A Unified Processor Model for Compiler Verification and Simulation using ASM

Roland Lezuo, Andreas Krall

Institute of Computer Languages
Vienna University of Technology
Argentinierstr.8
A-1040 Wien Austria
rlezuo,andi@complang.tuwien.ac.at

Abstract. For safety critical embedded systems the correctness of the processor, toolchain and compiler is an important issue. Translation validation is one approach for compiler verification. A common semantic framework to represent source and target language is needed and Abstract State Machines (ASMs) are a well suited and established method. In this paper we present a method to show correctness of instruction selection by performing fully automated simulation proofs over symbolic execution traces of state transformations using an automated first-order theorem prover. We applied this approach to an industrial-strength compiler and created the ASM models in such a way that we are able to reuse them to create a cycle-accurate simulator. To achieve fast simulation we compile the ASM models to C++ and present the compilation scheme in this paper. Finally we present our preliminary results which indicate that a unified ASM model is sufficient for proving correct instruction selection and generating efficient cycle-accurate simulators.

1 Introduction

Todays safety critical systems often require application specific processors to fulfill the demanding performance and efficiency requirements. Correct behavior of the processor and the corresponding toolchain is an absolute requirement making formal specification and verification necessary. We are interested in using the same formal methods for compiler verification and simulation. Abstract State Machines are a well established method for specification and analysis of programming languages and systems providing a simple practical framework offering important features for industrial usage like decomposability and are readily understood [1].

Section 2 describes the generation of (first-order logic) proof scripts using symbolic execution of ASM models to perform translation validation [6] of instruction selection [3]. Section 3 describes our approach to generate an high-performance simulator using compilation to C++. Section 4 presents our preliminary results and concludes the paper.

2 Correctness of instruction selection

Zimmermann and Gaul [9] describe constructing correct compiler backends for DEC Alpha using ASMs. DEC Alpha has some nice properties making it very suitable for formal description [4]. The processor used in our project is a very long instruction word (VLIW) architecture with digital signal processor features and a non-interlocking pipeline. It supports wrap-around and saturation arithmetics, single instruction multiple data instructions, predicated execution, hardware loops and store/load with updates to the address register.

During instruction selection a (sub)tree of intermediate representation (IR) nodes is matched with a sequence of machine instructions, IR variables and temporaries (*operand*) are mapped to registers (*regmap*). We assume an infinite number of registers at this stage and allocate real registers later.

Such a translation is correct if the transformation described by the IR tree ($result_{tree}$) and the transformation induced by execution of the machine instructions ($result_{instr}$) is semantically equivalent. Semiformal this can be stated in first-order logic as: $\forall$ *operand : regmap(operand)* $\equiv$ *operand* $\Rightarrow$ *regmap(*$result_{tree}$*)* $\equiv$ $result_{instr}$. Some trees and instructions may however have side-effects (e.g. a modified memory cell) which are modeled as updates to ASM functions. For correctness the IR tree and the machine instructions must induce the same side-effects, semiformal this can be stated as $\forall$ $updates_{tree} \Rightarrow \exists$ $update_{instr}$ *:* $update_{tree}$ $\equiv$ $update_{instr}$ and vice versa.

To determine whether $result_{tree}$ and $result_{instr}$ are equivalent, ASM models (see next section for more details) using a common semantic vocabulary defining IR tree operations and machine instructions have been developed. The common semantic vocabulary is modeled as external functions in the ASM models. To generate a proof script the ASM execution engine logs an invocation of the external function f with arguments a returning result r as predicate $f(a, r)$. As concrete values for the operands are not known at instruction selection time the ASM has to be evaluated symbolically. The ASM execution engine performs the following steps when evaluation of a function f at location l results in *undef*. First create a new symbolic value s for f at l, then directly modify the definition of $f(l)$ so each evaluation of $f(l)$ returns s. The value *undef* is preserved when set explicitly, so evaluating $f(l)$ after a $f(l)$ *:=* *undef* will result in *undef* and not in a new symbol s. Finally log the creation of the new symbol as predicate $f(l, s)$.

The resulting log is a sequence of predicates stating facts about (symbolic) values of dynamic functions (e.g. contents of registers) and invoked external functions (i.e. the common semantic vocabulary). Given a set of axioms describing relations of the semantic vocabulary and the a priori known mapping of IR operands to registers a theorem prover can now show semantic equivalence of the state transformation described by the IR tree and the machine instruction induced state transformation.

3 Fast cycle-accurate simulation

Teich et al [8] have shown that ASM models can be used to generate a simulator for a processor. What they call bit-true arithmetic functions is equivalent to our common semantic vocabulary. Our simulator core is itself described in ASM notation (approx. 200 LOC). In contrast to [8] we are interested in efficient industrial-strength simulators. We initially tried the CoreASM execution engine [2] but simulating 50 CPU cycles took around 13 seconds. We considered compiling the CoreASM language to C++, but efficient compilation is difficult due to the dynamic type system (i.e. its List background). By adding type annotations to lists and restricting ourselves to a statically typed subset of the CoreASM language we were able to develop an efficient compiler.

Our compilation scheme preserves the static structure of the ASM model, and we follow the formal definition of ASM very closely. Each evaluated rule produces an update set, which is aggregated and composed as described in [2]. We support a (static) subset of the following CoreASM language elements: *seqblock, par, let, ifthenelse, :=, debuginfo, push, pop, forall, call, case, enum, derived, static, cons, nth, peek, tail, program, self* and lists with the restriction of all elements being of the same type (may be another list type). As we support the P *seq* Q rule we may need a (local) copy of the state to apply P's updates before evaluating Q. Such a copy however would be very expensive as the state contains huge functions (e.g. system main memory). That is why we introduced a so called *PseudoState* which only contains updates which should have been applied to the state already. When querying the *PseudoState* for a function f at location l a hashmap containing the pending updates is searched for $f(l)$ and if such an update is found its value is returned, if no such update can be found the global state is queried for $f(l)$.

It turned out that handling of the update sets is crucial to the performance of the simulator. As updates can not be stack allocated, and dynamic memory allocation using malloc/new would be too expensive a memory pool allocator is used. Memory management overhead is minimal as just the pointer to the next free memory cell needs to be incremented after each allocation. As soon as evaluation of the top level rules terminates (called a step in [2]) the resulting updates are applied to the global state and the memory pool is reset. This enables efficient simulation but large update sets are still troublesome for the simulator performance.

4 Preliminary results and Conclusion

The proof generation system is capable of compiling the ANSI C Rijndael reference implementation v2.2 resulting in approx. 1650 proof scripts. About 700 scripts are successfully proven as correct. Most of the other scripts can't be proven due to missing semantic description of the involved IR nodes and machine instructions. We currently are able to handle basic copying and converting instructions (e.g. register moves), basic arithmetic operations (e.g. addition),

memory access (load and store), memory access with pointer increment, but also conditional branches with symbolically evaluated conditions.

We are able to correctly simulate all fundamental features of the CPU like instruction fetch, bundling decoding, predicated execution, hardware loops and the pipeline. Due to missing ASM models of the instruction set only one test program is executed correctly. For this case our simulator is slightly better than the manually coded simulator provided by the hardware vendor. The compiled models execute approx. 3000 times faster compared to interpretation by the CoreASM execution engine.

We have presented an approach to translation validation using ASMs and theorem proving targeting a processor architecture with many difficult to model features. We then used the very same semantic models to generate a fast cycle-accurate simulator with performance comparable to a manually coded vendor provided simulator. To achieve efficient simulation we restrained ourselves to a static subset of the CoreASM language but nonetheless found creation of the models easy.

Ongoing work is creating the missing ASM models to show the verification method is suited to prove industrial strength programs and increase the number of applications which can be simulated.

Acknowledgment: This work is supported in part by the Austrian Research Promotion Agency (FFG) and by Catena DSP GmbH. We would also like to thank Laura Kovács for her valuable help with the vampire [7, 5] theorem prover.

References

1. Börger, E.: Abstract state machines: A method for high-level system design and analysis (2003)
2. Farahbod, R., Gervasi, V., Glässer, U.: CoreASM: An extensible ASM execution engine. In: Proc. of the 12th International Workshop on Abstract State Machines. pp. 153–165 (2005)
3. Fraser, C.W., Henry, R.R., Proebsting, T.A.: BURG: fast optimal instruction selection and tree parsing. ACM Sigplan Notices 27(4), 68–76 (1992)
4. Gaul, T.S.: An abstract state macine specification of the DEC-alpha processor family (1995), ftp://www.jair.org/groups/Ealgebras/alpha.pdf
5. Hoder, K., Kovács, L., Voronkov, A.: Interpolation and symbol elimination in vampire. In: Proc. of IJCAR. LNCS, vol. 6173, pp. 188–195 (2010)
6. Pnueli, A., Siegel, M., Singerman, F.: Translation validation. pp. 151–166. Springer (1998)
7. Riazanov, A., Voronkov, A.: The design and implementation of VAMPIRE. AI Commun. 15, 91–110 (Aug 2002)
8. Teich, J., Kutter, P.W., Weper, R.: Description and simulation of microprocessor instruction sets using ASMs. In: Proceedings of the International Workshop on Abstract State Machines, Theory and Applications. pp. 266–286. ASM '00, Springer-Verlag, London, UK (2000)
9. Zimmermann, W., Gaul, T.: On the construction of correct compiler back-ends: An ASM approach. Journal of Universal Computer Science 3, 504–567 (1997)