High Performance Microprocessor Emulation for Software Validation Facilities and Operational Simulators

Dr. Mattias Holm *Terma, Leiden, South-Holland, 2316 XG, Netherlands*

Micro-processor emulators are essential tools for development, testing and deployment of on-board software. Emulators are in principle simple systems, but achieving high performance is often contrary to other software development goals such as maintainability and portability. In addition, the advent of space qualified, multi-core high performance processors have significant impact on the design and implementation of emulators for use in SVFs and operational simulators. This paper introduces the new T-EMU 2, a high performance, multi-core processor emulator, currently supporting the emulation of the SPARCv7 and v8 architectures (including the ERC32 and LEONx). T-EMU 2 is 50% faster than the nearest competing system and is bundled with a sophisticated API whose purpose is to simplify device modelling and ensure the absence of the so called I/O bottleneck. T-EMU has been designed using the LLVM compiler framework to be extendable with new target architectures and new emulation methods such as binary translation.

Nomenclature

Host	The computer platform on which the emulator is running
Target	The emulated computer platform.
API	Application Programming Interface (the contract between programmer and compiler).
ABI	Application Binary Interface (the contract between compiler and processor).
SRT	Simulated Real Time (simulated time, may progress faster or slower than WCT)
WCT	Wall Clock Time (real-world time as on your wall clock)

I. Introduction

Software based micro-processor emulators, which are simulating a micro processor's instruction set, are important tools in the development and testing of on-board software, operational procedures, and for the training of spacecraft operators.

Emulators are pieces of software that provides three primary functions, firstly they provide an instruction set simulation of a micro-processor, secondly they provide a mechanism for memory simulation and thirdly they provide means for simulating I/O operations. Each of these functions is usually tightly coupled, as for example the instruction set simulator must have access to memory, and the I/O simulation system is often exposed as memory mapped I/O operations (MMIO) (coupling the I/O simulation with the memory simulation system).

The instruction set simulation part is typically either an interpreter or a binary translator. In order to achieve high performance, instruction set simulators often have to optimised on a very low level. The main goal for optimising an emulator, is actually to minimise the number of host instructions needed to simulate one target instruction. In-fact, to achieve high performance, every design decision must be taken with this in mind.

Interpreters can be implemented in various ways. The naïve implementation of an instruction set interpreter would see the implementation of one function per target instruction and then a large main emulation function that decodes a fetched instruction word to a function pointer and then calls that function. However, such an emulator is often very slow due to the overhead of a function call per instruction executed. Therefore,

interpreters are often *threaded*, which means that instead of having separate functions per emulated target instruction, all instruction implementations will be embedded in one main function. Instruction dispatching is then done using *computed gotos* instead of indirect function calls. This saves the overhead of preparing the stack for the called function.

Binary translators can be either static or dynamic, most emulators use dynamic translators (and static translation will not be discussed further in this paper). In a dynamic binary translator, target instruction blocks are translated host code at run time. The main advantage of a binary translator is that common operations (such as updating cycle and program counters) can be eliminated and done once per translated instruction block, in addition to this, the instruction decoding can be eliminated as well saving a few extra host cycles. Binary translators are however firstly quite complex to implement, and secondly relatively slow while translating target code. In the common case where code is repeatedly executed, a binary translator is usually a lot faster than an interpreter. It should be noted that it is possible to combine both interpretation and binary translation in order to eliminate the translation step for code that is executed only once, such hybrid execution is often done in high performance scripting language interpreters. For example, in the WebKit JavaScript engine, a multi-tier approach is used[1] where several interpretation and optimisation strategies are mixed with binary translation. The different execution strategies are selected based on how many times a code segment is executed. Similar approaches can be applied to an emulator as well.

It should be noted for those that are familiar with simulation software, that an instruction set simulator is a discrete event simulator with special case handling to speed up execution of sequences of emulated instructions (each executed instruction can be seen as a discrete event). Virtually all emulators provide abilities to post events in a simulated processor's timed event queue, interleaving instructions with other timed events. These timed events will in many cases be used to simulate various delays such as simulating on-chip timers, the time taken for transmitting messages on some bus or the time taken to execute a DMA[a] transaction. In addition to the hardware events, it is possible to post other timed events such as integration steps on the CPUs event queue.

During software development, emulators offers the ability for every developer to use a representative system (with the correct endianess, device models, etc) without needing access to the hardware which often is not yet available or simply too expensive to offer to every developer. For example, when NetBSD was ported to the AMD64 architecture, the port was done using the Simics full-system simulator[3] (a full-system simulator is essentially an emulator with a full set of device models that in the end provides the simulation of a whole computer system).

Additionally, emulators offers added advantages such as the ability of doing *non-intrusive* debugging. Also, by being *fully deterministic*, it is always possible to replicate a bug which has already been observed in the emulator. This means that the elusive *heisenbugs* that hides themselves when being observed in a debugger, can easily be replicated and tracked down as observation does no longer change the state of the software.

The use of emulators is not only limited to direct software debugging and development, but they can also be used for more complex system, including the use modelling of networks of either multiple computers or different types of modelled remote terminals. For example a virtual 1553 bus can be created inside an emulator, connecting two different virtual computer systems together.[11] Such setups can for example be used for bus-traffic budget analysis and low level bus protocol debugging.[11]

A special case use of emulators which is commonly used in the space sector are the *software validation facilities* (used for on-board software testing), and *operational simulators* (used for training of operators and testing of operational procedures). These, although very similar in concept, have different requirements of for example timing accuracy and performance. With an SVF, it is acceptable to sacrifice performance for more accuracy, while in an operational simulator, the accuracy will be sacrificed in order to achieve higher performance. In operational simulators, using the real flight software provides a significant advantage as it replicates the majority of the software bugs that exist in the real spacecraft. Not simulating these bugs when testing operational procedures could result in the loss of mission in worst case.

Modern ESA missions use the SPARCv7 or SPARCv8 processor architectures for their flight computers processors, more specifically the single core ERC32 and LEON2 processors. Upcoming missions will however be able to use the quad-core NGMP processor, a new high performance LEON4-based processor.

The more modern flight qualified processors are causing problems for the use of emulators in several aspects. Firstly, interpreted emulators typically have a slowdown of 25-50 times compared to the host

[a]Direct Memory Access

processor (i.e. a 2.5 GHz CPU can emulate a 50-100 MHz target processor in real-time). Secondly, the introduction of multi-core processors results in the need to support the emulation of these systems also in the emulator.

The current operational simulators used in Europe are often struggling to achieve real-time performance, and with the advent the new high performance quad-core NGMP processor it is likely not practical with existing tools. As the main bottleneck is typically the emulator, the T-EMU 2 emulator framework project was started as an internal Terma project.

This paper is organised as follows, in Section I.A, related work is described, Section II discuss the architecture and algorithms used in T-EMU 2. Section III presents a more detailed overview of different use cases. Section IV describes performance measurements have been done and compares the numbers with other (single-core) emulators. Finally, Section VI draws conclusions and discuss future work on T-EMU 2.

I.A. Related Work

Several emulators have been written and implemented by others, there are a number of commercial and open source offerings available.

General purpose high performance emulator systems such as Simics[7,16] and OVPSim[4] exists. Simics is produced by Windriver Systems and provides very high performance emulation of multi-core targets. Simics is also equipped with a very sophisticated API that can be used to implement device models. Similar tools are also available in OVPSim. Another system is QEMU,[8] an open source emulator. QEMU is well performing, simulates multi-core processors but is licensed under the GPL which at least some users find unacceptable for using for the development of a simulator (since the GPL would need to apply to the whole simulator). The mentioned systems are typically used as full system simulators, meaning that they simulate a complete computer system (including busses, remote terminals, sensors, and actuators).

In addition to the fully system simulators, a number of other SPARCv8 emulators exist including the ESOC emulator and TSIM. Both of these are instruction set simulators, and provides only a rudimentary set of device models (such as timers and UARTs). These emulators are typically used as part of a larger simulator, where the emulator is seen as a component and not the central part of the system. Neither the ESOC emulator nor TSIM supports the emulation of multi-core processors however.

II. Architecture

Writing a processor emulator is not conceptually difficult. However, achieving high performance is difficult, as high emulation performance typically requires low level optimisations that can only be done in assembler, or in very low level C code. When implementing emulators on this level, the implementation becomes target and host specific, meaning that if other targets have to be added, items such as instruction decoders must be rewritten from scratch. In the case of instruction decoders, an emulator typically provides an assembler and dis-assembler and these need specialised versions of the instruction decoder. Thus, the same code has to be repeated at least three times.

The consequence of this is that it is very difficult, if not even impossible, to provide a really high performing emulator in a high level out-of-the-box programming language while ensuring the *maintainability* and *portability* of the system.

T-EMU solves this by using domain-specific languages and compiler intermediate representation code. Processors are described in an abstract way which ensures that the bit patterns used for instruction decoding are firstly humanly readable, and secondly, they can be analysed and transformed automatically by running a generic decoder generator that can generate instruction decoding for any fixed width instruction set. To save on development time, we did not implement a completely new language for implementing the emulated instructions, rather T-EMU reuses open source tools from the LLVM compiler project.[2]

II.A. Instruction Decoding and Semantics

Instructions in T-EMU are defined using a combination of the *LLVM TableGen* and *LLVM Assembler* languages.

TableGen is a data description language, which is processed with a program that assigns semantics to the TableGen records (very much like XML is assigned meaning by a processing step). TableGen was developed to provide a simple and maintainable way to encode the instruction bit patterns needed by instruction

decoders and encoders provided by the LLVM toolchain. TableGen provides record types that can be used to embed code snippets associated with the instruction bit patterns. In our emulator, these snippets are used to describe the emulated instruction semantics. Note that, while LLVM already have descriptions for instructions of many different architectures and it provides semantics for different target instructions in the form of the LLVM instruction selection DAGs[b], these DAGs does not encode the full semantics of the instructions (which for example also include raising traps when certain conditions are fulfilled); also in our emulator we would potentially want to support more targets (but not more hosts) than are available in LLVM so in the end T-EMU does not reuse the LLVM instruction definitions, but only the same tools.

In Listing 1 an example add instruction as provided in the TableGen file for the SPARCv8 target is shown.

The defm statement instantiates the ri_inst_alu class inserting the semantics between the register and immediate access procedures (see the `# sem #` concatenation operation). The multiclass ensures two instructions are emitted, the register-register and the register immediate variants. Note that the functions called with the *emu__* prefix are known as emulator intrinsics. They primarily serves as a way to hide the low level details of the CPU structure from the CPU definition file. Intrinsics will typically be inlined at some stage in the compilation pipeline, meaning that an actual function call in the code does not impact performance in any way.

Listing 1. Example TableGen Instruction Description

```
multiclass ri_inst_alu<bits<2> op, bits<6> op3, string asm, code sem>
{
  class _rr {
    let i = 0;
    let semantics = [{
        %r1 = call i32 @emu__getReg(i5 %rs1)
        %r2 = call i32 @emu__getReg(i5 %rs2)
    }] # sem # [{
        call void @emu__setReg(i5 %rd, i32 %res)
    }];
  }
  class _ri {
    let i = 1; // Immediate variant, note that inst{13} =  i
    let semantics = [{
        %r1 = call i32 @emu__getReg(i5 %rs1)
        ;; Sign extend instruction immediate field
        %r2 = sext i13 %simm13 to i32
    }] # sem # [{
        call void @emu__setReg(i5 %rd, i32 %res)
    }];
  }
}
defm add : ri_inst_alu<0b10, 0b1010101, "add", [{
    %res = add i32 %r1, %r2
}]>;
```

II.B. Compilation Pipeline

While initial implementations of our emulator generation pipeline, used multiple separate tools and code transformation passes the later versions (from T-EMU 2.2 and onward) consolidates most of the steps in a single program which simplifies the setup of the build system considerably. In any case, our initial approach which have been discussed in 13,14, is largely maintained in the latest versions except for the integration of the different steps in one tool.

An emulator core is generated by loading the TableGen file in the *emugen* program, *emugen* then emits an interpreter loop (currently it supports the *indirect threaded* interpretation algorithm with a single table

[b]Directed Acyclic Graph

decoder. Support for generating a *predecoded direct threaded* based emulator is under development (without having to change the instruction definitions). The two interpreter algorithms are described further in 17.

The general pipeline for the emulator generator is illustrated in Figure 1. Emugen loads the TableGen file, and emits an assembler and disassembler from the instruction descriptions. It then emits LLVM functions for each instruction, links in the "intrinsics"-file and emits the emulator loop/interpreter function into the same module.

After the interpreter has been generated and inserted into the LLVM module, several LLVM code analysis and transformation passes are executed on the module. The passes include for example instruction inlining (which eliminates the instruction functions and embeds them inside the main emulator function), and domain specific optimisations. For example, in the normal cases a trap or interrupt are raised using the indirect control transfer function longjmp, however for longjmp-calls inside the emulator loop this is not needed, and direct jumps can replace the call to longjmp to specific labels instead.

After the custom transformation pipeline has been executed, the LLVM bitcode is finally emitted by the emugen tool as a file. The bitcode is then compiled to object code which in turn is linked into a CPU emulation library.

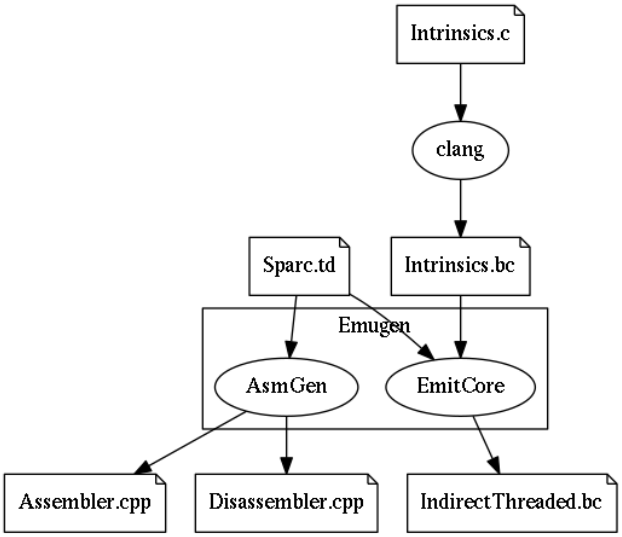

Figure 1. Emulator Generator Pipeline in T-EMU 2.2

II.C. C-API

T-EMU 2 was designed to not only provide the emulation of multi-core processors, but to be able to be used both as a plugin in an existing simulator where the emulator is a component, and to be able to use T-EMU 2 as a full system simulator without a separate simulation framework (the latter approach actually simplifies the design of the simulator as it centralises the model scheduling and event system to the emulator, but as T-EMU does not exist in a vacuum the emulator must provide ability to integrate as a component in existing systems).

Thus, T-EMU 2 provides a sophisticated C-API, centered around the T-EMU object system, which provides the most common capabilities needed by a simulator framework. Including interfaces, property publication and checkpointing (also known as breakpoints in some simulators).

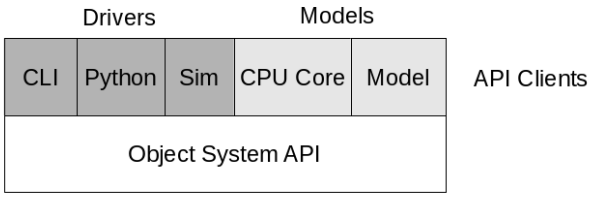

Figure 2. API From a Layered Perspective

In Figure 2 the layering and separation of components using the T-EMU API is illustrated. The *Command Line Interface* (CLI) is built on-top of the Object System API, the CLI does not have direct access to user provided models or the CPU models, all control is done using the Object System API. Typically one can divide the users of the Object System API in two categories, drivers and models, where the rivers are API clients that drive the simulation (i.e. call the CPUs models' run functions through their CPU interfaces exposed via the object system), this include the CLI, python scripts, third party simulators etc. Models on the other hand are simulation components, simulating for example a CPU or some other device (e.g. a UART), these components are typically passive and only activated due to a driver advancing the time.

The public API has been defined using the C language for a number of reasons, one of the main ones being that the emulator CPU core must be able to talk to the memory system. This is not practical would the API be implemented in C++ or some other language without a defined (and relatively simple) *ABI*. In addition, it is trivial to wrap a C-API for use in different languages such as Python, and it can be used directly from C++.

In Listing 2 a simple T-EMU plugin using the object system is shown. The `TEMU_PLUIGIN_INIT` function will be called when the plugin is loaded, it calls the `registerClass` function to register classes it provides with the temu runtime. T-EMU can then create new objects by class name. The advantage with this is that the user can easily at runtime create new objects in the command line interface and connect these to the rest of the system at any point of time. Using the plugin defined in the example, an instance can be created using the command: `object-create class="MyClass" name="obj0"`.

Listing 2. Class Example

```
typedef struct {
  uint32_t MyScalar;
} MySimpleClass;

void* createMySimpleObject(const char *Name, int Argc,
                           const temu_CreateArg *Argv);
void destroyMySimpleObject(void *Obj);

void scalarWrite(void *Obj, temu_Propval Pv, int Idx);
temu_Propval scalarRead(void *Obj, int Idx);

TEMU_PLUGIN_INIT {
  temu_Class *Cls = temu_registerClass("MyClass",
                                       createMySimpleObject,
                                       destroyMySimpleObject);
  temu_addProperty(Cls, "myScalar",
                   offsetof(MySimpleClass,MyScalar),
                   teTY_U32,
                   1, // Number of elements (1 = scalar)
                   scalarWrite,
                   scalarRead,
                   "Scalar is used for smurfing 42 smurfs");
}
```

Some emulators (e.g. the ESOC emulator and TSIM) provides a mechanism where a single callback is used for all memory mapped device accesses. The simulator writer then has to implement an address decoder, which often ends up as a list or vector search. In the best case this is done using a binary search or a map, but linear scanning for the device model based on the physical address of the memory access is not uncommon. These are of-course either $O(\lg n)$ or $O(n)$ in complexity with several memory accesses being performed in a cache unfriendly way. To avoid this performance bottleneck (which is known as the I/O bottleneck), T-EMU 2 provides a cache friendly $O(1)$ address decoder based on the same idea that is behind virtual memory page tables.

In addition, several standard interfaces have been defined using the object system APIs. These include the memory access interfaces, which can be used to implement memory mapped I/O models, serial port interfaces, interrupt interfaces etc.

In order to facilitate communication with external simulators, the object system support the notion of an external object. Using the external object support, existing SMP2 models can easily be integrated by implementing the memory access interface.

II.D. Simulation Methods

In order to achieve high performance when running a machine with multiple CPU cores (e.g. the quadcore NGMP), the simulated CPUs will be *temporally decoupled*. That means that the CPUs will not agree on what the current time is. To ensure that the CPUs in the end roughly agrees, the user must set a time quanta (in SRT) for the simulation and T-EMU will then run each processor using a single threaded round-robin scheduler (single threaded scheduling guarantees a fully deterministic simulation). Round robin scheduling is commonly used by several different emulators (including QEMU[9]) in order to emulate multicore systems. The time quanta prevents the CPUs from running ahead of each other too much. One could imagine the

havoc that would be generated by one core being in idle mode and simply advancing the event queue while another running a real application. The clocks in idle CPUs often advance hundreds of times faster than wall-clock time.

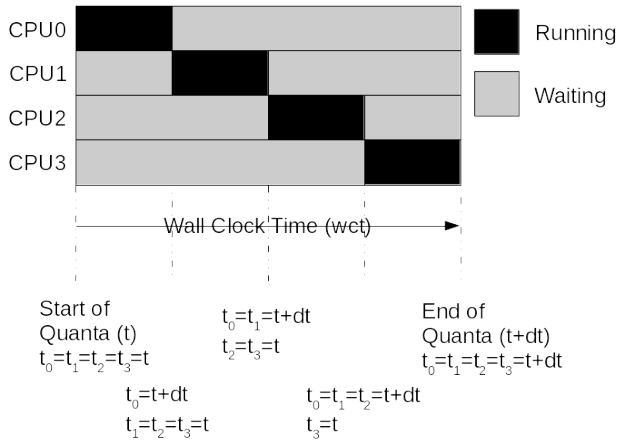

Figure 3. Wall clock vs simulated real time

This is better illustrated in Figure 3. As can be seen, initially at the start of the quanta, all CPUs have the same notion of $SRT = t$, when a CPU is scheduled, its clock is steadily increasing until the end of quanta time ($SRT = t + dt$). While a CPU is scheduled, the previously scheduled CPUs have already reached $SRT = t + dt$ and the following CPUs are waiting to be scheduled and are still at the start of the quanta ($SRT = t$).

As some type of models may assume a synchronised time between processors (e.g. timers), T-EMU supports the creation of synchronised events. Such events must naturally be posted with an offset of being at least in the next quanta. If events are posted as synchronised events, the different CPUs will have roughly the same notion of time, with roughly it is meant that the maximum difference in agreement is bounded by the largest instruction cycle cost (e.g. if a divide instruction takes 35 cycles, and if this was the last instruction in a quanta, then the timing offset of that CPU may be up to 35 cycles with respect to the other CPUs). The reason for this is that each instruction that is started will finish and advance the clock before the emulation cores are terminated.

Each CPU switch in the scheduler has a certain cost thus one typically achieve higher performance with a longer quanta, however, certain operations such as busy waiting on other CPU cores can be significantly slowed down when increasing the quanta (e.g. a core may wait for a whole quanta instead of a few cycles on spin-locks or IPI[c] deliveries). The best value of the quanta is thus something that must be experimentally determined by the user, and it can vary depending on the type and phase the application is in.

It is possible for the user to adjust the time quanta of the simulated machine at run-time. A use case of this dynamic quanta configuration is to initially run faster, and when homing in on a software bug such as a race condition which is dependent on the accuracy, the quanta can be reduced by the user.

The current mechanism for multiprocessor and multicore simulation in T-EMU is as mentioned a single threaded round robin CPU scheduler and utilises the algorithm described above. Future versions of T-EMU may include optional support for multi-threaded emulation which can increase performance at the expense of having a fully deterministic behaviour. That said, even a multi-threaded simulation approach will by necessity also use quantas to ensure that time is kept roughly in sync.

II.E. Memory Simulation

The memory simulation in T-EMU is based on a transaction model, with separation of fetch, read and write operation. Each device model that needs to provide memory mapped registers (or other memory buffers) must implement the memory access interface.

Whenever a memory operation is executed, the memory model will look up the device model associated with the address (using an approach based on the multi-level page tables used by memory management units) and invoke that model's read or write handler.

As the astute reader may realise, this would be very slow when dealing with normal memory (ROM and RAM) which is accessed at least once per instruction (in order to fetch the instruction from memory). As a solution to this problem, most emulators, including T-EMU 2, provides a construct which is best described as a *software TLB*. T-EMU 2 calls this mechanism an *Address Translation Cache* (ATC), while other emulators such as QEMU[5] and Simics[15] use different terminology. These caches are essentially directly mapped look-up tables that ensures that the common case for decoding an address to a memory page is only a few instructions long.

[c]Inter Processor Interrupt

Listing 3. Memory Access Interface

```
typedef struct temu_MemTransaction {
  uint64_t Va;       //!< 64 bit virtual for unified 32/64 bit interface.
  uint64_t Pa;       //!< 64 bit physical address
  uint64_t Value;    //!< Resulting value (or written value)

  //! Log size of the transaction size
  uint8_t Size;

  //! Used for device models, this will be filled in with the offset
  //! from the start address of the device (note it is in practice
  //! possible to add a device at multiple locations (which may happen in
  //! some rare cases)).
  uint64_t Offset;
  void *Initiator; //!< Initiator of the transaction
  void *Page;      //!< Page pointer (for address caching)
  uint64_t Cycles; //!< Cycle cost for memory access
} temu_MemTransaction;


typedef struct temu_MemAccessIface {
  void (*fetch)(void *Obj, temu_MemTransaction *Mt);
  void (*read)(void *Obj, temu_MemTransaction *Mt);
  void (*write)(void *Obj, temu_MemTransaction *Mt);
} temu_MemAccessIface;
```

II.F. Advantages of the Architecture

The architecture described in this paper has several advantages. The primary advantage is that the instruction definitions are independent from the emulation method. Consequently, it is relatively easy to extend the system with for example binary translation.

Another advantage for T-EMU is that the supporting tools (e.g. the emulator generator) have all been designed to be target independent. This target independence allows additional CPU architectures (e.g. ARM, OpenRISC, PowerPC, RISCV, etc) to be added relatively quickly would a customer need such support.

At the time of writing, a new interpreter method is being implemented, which in the end does not require any modifications of the instructions definitions, only a new interpreter emitter backend is needed.

III. Emulator Usage

III.A. Software Development

One of the more important use cases for an emulator is on-board software development and unit testing. As it is often fairly expensive to acquire space qualified hardware (boards are often 20k or more, sometimes well above 100k), especially early on in the project, having access to a simulation environment can be beneficial.

Running unit tests on an emulator instead of a completely different architecture, ensures that the software handles low level issues such as endianess properly and it provides stress for the target compilers early on in the development process. The risk of bugs in the target system compiler should not be understated, and the earlier these can be caught the better.

In addition to ensuring representative endianess and the use of the target system compiler, emulators provide additional benefits for software development. These include fully deterministic execution, ability to checkpoint simulation state and for some emulators the ability to run code backwards. One of the more important aspects is however the ability to use the emulator command line interface for running fully automated tests. T-EMU for example allows for the execution of Python scripts together with the emulator, these scripts can for example intercept and analyse emulated bus traffic (e.g. serial port traffic), greatly simplifying unit testing.

III.B. Performance and Budget Analysis

Another use case for an emulator is to run performance analysis. Although not fully representative in all cases, the accuracy of an emulator is in many cases much better than hand made estimates. Performance analysis can be executed routinely and automatically, even before the hardware is available.

While timing accuracy is likely not 100 percent accurate, memory and some bus traffic budgets can be fully accurate. A virtual 1553 bus could for example provide runtime warnings if the on-board software ever exceeds certain thresholds.

III.C. Software Validation Facilities

A software validation facility (SVF) is a full system simulator whose purpose is primarily to execute more complex system tests. SVFs can be used to test that the final on-board software does not exceed its budget (see Section III.B) and that it interacts with its environment (e.g. through simulated remote terminals) correctly.

For an SVF, unless hardware is in the loop, faster than real-time performance of the emulator is usually not critically important as a self-contained software-only SVF will use virtual time only. Tests in SVFs are often based on uplinking TCs and inspecting TM responses.

III.D. Operational Simulators

An operational simulator is a full system simulator (similar to an SVF) but with the integration into the ground infrastructure. In an operational simulator, the primary user interface to the emulator is therefore the mission control system. Operational simulators are used for both training of operators and for the testing of different types of on-board control procedures and telecommands on the real on-board software, but without having to risk the spacecraft or having the need to keep spare hardware around for this purpose.

Emulators provide a great deal of flexibility for operational simulators, and as they run the real flight software, most bugs in the flight software is also simulated. That is, when testing operational procedures, the test will be against the real software, not its specification.

IV. Experiments

In order to evaluate the performance of the emulator in its current state, three types of experiments have been run. These include primarily synthetic benchmarks and some real-world applications. A more detailed evaluation of the performance is available from 12.

When measuring emulator performance, there are two ways to do this, the first is to measure *Times Real-Time* (TRT) or its reciprocal *slowdown*, the second is to measure the number of emulated instructions per second (MIPS). The two measurements have different usages, the TRT numbers are useful for determining the end user performance (which is often tightly connected to the simulator requirements) or when comparing the emulator to real hardware. The MIPS number is useful when looking at the raw emulator performance and when comparing different emulators. It should be noted that these can be very different. TRT, but not MIPS, is effected by the virtual clock frequency of the emulated processor and whether the emulated CPU enters idle mode. In idle mode, the emulator will fast forward time until the next event (e.g. an interrupt). When the Linux system is idling at the prompt, it is not unusual to see the SRT advancing hundreds of times faster than WCT, while the MIPS rating remains roughly the same.

Synthetic benchmarks include the execution of the Dhrystone integer benchmark, and while its qualities as a benchmark can be discussed, it is the standard application used for testing emulator performance. Numbers from the benchmark are widely published by different emulator vendors. A slight complication from the use of Dhrystone, is that the benchmark produces its own MIPS numbers, these numbers does not have anything to do with the emulated instruction count and will be effected by properties such as the virtual clock frequency of the emulated processor. When referring to MIPS, here, unless explicitly stated, the reference is to millions of emulated instructions per second.

The real world benchmarks which the emulator undergoes include the booting of Linux. However, this being a non-widely available benchmark, we are unable to provide any comparative numbers for these tests. Suffice to say, boot performance of Linux produced roughly the same emulated MIPS as the Dhrystone benchmark did.

V. Results and Performance

Table 1 illustrates the performance achieved when running Dhrystone on different SPARCv8 emulators on a 3.5 GHz host PC.

Table 1. 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[d]
Simics no-mmu	300 MIPS[e]

We can draw two conclusions from these figures. The first conclusion is that the MMU implementation in T-EMU is superior in performance to the ESOC emulator. This stem from the use of a highly efficient address translation cache. The second conclusion to make is that for being an interpreted emulator (at the moment), T-EMU is very fast. In-fact, in 12 it was showed that the theoretical performance of an interpreter is around $1.5 - 1.7$ times that of the ESOC emulator (between 94 and 115 MIPS), and T-EMU 2 is very close to these numbers.

The only higher performing system we had access to data for, is Simics. Simics is a binary translator which therefore is able to score higher performance than an interpreter is capable of. As T-EMU 2 has been designed to be extensible enough that binary translation can be introduced, those numbers should be reasonably representative of the future performance of T-EMU

VI. Future Work and Conclusions

As mentioned in Section II, the architecture was designed with flexibility in mind. Thanks to the use of the LLVM toolchain and the abstraction of instruction decoders and semantics it is possible to migrate to more high performing emulation methods such as *direct pre-decoded threaded interpretation* and *binary translation* without modifying the instruction definitions. In the former case, the instruction decoder is replaced and the emulator generates intermediate code (IR) that contains pointers to the instruction emulation routines. The decoding for this type of IR is quite efficient and fast and allows for introduction of dynamic features (such as the dynamic enabling and disabling of fast cache emulation).

In a *binary translator*, target code fragments are translated either offline (*static* translation) or during run-time (*dynamic translation*) to native host code. The *dynamic binary translation* (DBT) approach is the most general. In the *static binary translation* (SBT) case, the translation may be subject to more offline optimisations during translation, but it suffers from being less general and not being able to deal with self-modifying code (which include among other items boot loaders).

To support a hybrid system, where interpretation is used initially but binary translation is used for hot code, it is advantageous to use the pre-decoded emulation method for the interpreter part as it allows for the replacement of IR instructions representing branches with jumps to the dynamically generated code without any significant performance overhead.

At present, Terma is working on supporting *pre-decoded binary translation* in T-EMU to pave the way to full support for binary translation later on.

In addition to more advanced instruction set simulation methods, the APIs of T-EMU and surrounding tools are also subject to evolve. A current development is a dedicated domain specific language designed to make it easy to implement memory mapped device models. Such a language eliminates the direct interactions the model writer must do with the T-EMU object system, and provides natural mechanisms for expressing memory mapped registers with fields, post emulator events and to do automatic check-pointing. In an experimental translation of a UART model written in C++, the line count dropped from around 1100 lines to around 300, which is substantial reduction in lines of code.

References

[1]Introducing the WebKit FTLJIT. https://webkit.org/blog/3362/introducing-the-webkit-ftl-jit/. Accessed: 2016-03-29.

[2]LLVM Website. http://llvm.org/. Accessed: 2016-03-14.

[3]NetBSD/amd64. http://wiki.netbsd.org/ports/amd64.

[4]OVPSim Website. http://www.ovpworld.org/. Accessed: 2016-03-14.

[5]QEMU internals: softmmu. http://vm-kernel.org/blog/2009/07/10/qemu-internal-part-2-softmmu/. Accessed: 2016-03-22.

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

[7]Windriver Simics Website. http://windriver.com/products/simics/. Accessed: 2016-03-14.

[8]Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[9]Jiun-Hung Ding, Po-Chun Chang, Wei-Chung Hsu, and Yeh-Ching Chung. PQEMU: A Parallel System Emulator Based on QEMU. In *17th International Conference on Parallel and Distributed Systems*, pages 276–283, 2011.

[10]Jakob Engblom and Dan Ekblom. Simics: a commercially proven full-system simulation framework. In *Simulation & EGSE Facilities for Space Programmes*, 2006.

[11]C. W. Mattias Holm. A Fully Virtual Multi-Node 1553 Bus Computer System. In *Data Systems in Aerospace (DASIA)*, 2006.

[12]Mattias Holm. Emulator Performance Study. In *SESP 2015*, 2015.

[13]Mattias Holm. T-EMU 2.0: The Next Generation LLVM Based Micro-Processor Emulator. In *EuroLLVM 2015*, 2015.

[14]Mattias Holm. The Terma Emulator Evolution. In *SESP 2015*, 2015.

[15]P. Magnusson and B. Werner. Efficient memory simulation in simics. In *Simulation Symposium, 1995., Proceedings of the 28th Annual*, pages 62–73, Apr 1995.

[16]P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, Feb 2002.

[17]James E. Smith and Ravi Nair. *Virtual Machines - Versatile Platforms for Systems and Processes*. 2005.