Fast Functional Simulation with a Dynamic Language

Craig S. Steele and JP Bonn
Exogi LLC
Las Vegas, Nevada, USA
steele@exogi.com

***Abstract*—Simulation of large computational systems-on-a-chip (*SoCs*) is increasing challenging as the number and complexity of components is scaled up. With the ubiquity of programmable components in computational SoCs, fast functional instruction-set simulation (*ISS*) is increasingly important. Much ISS has been done with straightforward functional models of a non-pipelined fetch-decode-execute iteration written in a low-to-mid-level C-family static language, delivering mid-level efficiency. Some ISS programs, such as QEMU, perform dynamic binary translation to allow software emulation to reach more usable speeds. This relatively complex methodology has not been widely adopted for system modeling.**

We demonstrate a fresh approach to ISS that achieves performance comparable to a fast dynamic binary translator by exploiting recent advances in just-in-time (*JIT*) compilers for dynamic languages, such as JavaScript and Lua, together with a specific programming idiom inspired by pipelined processor design. We believe that this approach is relatively accessible to system designers familiar with C-family functional simulator coding styles, and may be generally useful for fast modeling of complex SoC components.

Keywords-simulation; functional; dynamic language; scripting

I. Introduction

For system architectural exploration, early software development, and test generation, functional simulation is favored, making the trade-off of reduced timing information in favor of improved simulation speed. In contrast to the trend toward standardization of mid-level hardware modeling and simulation languages, such as System Verilog and SystemC, higher-level functional modeling remains largely a matter of ad-hoc simulators written in Java, C, or C++. The performance of these simulators is vastly superior to low-level cycle-accurate alternatives, but is becoming inadequate as focus shifts from uniprocessor modeling to many-core systems, where the increasing runtime involved in simulating complex systems becomes burdensome.

With the ubiquity of programmable components in computational SoCs, fast functional instruction-set simulation (*ISS*) is increasingly important. We demonstrate an approach to ISS, *TCOSim*, that achieves performance comparable to the best dynamic binary translating simulators by exploiting recent advances in just-in-time (*JIT*) compilers for dynamic languages, such as JavaScript and Lua, together with a specific programming idiom inspired by pipelined processor design. If effect, we code the simulator in the native style of the dynamic language, and let the JIT compiler optimize the code at runtime, rather than requiring mastery of a simulator-specific domain-specific language (*DSL*) describing the binary-code translation task.

This work performed under DARPA contract W91CRB-10-C-0129,
Dr. Robert Colwell, MTO Program Manager

II. Related Work

Interpretative instruction-set simulators (*ISS*) abound, with various levels of detail in the processor model. The *sim-fast* simulator from SimpleScalar LLC is a well-known representative of a straightforward C-based fetch-decode-execute iterative ISS interpreter, achieving more than 10 MIPS for basic functional instruction-set-architecture (*ISA*) simulation [1]. This naïve approach is a software analog of a non-pipelined hardware processor implementation, simple but slow.

Many variations of binary translation for ISS exist, translating instructions in the simulated ISA into code sequences run on the host processor, ([2] contains a survey). SyntSim [2] is representative of mainly *static-translating* simulators emitting C code as an intermediate representation that is compiled on the host machine, incurring a slowdown factor of around 7 to 11 on a SPECcpu2000 sample compared to native host execution. In addition to the typically complex build procedure, one problem with static approaches is that it is awkward to alter the tradeoff between simulation speed and detailed visibility or interaction for different parts of the simulated process. For example, a user may want to carefully examine a suspect region of new code after skipping past lengthy initialization and warm-up phases. SyntSim has both interpretive and compiled modes to address this problem.

QEMU [3] is a popular open-source *dynamic-binary-translating (DBT)* ISS that uses a per-ISA library of "micro-ops" (*TCG*) to define host-machine equivalents for each instruction, which are then dynamically assembled into executable basic blocks to get a reasonable balance between speed and hardware detail, for example, being able to recognize interrupt signals well enough to allow high-level system modeling. QEMU supports a number of simulated ISAs at good speed, though most of its community effort is devoted to x86 platform virtualization. Reported performance for Intel 386 platform emulation is a 15x slowdown on a recent host [4]. Support of a new ISA requires detailed understanding of both the simulation-target architecture and the QEMU/TCG micro-op intermediate representation, which is a barrier to adoption compared to use of a general-purpose programming language for instruction specification. In contrast to a general-purpose programming language, the effort invested in learning a simulator-specific domain-specific language (*DSL*) is not transferable to other tasks, and the supportive user community may be unhelpfully small.

QEMU is a useful and influential DBT simulator project, but it has some significant drawbacks for use in exploratory system design.

- The dominant community focus on x86 platform virtualization has made the whole system large, complex and non-portable.
- Grafting dynamic code generation onto a static language like C compels mastery of a DSL, TCG, peculiar to that project and various host-dependent tricks to generate, load and execute translated code, all of which is tediously orthogonal to the motivating system architectural questions.
- The simulator core is monolithic and has resisted efforts to use multi-core capabilities of host machines.
- The QEMU codebase is a mélange of various open- and closed-source licenses, which is a concern for possible impairment of intellectual-property rights embodied in new system components.

QEMU supports ISA simulation of the LatticeMico32 processor [5], a small in-order pipelined 32-bit RISC processor, optimized for FPGA implementations, for which Verilog HDL is available under a permissive open-source license. We are developing TCOSim as a high-performance alternative to the QEMU simulator for this processor ISA and experimental variants supporting multiprocessor SoC architectures.

III. Pipelined Processor Analogy

Like QEMU, TCOSim works by translating each source ISA instruction into a sequence of tiny operations, but in TCOSim this sequence consists of a moderately deep series of nested function calls in the Lua scripting language [6,7], rather than a sequence of statements, as is typical of a C-language simulator, or TCG declarations, as in QEMU. The stack of function calls for each instruction has a close analogy to the multiple (post-decode) stages of a classic RISC hardware pipeline: the first function call in that stack reads a register value or constant, the next reads the second register operand if necessary, the third may perform an ALU operation, etc. The first function call has a fat list of parameters that are successively reduced at each subsequent level. The function call pipeline has just a couple of defining ideas:

A. *Each stage in the instruction processor pipeline is translated to a function call that binds or consumes one or more parameters*

B. *Each stage function in the pipeline ends with a tail call to the next-stage function or to the start of a successor instruction's pipeline function stack.*

If the bottom function in the per-instruction stack calls another instruction, it's straightforward to string instructions together to execute an entire basic block at one invocation, greatly improving JIT compiler optimization opportunities. The bottom function returns a next-instruction reference; a conditional branch instruction may have a couple of next-instruction references held in a closure variable, for taken and not-taken branch conditions.

A given RISC ISA will have a handful of instruction formats, each with a fixed sequence of register or memory accesses. For each format, TCOSim has a "Lua string builder" function that writes the Lua code for a specific instruction instance of that format, for example, converting a read-register field value into a literal array indexing expression with a constant index to aid optimization. The hallmark of a dynamic language is the relative ease of converting this language string into a chunk of executable code via some kind of *eval* function; in Lua it's called *load*. The executable version of the per-instruction function stack, together with the parameter bindings, is stored as a Lua function closure, which can be executed as required. The basic associative array feature (*table*) in Lua makes caching of decoded instructions easy.

IV. Implementation Language

The TCOSim approach requires an implementation language with good support for a functional-language programming style. Specifically, its efficiency requires a good implementation of function closures and tail-call optimization (*TCO*), a concept largely popularized by the Scheme dialect of the Lisp language [8]. Functions make the instruction pipe look tidy, TCO makes it fast, and use of closure variables for storing parameter bindings and next-instruction linkage makes it flexible and dynamically tunable.

A. *Lua Language*

Lua is a small, multi-paradigm "scripting" language [6] that supports a functional programming style. Since its inception in 1993, the language designers have preserved Lua's small conceptual and resource footprint to enhance its embeddability and portability [7]. The Lua interpreter is written in ANSI C, with few further dependencies, and has a well-defined C-interface API, for use as an embedded scripting language in C-family programs and for access to external libraries. Lua has been adopted as an embedded scripting and template language by developers of TeX and Wikipedia and is likewise popular in the computer-gaming developer community.

B. *LuaJIT Language Implementation*

LuaJIT [9] is a high-performance implementation of Lua that contains both a target-specific interpreter written in assembly language and a well-engineered trace-based just-in-time (JIT) optimizing compiler producing native code for high-use code sequences. Target ISAs now include x86/x64, ARM, PowerPC, and MIPS. LuaJIT 2.0 performance compares very favorably to other dynamic-language implementations and closely approaches static-language compiler performance on many benchmarks. Also, LuaJIT has an excellent foreign-function interface (*FFI*) mechanism for C, simplifying interface coding and allowing direct access to primitive static data types, which can improve storage and hardware simulation efficiency compared to Lua's standard dynamic data typing.

C. *Advantages of a Dynamic Language and a Tracing JIT*

Tail-call optimization allows a series of function calls to be optimized into straight-line sequential code without superfluous branching instructions, which improves performance. Using dynamic code generation to build a stack of function calls for each instruction, combined with use of explicit “next instruction” references stored in instantiated function closure variables, allows the JIT compiler to minimize conditional branch instructions in “hot” execution traces. In contrast, statically compiled simulator code will tend to litter the instruction stream with conditional checks for even rare conditions and uncommonly activated feature support. Even unused features and checks will slow simulation speed and clutter the simulator code. Tail-call optimization allows better code organization by use of shared functions with little performance impact except for slowly growing register pressure for excessively deep sequences of function calls. Dynamic generation of simulator code sequences allows flexible tradeoffs between performance and visibility, for debugging or verification. Various levels of detail can be set selectively or locally, for example by setting memory-write watchpoints at per-basic-block granularity in new code routines. Visibility of PC updates, precision of arithmetic exceptions and latency of interrupt recognition can have similarly advantageous control of granularity.

V. Experimental Results

For the LatticeMico32 simple 32-bit RISC ISA, we have compared simulation speeds of the popular dynamic-binary-translating ISS QEMU to our alternative, TCOSim, using binaries created by a clang/LLVM version 3.1 compiler with a LatticeMico32 code-generator backend we have written. At the present early state of development, TCOSim simulation rates generally are close to that of QEMU, roughly in the 500-900 simulated MIPS range using one core of an Intel i7 2.2 GHz laptop host. We may improve performance with further optimization, but believe that we have captured most of the available single-thread performance. The TCOSim simulator runs an order of magnitude faster than a simple iterative static simulator, at perhaps double the code complexity. Our main interest is now to extend the base design to support parallelism on both the host and simulation target sides. We are using a mix of ad-hoc and standard benchmark codes, such as CoreMark [10], for validation and performance evaluation.

We observe that the LuaJIT tracing compiler is less likely to optimize short-runtime code sequences with its default parameter set. Usually this is a correct heuristic tuning, but sometimes we can see that this puts TCOSim at a disadvantage compared to QEMU, which performs its optimizations at the start of runtime without needing to acquire program trace data. Fig. 1 shows the lower performance of the initial interpreted mode of a memory-access intense vector-add simulation (two input and one output vectors) compared to the optimized code dynamically generated for longer-running sequences. We can put this in perspective by noting that the lower “short-sequence” LuaJIT interpretive-mode performance is on par with or better than simple iterative static simulators. The optimized compiled mode improves the performance by a factor of 15.

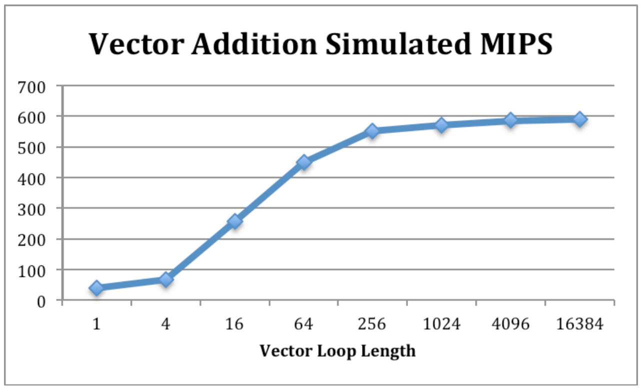

Figure 1. Tracing just-in-time (JIT) compiler optimization is more likely to be applied to more-used code sequences.

VI. Conclusions

We have demonstrated that a flexible dynamic scripting language, Lua, can be used to implement a relatively simple fast functional instruction-set simulator that rivals more complex DBT simulators. The availability of a remarkably effective trace-optimizing JIT compiler, LuaJIT, makes it possible to write dynamic code generators for ISA simulation with a straightforward programming idiom in a compact scripting language. We believe that our approach, inspired by the canonical RISC processor pipeline, will be readily understood by hardware practitioners, and may be useful for producing fast functional simulators for varied complex SoC components, whether programmable or fixed-function.

Acknowledgment

The authors would like to thank LuaJIT creator Mike Pall for his informative postings about LuaJIT optimization in general, and tail-call optimizations as a basis for efficient simulator chunk linkage in particular [11].

References

[1] D. Burger and T. M. Austin, “The Simplescalar Tool Set Version 2.0.” Technical Report 1342, Computer Sciences Department, University of Wisconsin. June 1997.

[2] M. Burtscher and I. Ganusov, “Automatic Synthesis of High-Speed Processor Simulators,” IEEE MICRO’04, pp. 55-66, December 2004.

[3] F. Bellard, “QEMU, a Fast and Portable Dynamic Translator,” M. B *USENIX 2005 Annual Technical Conference*, pp. 41-46

[4] C. Guillon, “Program Instrumentation with QEMU,” 1st International QEMU Users' Forum, DATE’11 Workshop

[5] LatticeMico32 website, http://www.latticesemi.com/mico32

[6] R. Ierusalimschy, “Programming with Multiple Paradigms in Lua”, Proc. 18th international conference on Functional and Constraint Logic Programming, pp. 1-12, WFLP'09, Springer-Verlag, Berlin, 2010

[7] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, Waldemar Celes, **Passing a Language through the Eye of a Needle:** How the embeddability of Lua impacted its design”, ACM Queue, Vol. 9 No. 5 – May 2011, http://queue.acm.org/issuedetail.cfm?issue=1978862

[8] Guy L. Steele, Jr., "Debunking the 'Expensive Procedure Call' Myth, or, Procedure Call Implementations Considered Harmful, or, Lambda: The Ultimate GOTO". MIT AI Lab. AI Lab Memo AIM-443. October 1977.

[9] LuaJIT website, http://luajit.org, c. Mike Pall 2005-2012

[10] CoreMark, http://www.coremark.org

[11] Mike Pall, http://lua-users.org/lists/lua-l/2011-02/msg00671.html