Two-level Languages and Circuit Design and Synthesis

Walid Taha[*]
Rice University, Houston, TX, USA.
taha@rice.edu

The next two decades are anticipated to move digital circuit design from the million transistor level to the billion and trillion transistor levels. In addition to challenges that this goal poses at the physical level, fundamental computational complexity barriers suggest that common design and verification tasks can also become a bottleneck. Examples include placement and routing, as well as a host of design rule checking (DRC) techniques. Increase in circuit size will increase both on the time needed for DRC (from days to weeks or months) as well as the overall effort needed to produce a design likely to pass a DRC check. At the same time, increasing variability in implementation technologies and their characteristics will fuel the need for better methods to manage families of related circuits.

New programming language techniques recently developed to improve software design can provide a powerful tool for managing and checking families of related circuits. Program generation techniques in general, and two-level languages in particular, have been proposed and found to be useful for managing families of related software products. Static type checking in general, and dependent type systems in particular, have been proposed and found to be useful for early checking of a wide range of properties that would otherwise be expensive to check in generated programs. Our goal is to show that adapting these techniques to the specific needs of circuit design can lead to fundamental changes to the design process. In particular, it would allow the capture of significant design experience in the form of executable *and statically checkable* specifications for families of related circuits. Such specifications would be highly parameterized with respect to the specifics of the manufacturing technology, as well as the specifics of the problem being solved and the rest of the design. Comprehensive, manifest interfaces would allow fast, compositional checking of compatibility with the rest of the design.

Over the last two years we have made concrete advances toward this ambitious, long-term goal. Our first study showed that a standard type system for two-level languages can be systematically integrated with a type system for a resource-bounded language [5]. The result of such an integration, called a resource-aware programming (RAP) language [4], provides an expressive (non-resource bounded) language for writing generators of resource bounded computations. At the same type, a static type system is provided that checks that a generator can only generate well-formed, resource-bounded computations. Depending on the specific resources considered, such resource-bounded programs can be embedded software systems or hardware circuits.

A case study focusing on FFT showed that annotated versions of the basic Cooley-Tuckey recurrence can be executed as generators to produce high-quality circuits [2, 3]. In addition to confirming that this family of circuits specified by a generator closely resembling the textbook form of the standard recurrence, the experiment lead directly to two intriguing insight about FFT: First, unlike what the work on FFTW suggests, only a small number of domain-specific optimizations is needed to generate FFT circuits with the same arithmetic operation count as Split-radix or FFTW. Second, producing circuits that have counts identical to either Split-radix or FFTW requires *only* changing the definition of complex multiplication.

The RAP approach uses a purely functional language to describe hardware circuits, and so should be viewed as a direct descendant of Sheeran's family of hardware description languages. Focusing on two-level languages amounts to pursuing the insight that circuits are a strict subset of the generation language. Focusing on statically typed two-level languages reflects emphasis performing the checking at the level of a *family* of circuits rather than on an individual circuits. It is useful to note that this approach is complementary to model checking, which can perform more extensive, albeit more computationally intensive, checking on individual circuits.

Our emphasis on static checking discourages the transformation of circuits after they are generated. This contrasts with the transformational approach promoted by reFLect. Instead of first generating and then transforming, our approach focus on incorporating domain-specific optimizations directly in the generator. This approach can have a number of benefits. First, the designer can follow the methodology of abstract interpretation, widely used for program analysis, as a method for building optimizing generators that are correct by construction. The approach allows us to preserve the extensional nature of generated objects, and preserves strong reasoning principles. Second, avoiding the generation of numerous intermediate circuits can greatly improve the efficiency of the generation process.

Our recent and ongoing work focuses on the formal treatment of the connection between circuits and programs to allow the incorporation of various non-textual concepts into standard formal accounts of two-level languages [1]. Over the last year, we worked on building a prototype implementation to facilitate further work in this particular research direction. The prototype, called PreVIEW, implements the translations between the graphical and textual representations used in our formal studies, in addition to implementing basic circuit layout algorithms.

1. REFERENCES

[1] Stephan Ellner. PreVIEW: An untyped graphical calculus for resource-aware programming. Masters thesis, Rice University, 2004.

[2] Oleg Kiselyov, Kedar Swadi, and Walid Taha. A methodology for generating verified combinatorial circuits. *EMSOFT '04*, Pisa, Italy, 2004.

[3] Oleg Kiselyov and Walid Taha. Relating FFTW and split radix. *ICESS '04*, Hangzhou, China, 2004.

[4] Walid Taha. Resource-aware programming. *ICESS '04*, Hangzhou, China, 2004. Invited Paper.

[5] Walid Taha, Stephan Ellner, and Hongwei Xi. Generating Imperative, Heap-Bounded Programs in a Functional Setting. *EMSOFT'03*, Philadelphia, PA, October 2003.

[*]Joint work with Stephan Ellner, Oleg Kiselyov, and Gregory Malecha. Funded by the National Science Foundation, the Texas Advanced Technology Program, National Instruments, and a grant from Rice University.