Hardware Synthesis from Functional Embedded Domain-Specific Languages*

A Case Study in Regular Expression Compilation

Ian Graves[1], Adam Procter[1], William L. Harrison[1], Michela Becchi[2], and Gerard Allwein[3]

[1] Department of CS, University of Missouri, Columbia, Missouri, USA
[2] Department of ECE, University of Missouri, Columbia, Missouri, USA
[3] US Naval Research Laboratory, Code 5543, Washington, DC, USA

Abstract. Although FPGAs have the potential to bring software-like flexibility and agility to the hardware world, designing for FPGAs remains a difficult task divorced from standard software engineering norms. A better programming flow would go far towards realizing the potential of widely deployed, programmable hardware. We propose a general methodology based on domain specific languages embedded in the functional language Haskell to bridge the gap between high level abstractions that support programmer productivity and the need for high performance in FPGA circuit implementations. We illustrate this methodology with a framework for regular expression to hardware compilers, written in Haskell, that supports high programmer productivity while producing circuits whose performance matches and, indeed, exceeds that of a state of the art, hand-optimized VHDL-based tool. For example, after applying a novel optimization pass, throughput increased an average of 28.3% over the state of the art tool for one set of benchmarks. All code discussed in the paper is available online [**?**].

1 Introduction

FPGAs are notably difficult to program and this has motivated research into high-level synthesis (HLS) from high level programming languages and, in particular, from domain-specific languages [**?**]. This language-based approach is attractive because of its potential to make hardware engineering more like software engineering with its support for modularity, reuse, and abstraction, and thereby create a wider group of developers for programmable hardware. This paper describes a methodology for deriving performant hardware implementations directly from high-level functional embedded domain-specific languages (EDSL).

This work makes the following contributions. We present ReWire [**?**], a subset of the Haskell functional language as a compiler target for compiling

* This research was supported by the Office of the Assistant Secretary of Defense for Research and Engineering, the U.S. Department of Education under GAANN grant number P200A100053, NSF CAREER Award 00017806, and NSF award CNS-1319748.

domain-specific languages to FPGAs. We show that ReWire can be effectively used as a compiler target because it supports the compilation of large input programs (over 100K LOC) and can generate competitively fast hardware implementations versus state of the art, domain-specific tools.

These contributions comprise a methodology supporting the "three P's" [**?**] for programming reconfigurable hardware: productivity, performance and portability. DSLs address the first two P's directly because domain specialization supports programmer productivity and, furthermore, allows aggressive optimization of domain-specific idioms. Portability is achieved by using ReWire, a retargetable language for specifying hardware devices.

New language constructs raise issues with respect to performance. Is there a performance price to be paid and, if so, is the increased expressiveness worth it? Does the increased expressiveness enable better performance and programmer productivity? In light of these questions, we evaluate our methodology via two case studies. The case studies presented here consider a purely functional framework for REHC construction, called RexHacc (for "Regular EXpression HArdware compiler-compiler"). RexHacc is an EDSL-structured compiler-compiler, implemented in Haskell, for Perl-compatible regular expressions (PCRE) similar to those seen in popular intrusion detection systems (e.g., Snort [**?**]).

Overview of Methodology. The methodology factors the problem of HLS into a series of translations between EDSLs. An EDSL is a domain-specific language that is defined as a collection of constructs within an existing high level language. The methodology is illustrated in the inset figure. A problem domain can be realized as a DSL embedded in Haskell. DSL cross-compilers targeting ReWire enable synthesis onto an FPGA via the ReWire compiler. Sec. 2 presents a more in-depth discussion of our methodology.

Fig. 1. FP Methodology for HLS

The case studies involve regular expression to hardware compilation (see Fig. 2) in which we generate artifacts that perform as well as and often better than state of the art approaches. The case studies reported here consider the problem domain of regular expression to hardware compilers (REHC) [**?**]. Following Fig. 1, we developed a reusable and modular framework for REHC called *RexHacc* and demonstrated that circuits produced with it meet or exceed the performance of state-of-the-art REHC.

The RexHacc Framework. We performed an experiment in which we compared RexHacc to the performance of the state-of-the-art REHC of Becchi and Crowley [**?**] (henceforth `reg2vhdl`) against its own benchmarks. The goal is to demonstrate both the productivity gain and high performance achievable via our methodology in the construction and testing of compilers generated by RexHacc. The presentation here is deliberately high-level. We suppress the definitions of functions and data types; the code is online [**?**].

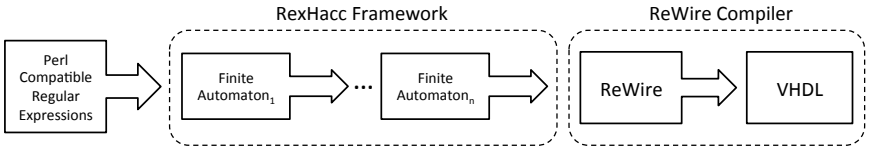

Fig. 2. Combining the ease of use of traditional EDSLs with the power and run-time performance of a virtualized language.

The entry point for RexHacc is the function `rexhacc` with Haskell type:

```
rexhacc :: (NFA a -> NFA a) -> RegEx a -> ReWire
```

The declaration form "`::`" is pronounced "has type". The function `rexhacc` takes two inputs, an optimization function (of type `NFA a -> NFA a`) as well as a regular expression (of type `RegEx a`). The type `NFA a` (resp., `RegEx a`) represents non-deterministic finite automata (resp., regular expressions) over an alphabet of type `a`. A regular expression compiler is generated with RexHacc by applying the top-level `rexhacc` function to an optimization pass, `opt`:

```
                compiler :: RegEx a -> ReWire
(‡)             compiler = rexhacc opt
                      where opt = (o1 . ··· . on)
```

Each o_i is an optimization pass of functional type `NFA a -> NFA a`, all of which are composed using Haskell's function composition operator (i.e., the infix "`.`") into a single pass. This composition corresponds to the middle box in Fig. 2 and each o_i is a phase inside that box. The generated `compiler` takes a regular expression over an alphabet of type `a` and converts it into an `NFA a`, which is then fed to the optimization pass `opt`. The optimization pass produces an `NFA a` from which ReWire code is generated. The ReWire output from this compiler can either be translated into VHDL by the ReWire compiler or executed as software in any standard Haskell environment.

Summary of Case Study Results. Secs. 4 and 5 each describe the definition of an REHC in the RexHacc framework. Each case study was tested against `reg2vhdl` using existing test suites [**?**] with respect to standard metrics for circuit size, clock speed and throughput (see Fig. 3). The first case study (Sec. 4) implements the same optimization passes as `reg2vhdl`, and it was clear that this compiler generally matched or exceeded the performance of the hand-optimized compiler `reg2vhdl` with a tiny increase in circuit size. It was observed that one of the benchmarks (`tcp25`) seemed to be particularly challenging for both the first case study compiler and `reg2vhdl` with respect to throughput. This observation motivated the second case study (Sec. 5), which improves on the first with an (apparently novel) optimization pass that results in better performance than `reg2vhdl` on the `tcp25` benchmark.

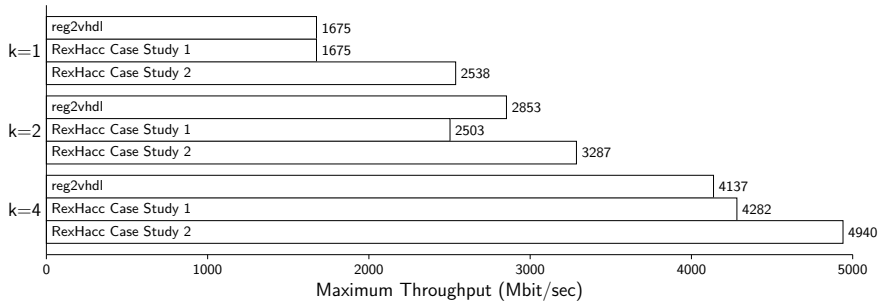

Fig. 3. Maximum throughput for the `tcp25` benchmark, comparing **`reg2vhdl`** and the RexHacc case study compilers (Secs. 4 and 5). Parameter `k` indicates stride length (Sec. 4). Case study 2 shows an average of 28.3% throughput increase over **`reg2vhdl`**.

2 A Methodology for Synthesis from Functional EDSLs

Synthesis from pure functional languages (e.g., Haskell, `www.haskell.org`) is appealing because combinational hardware is functional in nature, functional languages have powerful features supporting programmer productivity (e.g., modularity, expressive data types, static type inference, etc.), and the absence of side effects (e.g., destructive update) simplifies synthesis. But general purpose functional languages also contain a number of features that cannot be represented in hardware (e.g., general recursion and garbage collection) and this makes HLS directly from existing functional languages more challenging.

ReWire [**?**] is a proper *sublanguage* of Haskell—i.e., any ReWire program is a Haskell program, but not all Haskell programs are ReWire programs. ReWire programs, in contrast with general purpose functional languages like Haskell, are always synthesizable to hardware. ReWire restricts Haskell by disallowing the use of higher-order functions and general recursion at runtime (though techniques like partial evaluation may enable their use at compile time). RexHacc uses the ReWire hardware compiler as a back-end for producing VHDL implementations.

Front End. The RexHacc compilation process begins with a collection of regular expressions written in Perl-compatible regular expression (PCRE) syntax. We use the parser combinator library Parsec in Haskell to parse the regular expressions in the source file. The regular expression is converted to the `NFA` type via a textbook translation of regular expressions to NFAs [**?**]. The resulting `NFA` is passed to the optimization portion of the compilation chain.

Simulating Circuits in Haskell. Because ReWire is a sublanguage of Haskell, we can execute ReWire code as software in any Haskell environment with a test harness for executing reactive resumptions. The implementation of `rexhacc`was tested and debugged using a test harness in Haskell which is included in the code base [**?**].

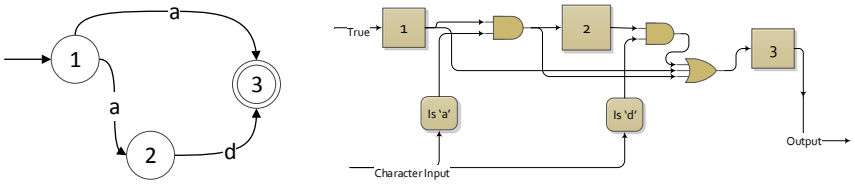

Fig. 4. An NFA and its corresponding Sidhu and Prasanna-style implementation.

3 Related Work

The conversion of sets of regular expressions into NFAs is a well-known procedure [**?**]. Sidhu and Prasanna [**?**] have proposed an efficient FPGA implementation of NFAs. Their solution is based on the one-hot encoding scheme; the use of an NFA representation avoids the $O(2^n)$ space complexity that is characteristic of DFA (deterministic finite automata) representations, typically adopted in memory-based regular expression matching implementations [**?**,**?**,**?**,**?**]. Subsequent efforts on FPGA [**?**,**?**,**?**,**?**] have refined Sidhu and Prasanna's implementation and achieved gigabit/sec processing throughputs on real-world pattern sets.

There are a number of efforts to apply ideas and techniques from functional programming to hardware design and synthesis. Arvind [**?**] describes the Bluespec synthesis language as "a relatively simple DSL (GAAs *[Guarded Atomic Actions]* and modules) with a fully functioning Haskell-like meta programming layer on top." The methodology advocated here employs metaprogramming as well, in that ReWire programs (which are also Haskell programs) are ultimately produced by the `rexhacc` function. Within the Haskell community, perhaps the most well known system for hardware synthesis is Lava [**?**]. Lava is a domain-specific language for hardware specification embedded in Haskell. Primitives in Lava are essentially structural and specify circuits at the level of signals. ReWire, by contrast, compiles a subset of Haskell itself to hardware circuits, and relies on an abstract set of behavioral primitives. The primary motivation for developing ReWire is as a vehicle for the design, implementation, and formal verification of high assurance hardware.

Cλash [**?**], is a compiler for a subset of Haskell to VHDL. Like ReWire, Cλash uses Haskell itself as a source language. Cλash requires some limits be placed on the kinds of algebraic data types used as well as the basic operating types. ForSyDe is a platform to compile models of hardware written in Haskell to circuitry [**?**]. This paper demonstrates that the ReWire compiler works at scale as the generated ReWire programs are on the order of 100K LOC. Great care was taken in the design of ReWire so that it possesses a rigorous denotational semantics to support formal verification while maintaining synthesizability for all of its programs [**?**].

The Delite DSL compiler framework [**?**] seeks to address the "three P's" with respect to implementing software on parallel, heterogeneous systems. Delite

addresses portability (i.e., retargetability of DSL compilers to a broad range of parallel hardware) through *language virtualization*. ReWire is also a virtualized DSL in that it has a separate compiler backend for producing FPGA-based implementations while reusing large parts of its host language's infrastructure—including Haskell's type system, front end, etc. In George, et al., [**?**], the Delite framework is adapted to the generation of hardware from DSLs, specifically the hardware acceleration of kernels in a heterogeneous setting.

4 Case Study 1: Matching State of the Art

We undertake the construction of a tool equivalent in functionality to the state of the art [**?**] (`reg2vhdl`) and to examine the feasibility of duplicating this functionality with our approach. The purpose of this case study is to demonstrate the *ease* with which such a tool can be constructed. The optimizations were chosen to match those of Becchi and Crowley [**?**] and include *head zipping*, *striding*, *alphabet compression*, and *epsilon elimination*. These results indicate that the `rexhacc`-based compiler compares favorably to and often surpasses `reg2vhdl` where throughput is concerned, and area utilization is similarly competitive. Each optimization phase was implemented in a few dozen lines of Haskell code; this is a rough indication that the amount of programmer effort required is small.

- *Head zipping.* Head zipping is a transformation that merges outbound transitions from a state that have the same transition labels. Nodes with more than one inbound transition are not head zipped because this would result in a non-equivalent NFA. Head zipping is performed by merging the destination nodes of the matching transitions into one node that includes all of the outbound transitions from the merged nodes.
- *Striding.* Striding is an optimization pass that doubles the number of characters an NFA matches at each transition. Striding traverses the graph's edges and looking two transitions ahead from each state, converts two-transition sequences to a single transition consuming two characters.
- *Alphabet compression.* Alphabet compression is a technique that increases sharing of logic by exploiting the identical treatment of different characters by an NFA. If two characters always result in the same transitions between all states, then these characters are compressed into one character class.
- *Epsilon elimination.* Eliminating ϵ-transitions reduces the complexity and size of NFAs and simplifies code generation. NFAs with ϵ-transitions allow state transitions without consuming input. States connected to an NFA solely by ϵ-transitions can be eliminated. Eliminating unnecessary states reduces the number of flip flops required to implement the NFA on an FPGA. A textbook ϵ-elimination algorithm is used [**?**].

Experiments and Evaluation. To test the performance of RexHacc, we selected three benchmark sets of regular expressions from the literature [**?**,**?**].

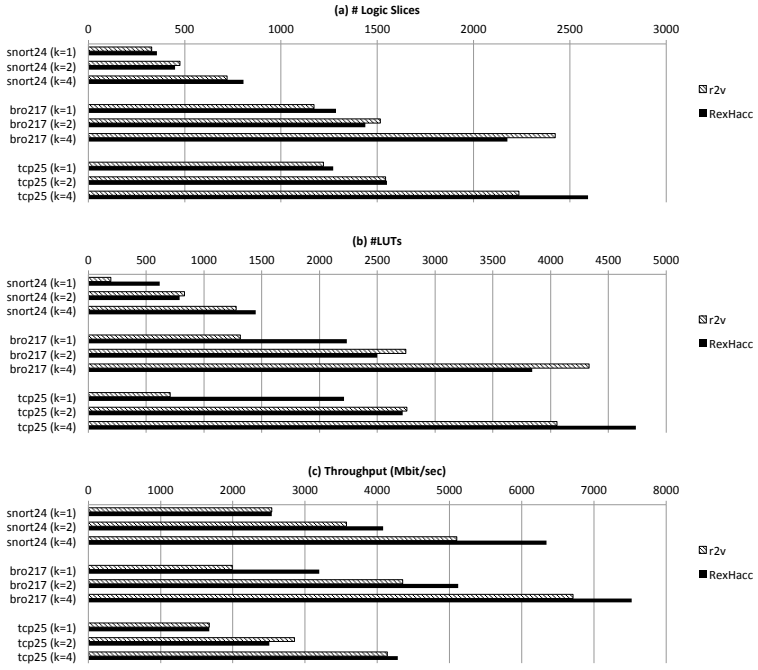

Fig. 5. Performance comparisons of RexHacc to `reg2vhdl` tool (here, "r2v").

`Snort24` is a set of 24 regular expressions drawn from the Snort network intrusion detection system [**?**]. `Tcp25` is a set of 79 regular expressions designed to match malicious SMTP traffic, also drawn from the Snort NIDS. `Bro217` is a set of 217 regular expressions drawn from the Bro NIDS [**?**]. Matchers for each of these benchmarks were generated using `reg2vhdl`, as well as RexHacc. Each benchmark was tested at stride lengths $k = 1$, $k = 2$, and $k = 4$, producing circuits that consume input streams at one, two, and four bytes per clock cycle. The resulting VHDL was then synthesized using Xilinx's XST synthesis tool for the Xilinx Spartan-3E X3CS500E FPGA, speed grade -4. The synthesis tools are optimized for speed. The frequencies that we list are synthesis estimates.

Fig. 5 compares the resulting circuits in terms of three performance metrics: (a) logic slice utilization, (b) LUT utilization, and (c) maximum throughput as measured in megabits per second. (Flip flop utilization was extremely close between the two tools and thus is not shown.) RexHacc compares favorably with `reg2vhdl` on virtually all fronts.

Throughput. RexHacc matches or exceeds `reg2vhdl`'s total throughput for all but one of the nine benchmarks. In the best case (benchmark `bro217`, $k = 1$) throughput is around 60% higher. In the worst case (benchmark `tcp25`, $k = 2$) throughput is around 13% lower. Both tools, in all cases, are capable of

processing input at a rate of more than 1 Gbit/sec. In the best case, RexHacc is capable of handling input rates up to 7.5 Gbit/sec on a Xilinx Spartan-3E FPGA at a relatively low clock rate. Tests on a Xilinx 7-series platform (not presented here, but available online [**?**]) indicate that throughputs of up to 25 Gbit/sec are achievable with a more modern FPGA.

Logic utilization. With the exception of the single-strided ($k = 1$) benchmarks, LUT utilization for RexHacc-generated circuits ranged from 88% to 116% of their `reg2vhdl` counterparts. In the specific case where $k = 1$, RexHacc tends to produce circuits with higher LUT counts (up to 219% higher), suggesting that the combinational next-state logic produced by the RexHacc code generator is more complicated for these circuits. For all benchmarks, flip flop utilization for RexHacc was close to, but slightly higher than, the results generated by `reg2vhdl`. This is not surprising since each state in the NFA is represented by a single flip flop, and both tools tend to generate similar numbers of NFA states. RexHacc, however, pays a small penalty here, because it generates output signals synchronously, storing them in flip flops, while `reg2vhdl` does not. Please note, however, that the choice of synchronous outputs rather than asynchronous ones is optional in the most recent version of ReWire.

The results exhibited here suggest that the case study compiler is competitive with the state of the art. The extra flexibility of the modular, purely functional design does not come at a prohibitive cost in terms of circuit size, and indeed brings substantial benefits with respect to throughput.

5 Case Study 2: Surpassing State of the Art

In this case study, we demonstrate the *agility* of the RexHacc approach by identifying an opportunity for an optimization, and rapidly implementing that optimization as a compiler phase in RexHacc. The modular nature of RexHacc made it easy both to identify a key performance bottleneck, and to implement a new optimization pass to address it.

Identifying the bottleneck. While conducting the experiments of Sec. 5, we noticed that one of the benchmarks, `tcp25`, stood out for its relatively low maximum throughput when processed by RexHacc as well as by `reg2vhdl`. While striding enabled our compiler to produce circuits with maximum throughput in excess of 6 Gbit/sec for `snort24` and `bro217`, maximum throughput for `tcp25` just barely exceeded 4 Gbit/sec. The throughput advantage over `reg2vhdl` observed for `snort24` and `bro217` was essentially nonexistent for `tcp25`.

To explore the reasons for this, we instrumented our compiler pipeline by using the Haskell Functional Graph Library's built-in support for generating graph visualizations via GraphViz (www.graphviz.org). We observed that the `tcp25` NFA exhibited a structural feature that was not present in the `snort24` and `bro217` NFAs. Specifically, the `tcp25` NFA contained one state that had

a large number of inbound transitions. A simplified example of this problem is exhibited in Fig. 6 (left), where state 9 has eight inbound transitions. A large number of inbound transitions emerges when the source regular expression contains a long chain of choice operators. This pattern is not uncommon in packet inspection rulesets (e.g., consider a long chain of alternative filenames followed by the common suffix ".exe").

In the circuit implementation the inbound transitions translate to a large fan-in of signals that must be ORed together to determine whether to activate that state. As the size of this fan-in grows large, the combinational logic involved begins to dominate the critical path of the circuit. The result is a sharp reduction in maximum operating clock frequency, and therefore throughput. This suggested an opportunity for optimization: namely, to transform the NFA in such a way as to reduce the number of inbound transitions to heavily-loaded states.

State Splitting Optimization. To address the performance bottleneck, we extended the compiler of Sec. 4 with an optimization called *state splitting*. Suppose we have in our NFA a state s with inbound transitions $e_1, \cdots, e_n$, and assume without loss of generality that s has no self-loops. Observe that we can produce an *equivalent* NFA by "splitting" s in two: that is, introducing a new state (call it s'), and reassigning half of the inbound transitions (say, $e_1, \cdots, e_{\lceil n/2 \rceil}$) to s' instead of s. State splitting works by applying this transformation to each node whose indegree exceeds a certain fixed threshold t. Fig. 6 (right) illustrates the results of applying state splitting to the NFA for $t = 2$. N.b., the maximum indegree has been reduced from 8 to 2 in this example.

The reader may note that this optimization may have the effect of *increasing* the number of inbound transitions for successor states of split nodes. This is generally not a problem for two reasons: first, as long as state splitting succeeds in reducing the *maximum* indegree, it is likely to pay off even if some states see their number of inbound transitions increased. Second, state splitting may be iterated; if the splitting of state s_1 results in state s_2 exceeding the split threshold, s_2 itself may be split.

The full code for the state-splitting optimization, consisting of 17 lines of code, is given as the `splitStates` function in the code base [**?**]. We can insert the state-splitting into the optimization pipeline simply by adding an extra phase to the `rexhacc` call; this is an instance of (‡) from Sec. 1:

6 Conclusions and Future Work

This research is a substantial case study utilizing the ReWire compiler at scale. ReWire is a subset of Haskell limited in expressive power to ensure the synthesizability of every ReWire program. There is a potential drawback to such restrictions: it excludes many powerful functional programming idioms. In spite of this potential drawback, we demonstrate that ReWire maintains sufficient expressiveness to support the design and implementation of high level DSLs for specifying

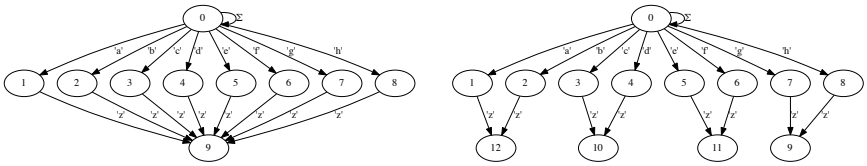

Fig. 6. NFA for $(a|b|c|d|e|f|g|h)z$, before state splitting (left) and after (right).

fast hardware accelerators. Future work aims to improve the resource usage of ReWire-generated devices by optimizing ReWire's code generation stages.

The methodology leverages the intrinsic power of Haskell and functional programming. RexHacc is modular and customizable in the sense that optimization passes can be easily added and removed. Because the ordering of passes is exposed as function composition in Haskell, experimentation with optimization ordering is enabled. A RexHacc-generated compiler can be instrumented in a straightforward manner as we did with GraphViz and take advantage of existing external Haskell tools.

The flexibility of the RexHacc framework derives from the cross-compilation to ReWire and the ability of ReWire to generate VHDL synthesizable to efficient circuits. The methodology we have introduced lowers the barrier to entry for reconfigurable computing for functional programmers. At the same time, it provides an opportunity for hardware designers to leverage the power of the functional paradigm to improve productivity. The choice of a purely functional language does not come at a performance cost: our benchmarking demonstrates that we match or exceed the performance of a state-of-the-art hand-tuned compiler for a number of real-world tests.

The two research directions we are pursuing have to do with increasing the expressiveness of the type system to support metaprogramming and hardware security. The current methodology is based on metaprogramming (i.e., ReWire/Haskell programs are generated by Haskell programs) and there are type systems for staged programming (e.g., MetaML [**?**]) that we believe will improve programmer productivity further while automatically enforcing type safety. We developed a type system for enforcing fault isolation on ReWire [**?**] and we are currently extending to information flow security.

7 Acknowledgments

The authors would like to thank Jason Agron of Intel Corporation and David Andrews of the University of Arkansas for their helpful feedback.

References

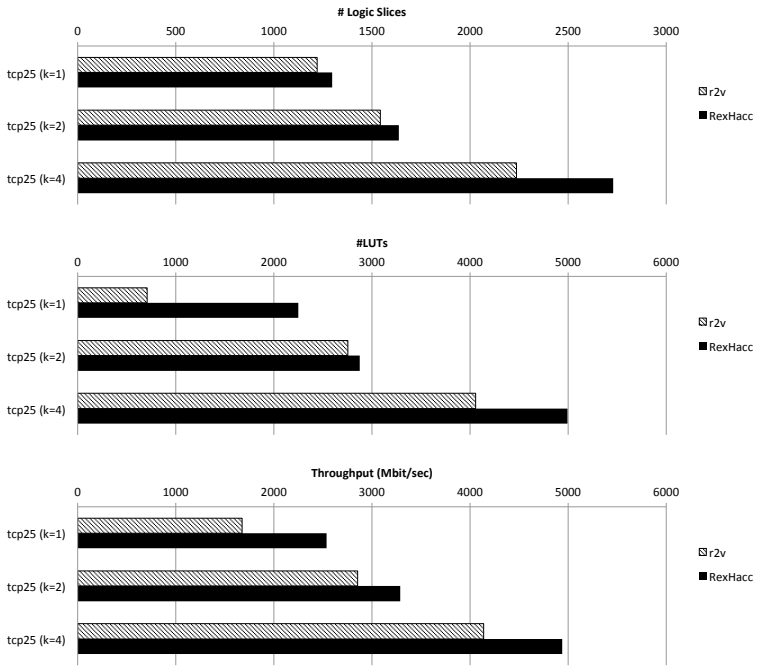

Fig. 7. Comparisons of RexHacc with state splitting enabled to `reg2vhdl` (here, "r2v") tool.