2.0.

1 Introduction

Micro-processor emulators are performance sensitive systems that are critical in today's on-board software development, testing and deployment. During development, the use of emulators is critical in order to ensure that the software runs in an environment with the same behaviour as the hardware (e.g. endianess, floating point behaviour, etc). When deploying a spacecraft, an operational simulator is also deployed in primarily in order to train ground personnel on how to control the spacecraft and to test operational procedures before executing them on the real satellite.

The different activities have different demands on performance and accuracy on the emulation. For example, in an operational simulator, humans are in the loop, so high performance (faster than real-time) is important, while timing accuracy can be relaxed. For software validation higher accuracy is often needed, while at the same time, performance can be relaxed (within reason), at least in a fully virtual environment.

In order to get a better understanding of the performance of emulators, an internal study was carried out. This served two purposes, firstly to be able to predict achievable performance of an emulator and secondly to have numbers for guiding future developments. In the study the following emulators where examined: *T-EMU 2.0*[3], *TSIM*[2] and the *ESOC Emulator Suite 2.0*[1].

Emulator performance can be measured in two primary ways. The direct approach is to measure the number of emulated instructions executed per second for some test application. This figure is known as *Instructions Per Second* (IPS). The *IPS* number is usually due to the volume of instructions expressed in *MIPS*[1] although other denominations are common as well (eg. *kIPS* and *GIPS*). The *IPS* measurement is depending on the efficiency of the emulator and the host machine running the emulator.

The second way to measure emulator performance is to compare the progress of *simulated real-time* (SRT) to *wall-clock time* (WCT). When these numbers are compared, we get either $S = t_{srt}/t_{wct}$ expressing speedup in *times real-time* (TRT) or the reciprocal slowdown (times slower than real-time). The S value depends on the efficiency of the emulator, the host machine and the clock frequency of the emulated target processor.

Measurements in *MIPS* are useful for comparing raw emulator performance between different emulators (e.g. TSIM vs ESOC Emulator or different emulator versions). The second measurement S is useful for comparing actual performance of an emulated system in real-world scenarios and takes into account the effect of optimisations such as idle loop detection.

The remainder of this paper goes into the detail of the methodology used for the performance study and the results achieved during the study.

[1] Not to be confused with Dhrystone MIPS which is a measurement of the number of iterations through the Dhrystone program per second dividied by 1757.

2 Methodology

The performance study was carried out in two phases. First a simplified theoretical model of an emulator was developed in order to get an understanding of what performance can be expected from an emulator. This was followed by an experimental study, which firstly attempted to validate the theoretical study, and secondly to quantify emulator performance in both synthetic benchmarks and real-world scenarios.

The model consisted of couple of idealised instructions including mandatory overhead from these when applying different emulation methods. The target instructions where broken down into numbers of implementing host instructions. In addition to this a mapping of the ESOC Emulator Suite to this model was done. The experiments on the other hand focused on executing firstly integer benchmarks primarily to validate the model. And secondly the integration of the ESOC Emulator Suite 2.0 in a Software Validation Facility (SVF) already running TSIM and comparing the real-time performance to each other.

2.1 Synthetic Benchmarks

In order understand the raw performance of especially the ESOC Emulator in different configurations, and to analyse the design and evolution of T-EMU 2.0, Two different types of benchmarks were executed, firstly single instructions where bench marked to validate the model, secondly the Dhrystone benchmark was executed, while measuring the emulated MIPS (not to be confused with the Dhrystone MIPS figure).

2.2 Software Validation Facility Integration

In order to understand the behaviour of the ESOC Emulator in particular, it was integrated in a LEON3 based SVF, using the TSIM API which was implemented as a wrapper on top of the ESOC Emulator APIs. In the SVF, speedup of the emulated target was measured for different tests. Firstly, the boot-performance was measured (i.e. the time from power on until the application software has been loaded and starts executing) and secondly the timing for TC/TM responses was measured.

3 Model

This section describes a simplified version of the idealised emulator model. There are different types of interpreted emulation methods, including *decode-dispatch* and *threading*. The different emulation methods described here can be found described in more detail in [4].

The models estimate the instruction count of the different emulation methods including the overhead from the emulator loop (whether threaded or not). A 3.5 GHz PC is used as reference with the assumption that it has a *Cycles Per Instruction* (CPI) of one.

3.1 Decode Dispatch Loop

An interpreted decode-dispatch emulator consists of a main emulator loop, which executes a loop similar to the following:

Listing 1: Decode Dispatch Loop

```
while (!stopEmuLoop) {
  instr = mem[pc];
  instrFunc = decode(instr);
  instrFunc(instr);
}
```

As can be seen, the main loop fetches an instruction from emulated memory, decodes the instruction and then calls the instruction function (this could also be done with a computed goto).

The decoding can be done in different ways, one way is to do this using a multilevel table matching the target processor *Instruction Set Architecture* (ISA). Another way is a multi level switch statement and a third is to try to compress as much as possible in a single table. For systems not using the single table approach (which may not be practical due to the ISA), speed can be improved using a decode cache. A good estimate is that the decoding takes up 10-20 instructions.

An add instruction can be implemented as follows:

Listing 2: Decode Dispatch Instruction

```
void
add(uint32_t inst)
{
  int RA = extractField(inst, raStart, raBits);
  int RB = extractField(inst, rbStart, rbBits);
  int RC = extractField(inst, rcStart, rcBits);
  reg[RA] = reg[RB] + reg[RC];
  PC = PC + 4;
  TIME = TIME + ADD_CYCLES;
}
```

Here the *extract field* function effectively results in a shift and mask operation, while the updates of the PC and TIME variables each results in a load, and addition and a store.

3.2 Threading

One approach to improve emulator performance is to apply a technique known as threading. In a threaded interpreter, the main loop is eliminated and threaded through each instruction using computed jumps (this can be done either using computed gotos or with tail-recursive calls). The best way to describe this is actually to illustrate it with an example using computed gotos:

Listing 3: Threaded Emulator

```
add:
  int RA = extractField(inst, raStart, raBits);
  int RB = extractField(inst, rbStart, rbBits);
  int RC = extractField(inst, rcStart, rcBits);
  reg[RA] = reg[RB] + reg[RC];
  PC = PC + 4;
  TIME = TIME + ADD_CYCLES;
  if (stopEmuLoop) goto exitEmuLoop;
  inst = mem[PC];
  instLabel = decode(inst);
  goto *instLabel;
  // ...
```

The key point with the threaded approach is that the return from the instruction to the main interpreter loop is eliminated resulting in the elimination of a branch. The mechanism illustrated is known as *indirect threaded interpretation*.

It is possible to apply additional optimisations to the threaded interpreter models. One such optimisation is to apply a technique known as pre-decoding. The idea is to store already decoded instructions in a cache (or in the memory model directly).

To get this done efficiently, an auxillary shadow program counter indexing the decoded memory data is introduced called *Thread Program Counter* (TPC). When this has been done we get target instructions that are implemented as follows:

Listing 4: Pre-Decoded Threaded Emulator

```
typedef struct {
  unsigned op;
  byte RA;
  byte RB;
  byte RC;
} decoded_instr_t;
// ...
add:
  int RA = decodedMem[TPC].RA;
  int RB = decodedMem[TPC].RB;
  int RC = decodedMem[TPC].RC;
```

```
reg[RA] = reg[RB] + reg[RC];
PC = PC + 4;
TIME = TIME + ADD_CYCLES;
if (stopEmuLoop) goto exitEmuLoop;
next = decodedMem[TPC].op;
instLabel = instTable(next);
goto *instLabel;
// ...
```

With the pre-decoding, decoded instructions will be less dense than the target ISA (resulting in RAM, cache, and memory load costs). On the other hand the decoding that is done in the instruction table is just a lookup. This can in turn be optimised a step further by eliminating the instruction address lookup by storing the instruction address in the decode table instead.

These two pre-decoding methods are known as *indirect pre-decoded threaded* and *direct pre-decoded threaded*.

3.3 Modelled Values

Table 1 illustrates the cost for an add instruction in the emulator in instructions. The overhead is quantified in Table 2. Table 3 illustrates the theoretical MIPS count that an emulator will achieve on a 3.5 GHz PC with a CPI of 1.

Table 1: Instruction Count for Add Instruction

Operation	**Mem Access**	**Ops**	**Total**
extractField	0	2	2
Reg-Reg op	3	1	4
PC update	2	1	3
Time update	2	1	3
Total	7	5	12

Table 2: Instruction Count for Emulator Overhead

Operation	**Instructions**
decode	10
fetch op	1
decode-dispatch extra	1

Table 3: Theoretical MIPS Values of an add instruction on a 3.5 GHz Host

Emulator Type	**MIPS (3.5 GHz)**
Decode-dispatch	114
Indirect threaded	122
Direct pre-decoded threaded	210

3.4 ESOC Emulator Mapping

The ESOC Emulator does not directly fit into the classifications made in Sections 3.1 and 3.2. The ESOC Emulator provides both the capabilities of a decode-dispatch interpreter and a threaded interpreter. It does indirect predecoding, but does not save decoded operands which are instead unpacked as normal. The best way to describe the ESOC Emulator is as *indirect pre-decoded dispatch loop* or *indirect pre-decoded threaded*.

In addition to the classification, the ESOC Emulator (when running on the x86 processor) utilise a Virtual Machine (named VAC) in which the emulated target machine is executed (i.e. a VM in a VM). The maintenance of this VMs state costs instructions and in the end performance. The overhead of this must be estimated. The cost of an add-instruction in the ESOC emulator is illustrated in Table 4.

By applying the extra overhead on a single add instruction, we get 116.6 MIPS on a 3.5 GHz machine (compared to 166.6 MIPS without the extra overhead).

Table 4: Instruction Count For Add Instruction in the ESOC Emulator

Operation	Mem Access	Ops	Total
extractField	2	2	4
Reg-Reg op	6	1	7
PC update	2	1	3
Time update	2	1	3
Total	12	5	17

4 Results

The following subsections detail the results of the experimental validation.

4.1 Synthetic Benchmarks

For the single instruction case, a special customised version of the ESOC Emulator was used that applied custom register allocation when compiling the emulator core; effectively eliminating the VAC overhead. The results for these tests are illustrated in Table 5.

Table 5: Theoretical and Experimental MIPS for Add Instructions

Config	Theoretical	Experimental
No VAC	166.6 MIPS	175.0 MIPS
With VAC	116.6 MIPS	100.8 MIPS

The Dhrystone benchmark was executed on different configurations of the ESOC Emulator and T-EMU 2.0 (the next generation Terma Micro-processor emulator). The results for these tests are illustrated in 6. The tests included the ESOC emulator in decode-dispatch and threaded mode and for the decode-dispatch mode, also with and without the MMU in the system. For T-EMU 2.0, tests where carried out using the same application as for the ESOC emulator.

Table 6: Measured MIPS with Different Emulators and Configurations

Emulator	MIPS
ESOC no-mmu dispatch	57.5 MIPS
ESOC no-mmu threaded	66.4 MIPS
ESOC mmu dispatch	25 MIPS
T-EMU 2.0 no-mmu	90.0 MIPS
T-EMU 2.0 mmu	90.0 MIPS
TSIM no-mmu	60 MIPS[2]

4.2 Software Validation Facility Integration

The boot performance is illustrated in Tables 7 and 8. The measured data include the wall-clock time for executing the first 100 SRT-seconds of the booting system, and the WCT for the second 100 SRT-seconds of the just booted software.

Table 7: LEON3 SVF Boot Performance (Wall Clock Seconds)

Emulator	0-100	100-200
TSIM	92.4	35.0
ESOC Emu	47.5	12.04

Table 9 gives the average value for the time between sending a TC to the running OBSW and the delivery of the TM response in both SRT and WCT for both SVF configurations.

Table 8: LEON3 SVF Boot Performance (Speedup)

Emulator	0-100	100-200
TSIM	1.08 TRT	2.86 TRT
ESOC Emu	2.10 TRT	8.30 TRT

Table 9: LEON3 SVF TC to TM Response Time

Emulator	SRT	WCT
TSIM	0.15165	115.0
ESOC Emu	0.15155	21.4

5 Discussion and Conclusions

We have defined a theoretical model predicting emulator performance that was also validated to be reasonably accurate by test. This together with the other evaluation described here, led to principles that have guided the development of T-EMU 2.0. Which is shown here to be 40% faster than the ESOC Emulator (without MMU) and 260% faster when including the MMU. The massive increase in the case of an MMU in the loop is due to the use of a highly efficient address translation cache in T-EMU 2.0.

We have also shown that the choice of emulator can have a major impact on SVF performance. Although, while TSIM can run faster than measured, the use of the AMBA emulation interface in this case results in significant slowdowns of the emulation (the ESOC Emulator kept at least the RAM and ROM models inside the emulator). On one hand this enables TSIM to be cycle-accurate as the AMBA-busses are simulated, on the other hand, this significantly slows down MMIO and memory emulation.

There is also the issue of slower MMIO device lookups when the address decoders are outside the emulator core and the custom address decoders are not written in an efficient way (this has been known in some circles as the I/O bottleneck). This problem is however an artificial problem, created by the use of inefficient data structures used by the address decoders. This has also been a driving factor in the T-EMU 2.0 development. T-EMU 2.0, which, although not quantified in this paper, does not suffer from this problem.

Future work includes extending the benchmarks and testing T-EMU 2 in the SVF configuration and running the synthetic benchmarks also on TSIM.

References

[1] SIMULUS. `http://www.esa.int/Our_Activities/Operations/Ground_Systems_Engineering/SIMULUS`.

[2] TSIM2 ERC32/LEON simulator. `http://www.gaisler.com/index.php/products/simulators/tsim`.

[3] Mattias Holm. The Terma Emulator Evolution. In *Simulation & EGSE Facilities for Space Programmes*, 2015.

[4] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.