High Performance Advanced Instruction Set Simulation

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

March 1, 2019

Abstract

Traditional interpreted emulators have served us well, however, due to advancement in flight computer processor performance, traditional fetch-decode mechanisms are no longer usable if one wish to achive high performance. This paper describes the development of an advanced interpreter engine based on the pre-decoded direct threaded method, and the integration of it with a binary translation engine.

1 INTRODUCTION

Existing interpreters of flight computer processors commonly used in European space craft, are mostly based on a technique known as interpretation. While interpretation can achieve real-time performance when simulating modern flight computer proecssors, the perormance achived is just above real-time. Faster than real-time execution is often seen as necessary for operational simulators, and is something that is nice to have for software validation facilities. The next generation processors that are currently being released adds significant problems due to higher clock rates. Thus other methods of emulation are needed.

The emulator cores embedded in simulators have been shown to be the main bottleneck in terms of improving performance. Terma has an existing full system simulation infrastructure indented for general purpose use (known as TEMU 2). The difference between TEMU and other simualtion infrastructures is that TEMU is centered around instruction level processor simulation. While the infrastructure is a full simulation system it has been designed to be embeddable (as a library) inside other simulators in order to fit the customer needs.

TEMU provides every feature that a simulation framework like SMP2 has on an interface level, but exposes a C API[1]. The use of a C API means that TEMU is ABI[2] stable and is easy to extend using other languages. Nothing for example prevents a user from implementing TEMU models in Rust, Python, C++ or any other language. This is a significant advantage for commercial deployment of the TEMU simulation engine. Note that integration with SMP2 interfaces are relatively straight forward using wrapper models that could potentially be automatically emitted if needed. Note that TEMU normally treats modelling as a coding activity since, most simulation models are simply translation of hardware documentation into code.

This paper introduces TEMU 3, the next version of the TEMU instruction set simulator. TEMU 3 builds on previous work on TEMU 2 as has been presented in [6, 7]. TEMU 3 adds two key capabilities: a new high performance interpreter using pre-decoded direct interpretation and a binary translation engine. The interpreter engine is based on the pre-decoded direct interpretation method [9], and the binary translation engine is based on LLVM [3] compiler framework.

A single TableGen based source file is the root source for the TEMU CPU core implementations, although it is exported into YAML for the generation of the binary translator code.

[1] Application Programming Interface
[2] Application Binary Interface

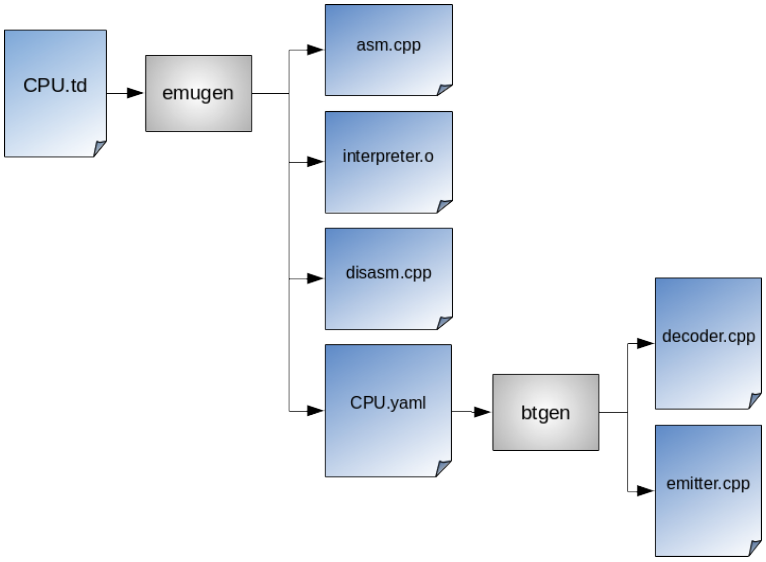

Figure 1: Transformation Flow of TEMU CPU Core

As can be seen in Figure 1 the CPU definition file in TableGen format (.td) is converted to a number of files. Including implementations of assemblers and disassemblers, an interpreter and an exported CPU core. The exported CPU core is loaded by the btgen tool which generates firstly an instruction decoder and secondly a code generator. Note that the TableGen file is an unstable format and the content itself is proprietary, the exported YAML file is in principle a stable format (it can be seen as a low level assembler like format for defining CPU cores, while not necessarily humanly maintainable).

2 PRE-DECODED DIRECT INTERPRETATION

The interpreter, which used to follow a traditional repeated fetch-decode dispatch model, which is fast enough for a SPARC and PowerPC processor, is not practical to simulate an ARMv7 processor due to the complexity in decoding the ARMv7 instruction set, in fact when introducing the ARMv7 model in the emulator, a mere 30 MIPS was achieved compared to the 250 MIPS of the SPARCv8 model.

By introducing an intermediate format that embedds a pointer to the instruction handler, it is possible to avoid the fetch-decode operation for every instruction. Instead a dispatch routine (or offset from a known address) is fetched as part of an intermediate format instruction. The intermediate representation format (IR) contains not only a pointer, but also parameters in the form of pre-decoded operands. While the operand format needs to be defined on a per instruction format basis, it is in general straight forward for a SPARC emulator core, since operands are packed in single fields on SPARC. For ARM and PowerPC, however operands are sometimes split in multiple fields. Meaning that the decode into IR will reorder the fields and ensure they can be extracted by at a single shift and mask operation.

The indermediate format is interesting in the aspect that it is possible to add pseudo instructions (or emulator control instructions) that do not exist in the simulated instruction set. This is used in order to eliminate additional branches as is e.g. required in the predecode format used in the ESOC Emulator (where each instruction fetch must consult a number of bits in the instruction word to check if it must be decoded). Thus the emulator comes with a number of special instructions that include, but is not neccesarily limited to:

Pseudo Instruction	**Params**	**Description**
pseudo.trampoline	Fragment pointer	Invokes translated code block.
pseudo.unknown	-	Undecoded instruction, will trigger fetch + decode.
pseudo.attribCheck	Attribute bits	Triggers checks for attributes (e.g. breakpoints)
pseudo.endOfPage	-	End of page marker

The attribute bits include in TEMU: *breakpoint*, *watch point read*, *watch point write*, *upset*, *faulty*, *user 1*, *user 2* and *user 3*. Which support runtime injections of SEUs, MEUs, non-intrusive break and watch points and bits that can be customised by the user in special attribute handling models.

The use a pseudo instruction to trigger attribute checks for example speeds up execution on pages with attributes with massive ammounts, since the pages in question can now be kept in the address translation caches and avoid unneccessary invokation of the memory models at every fetch.

There are also special target specific pseudo instructions for dealing with issues such as delay slot semantics for off page branches and delayed control transfer instructions (DCTI couples).

The pseudo instructions here could be used in order to add advanced functionality, e.g. we could add a pseudo instruction that triggers an , which would mean we could eliminate at least level one cache models and maintain reasonable performance.

In addition, the predecode format, as it is emitted at runtime, could be specialised by for example providing special instructions for instruction and data cache simulation, this would speed up cache simulation and make it possible to dynamically enable and disable it by simply invalidating the currently valid predecode IR.

Note that the memory system supports multiple pages of IR being associated with an emulated memory page. That means that supporting multi-ISA processors such as ARM (A32 + Thumb) and the new LEON micro controller (SPARCv8 + REX) is trivial, and switching instruction set simply switches the active IR page.

One of the main difficulties in the IR format is what to do when leaving a page by normal PC increments since we would like to avoid having to check for intermediate PC overflow at every instruction. The answer is simple, the *pseudo.endOfPage* instruction is added after the IR and any natural departure of the page will trigger the end of page handling, which include permission checks through the MMU model.

The other main difficulty is how to handle delay slots appropriately. Because if the target is in another page, then a permission check must be done, but as the target instruction is not executed right after the branch instruction (on architectures incorporating delay slots), the permission check must be delayed. This is accomplished using the two *pseudo.relinkPc/relinkNpc* instructions that triggers a reevaluation of the program counters (computing it from the intermediate PC) and permission checks. Thus, all branches are split into on or off page variants (meaning that decoding is dependent on the instruction and the PC). On-page branches can be handled quickly, while off-page branches which are more rare can be subject to additional checks.

Our intial implementation of the new interpreter used a 128 bit instruction format. This is suboptimal as we need two memory loads for each instruction and will need to use two registers on the host system to keep the IR instruction around. Despite this, the performance was inline with the previous fetch decode interpreter. A new 64 bit IR format is being developed, but it requires careful considerations on a per target basis, which is not the case for the 128 bit format. The 64 bit format will reduce memory pressure, and thus cache utilisation and we expect a signficant speedup when this is implemented.

The format for the 128 bit IR is defined as follows:

```
typedef struct {
  uintptr_t routine;
  uintptr_t params;
} ir_t;
```

While the 64 bit format is defined as follows:

```
typedef struct {
  uint32_t routineOffset;
  uint32_t params;
} ir_t;
```

The main difference is the reduction in parameter space, and the collapsing of the routine pointer into an offset from a known address (we use the emulator cores main interpretation function for this).

3 DYNAMIC BINARY TRANSLATION

The advances in the interpreter also allowed a general purpose pseudo operation *pseudo.trampoline* to be implemented (see Section 2), that is capable of invoking any out of core function following a specific signature.

TEMU 3 now comes with a binary translation engine that will translate code blocks at runtime. Code blocks are extracted using a single entry point (based on a branch target), and may have multiple returns (e.g. traps raised), the block is terminated either when reaching the next page, or when finding a branch. This is known as dynamic basic blocks, which differ slightly from static basic blocks as used in compiler frameworks. A static block has only a single entry point and

single exit point, while a dynamic basic block may consist of multiple static basic blocks if there is a fall through from one block to another. The identification of dynamic blocks is done at runtime.

The main advantage of this is two-fold. Firstly, instructions are further decoded by the translator since each instruction copy will be unique for the memory location it is associated with; in the interpreter one need to unpack dynamic instruction bits at runtime, such as register numbers and immediates. In the binary translator the translator does the unpacking and thus for an add instruction, the host code ends up more compact since immediate fields can be embedded in the host instruction stream. Secondly, by using the LLVM[3] framework for just in time code translation, the generated code can be further optimised using traditional compiler optimisations. For example a divide by an immedate power of 2, will now be emitted as a shift operation instead of having to execute the actual divide, and a divide by an immediate zero will omit any code for division operations and just raise an divide by zero exception (for SPARC processors that is).

Dynamic basic blocks are only executed if they will not trigger any events or breakpoints during their execution. The system then fall backs to interpretation, resulting in identical behaviour for fully interpreted mode. This simplifies testing and debugging of the translator since it is possible to run it in lock step with a reference interpreter only variant.

Since it would be costly to jump back and forth between the translator and interpreter for every block, basic blocks are chained dynamically when jumping on the same page.

A particular complexity was dealing with delay slots on the SPARCv8 target. This was solved by reordering the actual control transfer effect (when the block is exited (which could be chained)) and the delay slot instruction. Due to the relative complexity in getting this correct for DCTI couples, it was judged to be acceptable to fallback to the interpreter in those cases.

4 TIERED EMULATION

TEMU 3 uses well established ideas[4, 5], that today are commonly used in virtual machines for different virtual machine baesd programming languages and runtimes such as JavaScript[2], Java[8] and .net CLR[1], where compilation is done with multiple tiers, each tier resulting in successively better optimised code, but at the cost of increased translation time.

The principle of tierd translation was explored in the early SmalTalk 80 implementations[5], the Deutsch-Schiffman SmallTalk-80 implementation translated the source code into *V-code*, a byte code format, suitable for interpretation. V-code would subsequently be translated into *N-code* which was native code, that could use host machine control flow to a greater extent and would be better optimised since it ran on the machine directly.

TEMU 3 utilise a similar approach, but implements it differently. Each CPU model is described in an abstract CPU description language based on TableGen, the instruction sematics is emitted as LLVM IR, which can be either emitted in a function format suitable for calling from an interpreter (the interpreter core is also generated using an LLVM interpreter core generator and all instruction semantic functions are inlined in it). The same original description is then loaded by tool specialised in emitting a binary translator core by converting the LLVM IR code to C++ that re-emitts the same LLVM IR representation. The difference is that the C++ code can be specialised and pre-optimised at the time it is emitted. Meaning a more efficient IR is emitted since conditional code for avoiding reading data from e.g. the G0 register (which is hardwired to zero) can be eliminated before the code is even emitted.

In the level 1 tier, TEMU executes pre-decoded direct IR. Whenver a branch is executed a branch taken / not taken counter in the IR is incremented. When that branch counter overflows a value (which must be experimentally determined), the instruction notifies the level 2 tier that the branch is hot and the branch target should be optimised. The level 2 tier will then generate LLVM IR code, which is emitted and then installed by replacing the branch target level 1 IR instruction with a *pseudo.trampoline* instruction as described in Section 2.

5 PERFORMANCE OPTIMISATIONS

LLVM is not very fast if executed inline in the interpreter's instruction stream. To solve this problem, we only do a minimum amount of work when triggering a translation of a basic block, and the actual translation is after reading and decoding instructions, pushed into a stack which is then extracted by a translation thread. The translation thread executes the actual code generation and appends a patch list which is drained by the main emulator thread occationally. The main thread then patches the predecoded IR and installs trampoline routines in the IR which are responsible for invoking a translated target block directly.

In addition, when a branch at the end of a block jumps on-page (which is the common case), it is possible to chain the block and jump directly to the target block. Basic block chaining is infact fundamental to get good performance from a binary translator. Basic block chaining is currently implemented using low level patching of raw opcodes by keeping

track of when a branch delay slot has been executed. The locations for patches are defined using the LLVM patch point mechanism.

6 FUTURE WORK

We have established a baseline for multi tier binary translation in TEMU 3. That said, there are several items that can be improved.

A lost opportunity for optimisation is the register accesses. In TEMU, register windows are implemented using indirect register access, meaning it is very easy to switch register window (banked registers on ARM have the same problems). However, since the registers are pointed out indirectly, and are of the same type, LLVM's code optimisers have no idea whether or not the registers are aliasing. This can be resolved by eliminating the indirection, or by adding explicit aliasing information for use by the LLVM optimisers.

Another opportunity for improvement would be deep integration in TEMU, currently the binary translator is a separate plugin, by deeply integrating the binary translator core, a number of sub systems would be simplified since e.g. the code fragment manager could be integrated in the memory system. However it was decided to keep these separatelly initially in order to simplify debugging and speed up turn around time when developing the system.

One could also introduce additional tiers, where e.g. the LLVM based translator is moved to tier 3, and a tier 2 translator could emit translated code, without any significant optimisations. This would enable quicker emission of machine code, also speeding up code segements that are rarely executed.

It is also an option to add pre-translation when an ELF file is loaded. The translator could then statically infer all non-indirect branch targets and pre-translate these at load time. That will increase loading time, but will increase the amount of code that is executed in translated mode.

7 CONCLUSIONS

The introduction of dynamically translated code in both the interpreter (pre-decoding) and in the binary translator, means that dynamic instrumentation of code is now possible. Since code for instrumentation is omitted if it is disabled, there is no performance degredation in the nominal case just becasue the system supports instrumentation, which would be the case in a traditional decode-dispatch interpreter. Instrumentation plugins can include coverage collection, cache models, interference models and arbitrary user level code to be triggered based on certain conditions such as instruction type, opcode, etc.

In nominal case performance of the translated code is significantly faster than interpeted code thanks to pre-evaluation of instruction field extraction, and elimination of redundant operations (e.g. multiple reads of the same register).

REFERENCES

[1] A look at the internals of 'Tiered JIT Compilation' in .NET Core. `https://mattwarren.org/2017/12/15/How-does-.NET-JIT-a-method-and-Tiered-Compilation/`.

[2] Introducing the WebKit FTL JIT. `https://webkit.org/blog/3362/introducing-the-webkit-ftl-jit/`.

[3] The LLVM Compiler Infrastructure. `https://llvm.org/`.

[4] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 1–12, New York, NY, USA, 2000. ACM.

[5] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pages 297–302, New York, NY, USA, 1984. ACM.

[6] Mattias Holm. Emulator Performance Study. In *Simulation & EGSE Facilities for Space Programmes*, 2015.

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

[8] S. Oaks. *Java Performance: The Definitive Guide: Getting the Most Out of Your Code*. O'Reilly Media, 2014.

[9] 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.