Argument for Complex Languages and Compilers

Steve Meyer

Tachyon Design Automation
Boston, MA
`smeyer@tdl.com`

Presented September 29, 2022 NEPLS Boston

Slides posted on my web page www.tdl.com/˜smeyer/cmplx-lang-nepls2022.pdf

Idea - use problem specific flow graph compilation to solve problems

- I think first use of this idea goes back to APL. APL compilers were implemented in the 1980s. I ran into this paper: Budd, Timothy, *Dataflow Analysis in APL*, Proceedings International Conference on APL, June 1986, pp. 22-28.
- Budd adds declarations (APL may require run time parsing) and checks for and applies flow graph optimization code generation to declared variables.
- Next step is to use flow graph compilation to speed up matrix formula compilation. Budd did not make this step.
- By complex languages I mean application languages where data structures have complicated mapping onto current CPUs. Usually this means multiple machine words.

Talk background experience with the CVC Verilog compiler

- CVC Verilog HDL Compiler is described in paper. *Fast Complex Language Compilers Can be Simple*, arXiv:1603.08059.
- Older version of CVC source available for download at www.tachyon-da.com.
- Verilog has changed to System Verilog (SV) that adds c++ classes and changes Verilog to an expression language.
- IEEE 1800 Standard Language Reference Manual is now 1400 pages.
- SV was designed by a committee that resulted in different simulators compiling different language interpretations. Problem was solved by the 3 large Electronic Design Automation (EDA) companies choosing only one company to work with.

Standardized SV parse output accessed with data base interface

- That company sells a SV parser source that provides a data base interface for Verilog tools (simulators) to access.
- All SV tools need to use the data base because the LRM has many places open to interpretation.
- One example is the generate feature for defining different size models from one source description. Generate requires a loop of multiple elaborations that are then relaxed to a fixed point "true" source.
- Problem is that the possible generated SV sources are not unique. The standard parser chooses one that one particular simulator happened to converge to.
- Verilog simulation underneath SV is the hard part so CVC is still used either from machine generated old style Verilog or with a SV wrapper put around it.

Some comments on CVC experience

- Although the goal was to use virtual instructions close to actual CPU instructions, in reality instructions important for simulation speed are complicated.
- For other complex language flow graph compilers, I would look for better algorithmic instructions not necessarily close to modern CPU instructions.
- An alternative that some people prefer is to generate assembly idiom c code instead of assembly and to possibly skip the interpreter entirely.
- Implementing Verilog simulation in hardware was tried but failed. Problem is that although hardware execution for most constructs was 10 times faster, modern CPU pipe lining and multiple issue negated the direct map to hardware speed advantage.

A CVC simulation speed problem area - wide constants

- I believe CVC is faster for normal CPU type HDL models, but has speed problems for unusual model types.
- One problem model type uses around 6500 bit serial registers from touch screen hardware models. The CVC simple constant table and lack of value tracking results in noncompetitive speed (25-50% slower).
- We thought that low level constant decomposition optimization and wide bit vector constant value tracking was not needed.
- This was wrong. Value tracking and flow graph analysis are needed for wide concatenates and part select value selection.

Verilog flow graphs do not generalize - jump across problem

- Main reason for not following FUD SSA rules is that if only one prong of an if or case has a suspend, the other prongs can't be invalidated because simulation will be too slow.
- Common pattern is one or different prong calling an API that may invoke a very long computation (circuit specific differential equation solving for example) that is rarely called.
- One difficulty in designing problem specific languages that can use flow graph compilation is that resulting language is too hard to elaborate (parse).
- Someone should design a V– with a separate preprocessor, programming language like context (no cross module references) and test pattern specification moved to a different language (maybe specman-e).
- Currently, Verilog complex language flow graph compilation is not a good area for work because EDA is caught in politics.

Flow graph compiler generated assembly output fastest code

- My conjecture is that for applications that need fastest possible execution speed, flow graph compiling to assembly is best.
- Mapping to assembly can fit in modern processor pipelines and maximize multi-issue IPC rates.
- Unlike old DEC Alpha CPUs, X86_64 processors do not allow outputted assembly control of branching and multi-issue. Designing more complex processors with more addressing modes and issue control is good because flow graph optimization can be used.
- Flow graphs often have very short instruction distances between branches that can be optimized by the compiler.

Some ideas for complex languages to compile

- Revisit APL. Maybe data type problem specific APL. Improve APL matrix and vector optimization.
- Use problem specific languages and flow graph compilers to calculate solved problems in contrast to Matlab type systems for exploring problems.
- Protein folding. Currently, program like Alphafold just reconstruct geometry from patterns discovered using crystallography. Defining a language for crystallographers that is compiled might be useful.
- RSA code breaking matrix multiplication step in prime factorization does not parallelize. A language and hand crafted flow graph compiler for this one problem might be an improvement.
- Searching for patterns in specific types of big data. Say census data.

Conclusion

- I am sure there are many more applications. My suggestion is problem specific hand crafted compilers that expand and optimize virtual instructions defined by experts in the particular application.
- History follows George Polya's heuristics (quasi-empirical math) and Peter Naur's dataology pattern. Previously unknown documents on Polya's method have recently become available.