Analyzing the Static Elaboration of Parameterized Hardware Descriptions

Michael Pellauer
Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Lab

ABSTRACT

High-level synthesis languages allow hardware designers to describe modules at higher levels of abstraction and use techniques such as polymorphism to increase source-code generality and reusability. Abstract descriptions are concretized into specific hardware structures by the compiler during a phase known as *static elaboration*. Experience has shown that as hardware designers created increasingly generalized designs, static elaboration can become the most expensive compilation phase. We demonstrate that for a generalized PowerPC processor the Bluespec compiler spends 70% of total compilation time in static elaboration.

Our contribution is to reduce the execution time of the elaboration phase by applying techniques which have been previously developed to analyze and optimize software programs to the elaboration of hardware. We discuss the similarities between static elaboration and traditional software interpretation. We demonstrate that one particular technique, Higher-Order Abstract Syntax, resulted in an average improvement in elaboration time of 45% on five of six designs in our benchmark suite.

1. INTRODUCTION

It is generally accepted that hardware designs have increased in size and complexity by an order of magnitude since 1995, and are expected to increase by another order of magnitude by 2010. To manage this complexity designers rely on reusing existing blocks of intellectual property (IP reuse). A more generalized hardware description has greater potential to be reused in future situations.

In software, developers can use techniques such as polymorphism to achieve generalized, reusable descriptions. This generality is resolved at runtime through established techniques such as dynamic dispatch. In hardware design the output of the compiler is a concrete structural hardware description which may be used as input for a synthesis tool. Thus all abstract modules must be concretized during compilation, during a compiler phase known as *static elaboration*.

Current register-transfer level (RTL) languages such as Verilog allow designers to write modules which accept simple numeric parameters such as data width, latency, or number of read/write ports. In this situation the static elaboration step is straightforward - the hardware compiler syntactically duplicates circuit descriptions and alters the names of calls as appropriate. However, recent research in high-level synthesis has pointed to the advantages of a more powerful notion of elaboration [1]. Bluespec is a high-level hardware design language which uses the parametric polymorphism and generalized typeclass system of the functional programming language Haskell [4].

During static elaboration the Bluespec compiler:

- Instantiates modules with specific parameter values
- Applies (possibly-recursive) functions
- Resolves polymorphism and dispatching
- Executes and optimizes statically-determined control-flow
- Optimizes don't-care values as possible

In fact, a Bluespec description is not a single hardware design, but rather a program that can be executed to *generate* designs. Static elaboration in the Bluespec compiler can thus be viewed as the execution of this program by a software interpreter on a specific set of input parameters. This allows the designer to describe more generalized modules, such as sorting buffers parameterized by a sort function, or a Fast Fourrier Transform applicable to both signed and unsigned integers.

As we will demonstrate, for designs which make heavy use of generality the static elaboration phase is the most expensive phase of the compiler, consuming more than 70% of total compilation time. Amdahl's Law thus dictates that we should focus our optimization effort on this phase of the compiler.

As static elaboration becomes closer to traditional software execution, it is natural to explore whether methods of program analysis and optimization developed for software can be applied to optimize the elaboration process.

The specific technique we have chosen is Higher-Order Abstract Syntax (HOAS), developed by Pfenning and Elliot [5]. In this technique traditional abstract syntax trees (ASTs) are combined with lambda calculus constructs to improve the correctness and efficiency of evaluation. We will show that this technique is also applicable to hardware elaboration, as it resulted in an average improvement in elaboration time of 45% on five of six benchmark designs. However for the sixth testcase, HOAS resulted in an exponential increase in elaboration time. We analyze this design in detail and suggest possible resolutions of this problem.

1.1 Paper Organization

In Section 2 we review the Guarded Atomic Action model of hardware description and discuss alternative approaches to hardware elaboration. Then in Section 3 we describe our experimental methodology including our modeling language, reference interpreter, and benchmark suite. Section 4 contains details of the HOAS transformation and explores experimental results, including analysis problem it created with one of the benchmark designs. We conclude and present future work in Section 5.

2. BACKGROUND

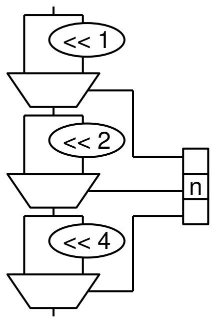
In this section we discuss static elaboration, and motivate its benefits using a shifter circuit. We explore the cost this generality can incur on compilation times and demonstrate that there is a need to improve the elaboration process. Finally, we review alternative approaches to static elaboration, and discuss the applicability of our approach to these projects.

2.1 Static Elaboration of Parameterized Hardware

As the complexity of hardware designs has increased, the trend in hardware description languages has been to increase the level of abstraction, first from schematic capture to RTL languages such as VHDL or Verilog, and more recently to high-level synthesis languages and techniques. The Guarded Atomic Action (GAA) model of hardware synthesis [4] is an abstraction developed to give the designer a semantic model to reason about the concurrent behavior of a complex hardware system. Bluespec SystemVerilog is a high-level synthesis language that is based around the GAA model. In this paper, rather than working with the full Bluespec language we will work with a restricted subset which we have named FSpec.

FSpec resembles the intermediate syntax used by the Bluespec compiler, and retains important features such as modularity and polymorphism. Details of the abstract syntax are given in the Appendix. FSpec allows the user to define modules in GAA style with rules (internal behavior) and methods (external interfaces). It includes support for functions, recursion, higher-order datatypes such as arrays, and parameterization both in the value domain (via module parameters) and the type domain (parametric polymorphism).

To motivate the need for parameterized descriptions, consider the case of a Shifter module which can left-shift a bit vector *v* by *n*, where *n*=0..7. Such a circuit can be implemented using $log_2(8)$=3 multiplexors and shifters:



So if *n* is 5, or 3'b101, *v* will be shifted by 1, then this value will be again shifted 4, for a total of 5. Note that the width of *v* itself is unimportant to this circuit (although it will generally not be less than 8 bits). Thus when we describe our shifter in FSpec we can make it parametrically polymorphic on the width of *v*:

```
module shifter3<v_width> {
    method v_width shift(v_width v,
                         bit[3] n) {
      v_width v1, v2, v3;
      v2 = (n[0] == 0) ? v : v << 1;
      v3 = (n[1] == 0) ? v2 : v2 << 2;
      v4 = (n[2] == 0) ? v3 : v3 << 4;
      return v4;
    }
}
```

During static elaboration the user will specify a concrete value for v_width and the elaborator will add the appropriate number of input/output ports and instantiate shifters and mux circuits of that width.

We can also conceive of generalizing this circuit to any *n*: add a mux and a shifter for each bit in the representation of *n*. This generalized shifter can be described in FSpec using a while-loop:

```
module shifterN<v_width, n_width> {
    method v_width shift(v_width v,
                         n_width n) {
        v_width tmpv;
        int k = 0;
        while (k < n_width) {
           tmpv = (n[k] == 0) ?
                       tmpv
                     : tmpv << (2^k);
           k++;
        }
        return tmpv;
    }
}
```

What is the meaning of the while-loop and the index variable k in the above code? It does not directly correspond to hardware in that there is no single circuit implementing the loop, as there is for the << operator. Instead, the loop is executed by the elaborator. Each execution of the loop will result in the creation of one mux/shifter pair.

The variable k is entirely statically determined, and thus does not survive the elaboration process. This means that the k++ and 2^k expressions are statically calculated – no circuitry is necessary. This is in contrast to the variable v, which is an input port to the module, and thus whose values are unknown at elaboration time. If v were hard-wired to a certain value the elaborator could also execute the left-shift.

```
evalStmt :: FStmt -> Result ()
...
evalStmt(FWhile cond body) =
 do
     b <- evalExpr cond
     case b of
       Static True -> do
           evalStmts body
           evalStmt (FWhile cond body)
       Static False -> return ()
       Dynamic -> error
...
evalExpr :: FExpr -> Result FExpr
...
evalExpr (FTri b tn el) =
 do
    b' <- evalExpr b
    --type-checking guarantees a bool
    case b' of
        (Static True) -> evalExpr tn
        (Static False) -> evalExpr el
        Dynamic -> do
            tn' <- evalExpr tn
            el' <- evalExpr el
            return (FTri b' tn' el')
```

Figure 1: Sample evaluator code in Haskell

Similarly if n were hard-wired then the control flow could be statically determined and muxes would not be created. Static elaboration thus bears some resemblance to partial evaluation, or to other static optimizations such as loop-unrolling.

Figure 1 shows sample code for the FSpec evaluator for while-loops and trinary ? : operators.

2.2 The Cost of Static Elaboration

The amount of compiler time required to perform elaboration increases as descriptions become more generalized. For complex designs, experience has shown that elaboration is the most costly compiler phase. For example, the UNUM project is a framework written in Bluespec to model microprocessors [3]. To maximize code reuse it consists of a library of generalized CPU components such as Fetch units, ALUs, etc.

To use UNUM an architect constructs a processor via the “tinker toy” approach. For example, the architect may wish to measure the impact of a new branch

Num Slots	Compile Time (seconds)						Dominating Function (% of execution time)		
	typecheck	elaborate	bool opt	schedule	...	total	1st	2nd	3rd
2	7.95	**105.59**	26.44	2.54	...	318.64	`getIdQual`(8.85%)	`getIdBase` (7.1%)	`checkUse` (6.3%)
3	7.93	**214.49**	36.64	5.15	...	607.00	`getIdQual` (9.5%)	`cmpE` (8.5%)	`getIdBase` (8.3%)
4	7.91	**823.45**	72.34	9.64	...	1557.18	cmpE (10.8%)	`getIdQual` (8.7%)	`getIdBase` (8.2%)
5	8.04	**2410.27**	96.19	18.80	...	3479.86	`cmpE` (12.8%)	`getIdQual` (8.8%)	`getIdBase` (7.9%)
6	8.01	**DNF**	-	-	...	-	-	-	-

Figure 2: Profiling information from the Bluespec compiler synthesizing a complex processor reorder buffer with a given number of slots. As expected the typechecking phase remains constant, whereas the boolean optimization phase and scheduling phases grow linearly. Elaboration, on the other hand quickly dominates the execution. At a low number of slots the majority of execution time is spent in the functions `getIdQual` and `getIdBase`, which are identifier-manipulation functions used throughout the compiler, including elaboration. As we increase the number of slots the `cmpE` function, which the evaluator uses to compare expressions for equality during control-flow optimization, becomes dominant. At six slots elaboration did not finish due to running out of memory. It should be noted that typical modern microprocessors use ROBs with 64 slots or more.

predictor algorithm. They would construct a new module to represent their algorithm, and instantiate library elements to represent the remainder of the processor, say a 32-bit PowerPC Decoder, a 64-element Re-Order Buffer, a Register File with 3 read ports and two write ports, etc. This allows the architect to leverage existing code, but means that 95% or more of the system will be generated via static elaboration.

Figure 2 shows the results of profiling the Bluespec compiler on a generalized UNUM reorder buffer (ROB). These results were created using the standard profiling features of the Haskell compiler GHC 6.4.1 on Bluespec compiler version 3.8.60. Clearly, the elaboration phase dominates, even beyond the boolean optimization phase. Overall, the function `cmpE`, which compares expressions for equality during control-flow optimization in elaboration, accounts for nearly 13% of the execution time of the entire compiler.

Our goal is to demonstrate that the application of techniques developed for the analysis of software can be used to improve elaborator performance. In this work we will not directly consider techniques to optimize the hardware which the elaborator produces, although such techniques could be added in the future.

2.3 Other Approaches to Static Elaboration

In this section we compare our approach to static elaboration to other existing projects. The elaboration approach we present is based on the Bluespec hardware description language and the Guarded Atomic Action semantic model. Rosenband [6] showed how static elaboration can be used to transform hardware designs to ensure that they meet user-defined performance specifications. To model these transformations he developed the languages MRL and FRL. These languages are predecessors to the FSpec language we used for this work - specifically FSpec can be seen as MRL extended with polymorphism, and thus requiring static elaboration.

An alternative approach to static elaboration is Lava [2], which advocates strong elaboration through a library of reusable circuit patterns. Lava is a Domain-Specific Embedded Language implemented in Haskell. As such the user can take advantage of the parameterization and polymorphism available in Haskell directly. IE in Lava, a module with a parameter is a Haskell function which takes a parameter and returns a module. This is in contrast to our language FSpec, which is intended to represent an Abstract Syntax Tree after parsing. Thus parameters and functions are modeled in the FSpec syntax tree, rather than in Haskell directly.

Design	Parameters	Description	Uses
RegFile	data and address width, number of registers	A polymorphic register file	arrays, recursive functions
FIFO	data width, buffer length	Queue of any datatype, any length	arrays, while loops
Shifter	data width, shift range	barrel-shifter of any width	arrays, recursive functions
FIR Filter	data width, coefficient width, number of taps	A generalized linear FIR filter	array of submodules, while loops
Hamming	data width, buffer lengths,	Generates the first k Hamming numbers (numbers like 2p3q5r)	arrays, FIFO submodules
Cache	data,address, and tag widths, cache size	A blocking, write-through, direct-mapped cache	arrays, RegFile and FIFO as submodules

Figure 3: Overview of Benchmark Designs

Recent research has developed a novel elaboration technique known as "shadow wires" to improve Lava's handling of irregular circuits [8]. Under this technique wires are threaded throughout the design which carry statically-determined high-level knowledge. This knowledge can be used to make decisions e.g. for boolean or low-power optimization. This fits under our notion of static elaboration as it can be viewed as statically determined data which influences evaluator decisions but which does not translate directly into hardware.

SAFL [7] is a functional language where memory is statically allocated, and thus is well-suited to hardware description, or hardware-software co-design. On the surface, SAFL's abstract syntax is quite close to that of FSpec which we present here. The major difference is that SAFL does hardware resource sharing through source code identifiers. Thus in SAFL if the user wants two copies of a multiplier, the user must textually duplicate and rename the multiply function in the source code. Static elaboration is one method that could be used to overcome this limitation, as the elaborator would essentially perform this duplication for the user.

It should be noted that there have been numerous approaches to synthesizing hardware from high-level software languages such as C. Although these transformations are static and performed by a compiler, we view this approach as distinct from static elaboration. In these cases the goal is to take a possibly-unsynthesizable imperative description and attempt to synthesize it while finding an optimal amount of parallelization, whereas in static elaboration the goal is to take a program which describes how to synthesize a hardware block over many possible values and perform the synthesis for a set of concrete parameters. We do not expect that analysis techniques to improve static elaboration will be applicable to sequential synthesis.

3. METHODOLOGY

To measure the performance of the FSpec evaluator we created a suite of simple benchmark designs, as described in Figure 3. These designs can be described concisely in FSpec, yet are generalized along various parameters. This allows us to measure the performance of our elaborator across a range of data points by varying parameter values.

The results of running these benchmarks through the initial reference elaborator across a range of parameter values are shown in Figure 4. Performance was measured using the standard code profiler included in the GHC Haskell compiler version 6.4.1, which allowed us to obtain results about specific function calls without the need for instrumentation. The FSpec evaluator is dominated by heap references and variable lookups, as shown in Figure 5.

The results demonstrate that our reference evaluator can exhibit hyper-linear growth, similar to the actual Bluespec compiler. In contrast to the Bluespec compiler however, heap accesses are often the bottleneck in our language. For example, in the Hamming benchmark heap accesses consumed 98% of the average execution time. Of concern is the dominance of the `lookupVar` function in the RegFile, Shifter and

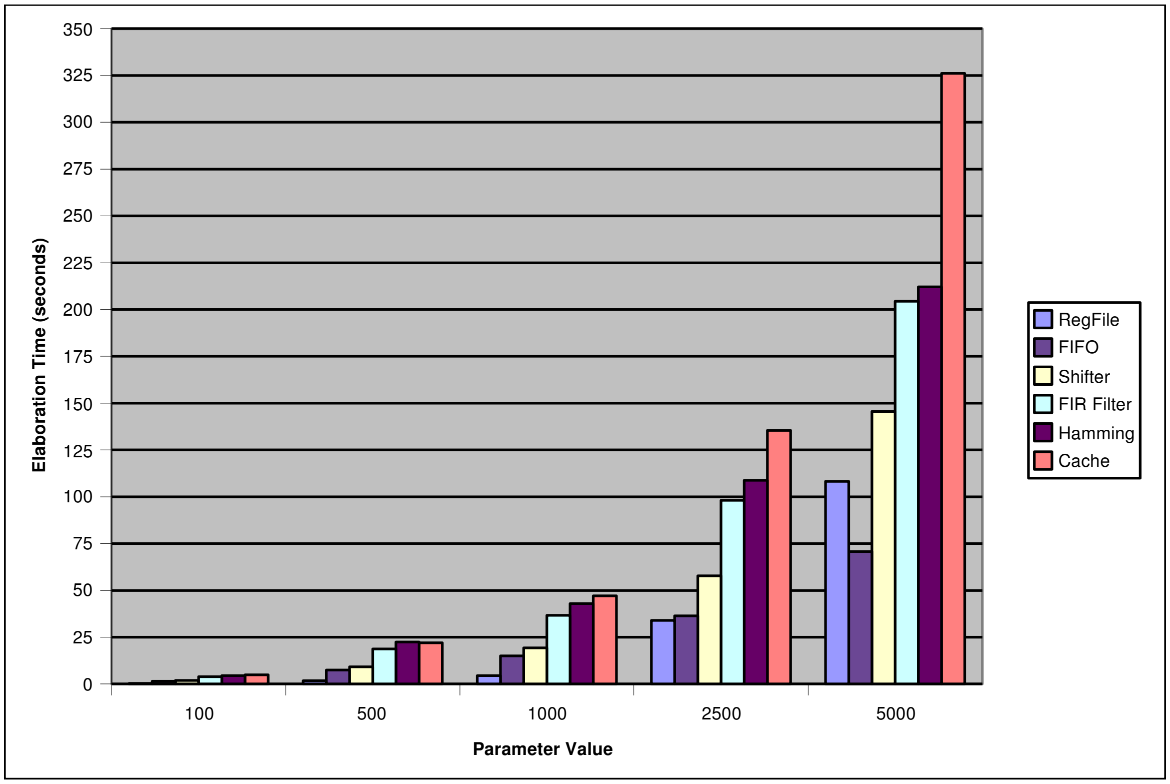


Figure 4: The results of running our benchmark designs on the reference evaluator. The parameter values of the designs were increased to measure the growth in evaluation time. The parameters were as follows: RegFile (num regs), FIFO (buf−fer length), Shifter (shift amount), FIR Filter (num taps), Hamming (buffer length), Cache (cache size). These results demonstrate that the elaborator can exhibit hyper-linear growth times. Although in practice there is not much need for a register file with 5000 registers, or a FIR Filter with 5000 taps, we view large values for these parameters as analogous to more complex systems composed of many generalized submodules.

Design	**Dominating Functions (percent of elaboration time)**		
RegFile	`deHeap` (32%)	`lookupVar` (31%)	`addHeap` (30%)
FIFO	`deHeap` (48%)	`addHeap` (45%)	`evalStmt` (2%)
Shifter	`lookupVar` (37%)	`deHeap` (32%)	`addHeap` (30%)
FIR Filter	`addHeap` (44%)	`deHeap` (39%)	`updateHeap` (13%)
Hamming	`addHeap` (49%)	`deHeap` (47%)	`updateHeap` (1%)
Cache	`lookupVar` (34%)	`deHeap` (32%)	`addHeap` (32%)

Figure 5: This table shows the three most dominating functions for the FSpec evaluator on the benchmark designs run at n = 5000. `deHeap` removes a variable from the heap cell after it has left scope. `addHeap` adds a new variable to the heap. updateHeap updates the value of a heap cell. `lookupVar` searches the environment for a variable and de-references it from the heap. `evalStmt` is a central evaluation function.

Cache (which uses RegFile as a submodule). This information indicates that any transformation that saves us from unnecessarily altering the state of the heap or looking up variables should result in an improvement in performance.

Given the properties of the reference evaluator we now attempt to use traditional software analysis techniques to improve its performance. The technique we have chosen is Pfenning's Higher-Order Abstract Syntax (HOAS) transformation, which should improve performance by decreasing the number of variable lookups and heap operations.

4. HIGHER-ORDER ABSTRACT SYNTAX FOR FSPEC

Higher-Order Abstract Syntax refers to the technique augmenting a standard abstract syntax tree with lambda calculus constructs. Practically, this means using implementation language features such as lambda-functions, to model similar constructs in the target language. For example, our abstract syntax for FSpec expressions contains the following productions to represent function definition and application:

```
data FExpr =
   ...
   | FELam Id FExpr          -- \x -> e
   | FEFuncApp FExpr FExpr   -- e1 e2
```

Thus we will parse the application:

```
(\x -> x + 5) (2 + 3)
```

into the following abstract syntax (simplified for readability):

```
  FEFuncApp (FELam 'x'
    (FEAdd (FEVar 'x') 5)) (FEAdd 2 3)
```

Note that when executing the above statement, the elaborator must explicitly bind the variable, and rename free variables to avoid capture:

```
evalExpr (FEFuncApp e1 e2) =
 do
    -- type checking guarantees this:
    (FELam v f) <- evalExpr e1
    e2' <- evalExpr e2
    newVar v e2'
    res <- evalExpr f
    delVar v
    return res
```

If we use HOAS the user-defined function will be stored in the AST as an actual lambda function:

```
data FExpr =
  ...
  | FELam (FExpr -> FExpr)   -- \x -> e
  | FEFuncApp FExpr FExpr    -- e1 e2

(\x -> x + 5) (2 + 3) ==>

  FEFuncApp (FELam (\x -> EAdd x 5))
            (EAdd 2 3)
```

Note that the variable `x` has changed from a constructor in the abstract syntax to a direct Haskell variable. Performing the application at elaboration time is now equivalent to performing an actual Haskell application:

```
evalExpr (FEFuncApp e1 e2) =
 do
    -- type checking guarantees this:
    (FELam f) <- evalExpr e1
    e2' <- evalExpr e2
    res <- evalExpr (f e2')
    return res
```

The variable `x` must still be defined and bound, but this binding will be handled by the Haskell runtime system, which we expect to be more efficient than being performed by the elaborator. References to `x` will be use the heap of the Haskell run-time environment, instead of our user-defined heap structure, which should improve performance.

The results of running our benchmark suite after transformation are shown in Figure 6. These results show an improvement on five of the six benchmark designs. The RegFile shows the largest improvement percentage, as it uses recursive functions which greatly benefit from the HOAS. The Cache design uses the RegFile as a submodule, and thus benefits similarly. The FIFO, FIR Filter, Hamming designs use while-loops instead of recursive functions, and thus do not benefit as much. One possible experiment would be to transform the while-loops into recursive functions to see if the benefit can be increased.

The sixth design, the shifter, also uses recursive functions, even more so than the RegFile. We would therefore expect that it would show the largest im–

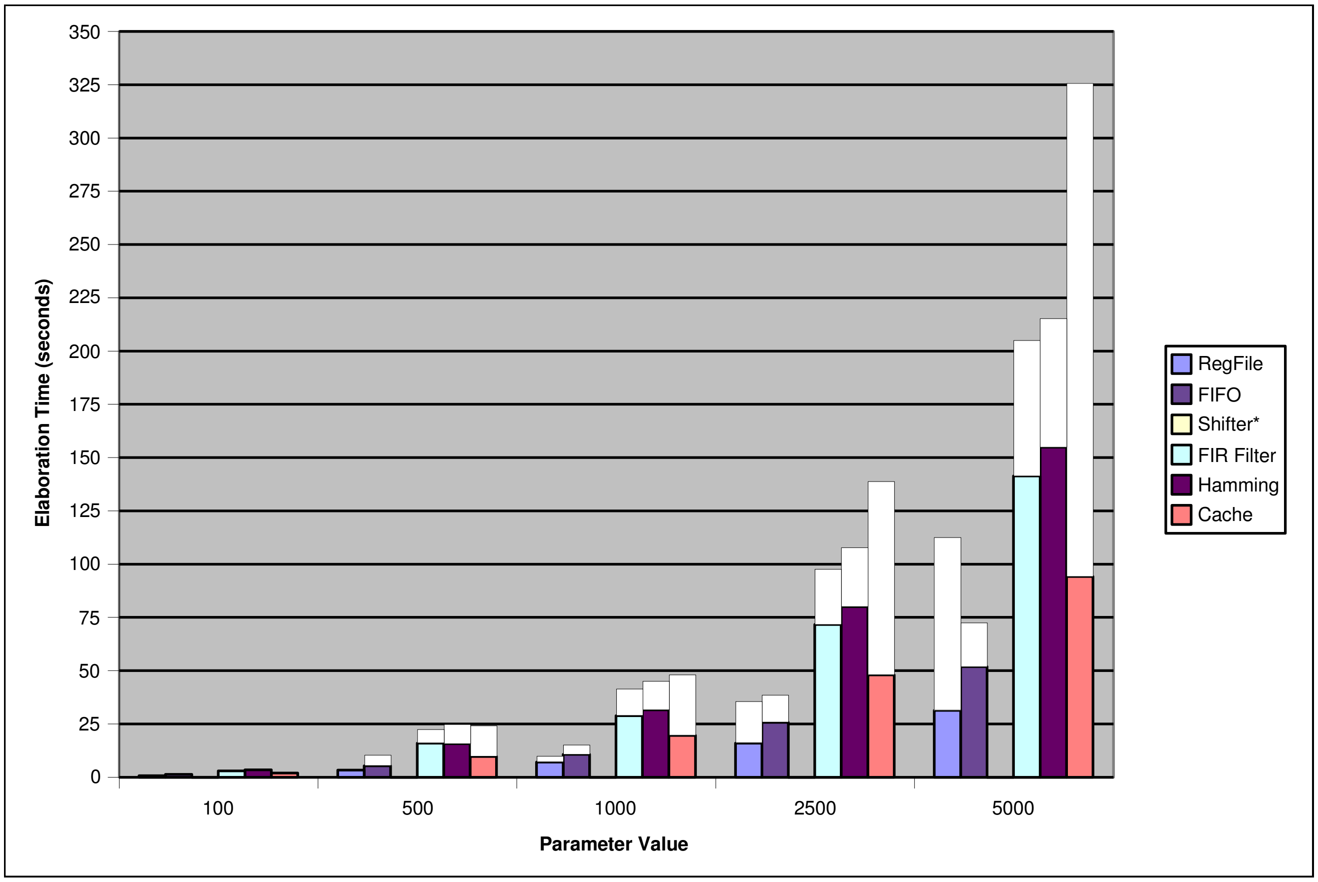


Figure 6: Evaluator performance results after implementing Higher-Order Abstract Syntax. Original evaluation times from Figure 4 are listed in white. The transformation resulted in an average improvement of 45.4% on 5 of the 6 designs, including improvements of greater than 70% on the RegFile and Cache (which uses both the RegFile and FIFO as sub–modules, so we would expect to show the largest improvement). For the shifter design, however, performance significantly degraded to the point where evaluation could only complete with parameter values of less than 20. For analysis onto the cause of this degradation see Section 4.1.

Design	Dominating Functions (percent of elaboration time)		
RegFile	`addHeap` (49%)	`deHeap` (46%)	`evalExpr` (2%)
FIFO	`deHeap` (48%)	`addHeap` (47%)	`evalExpr` (1%)
Shifter	-	-	-
FIR Filter	`addHeap` (43%)	`deHeap` (39%)	`updateHeap` (15%)
Hamming	`deHeap` (48%)	`addHeap` (47%)	`evalExpr` (1%)
Cache	`addHeap` (48%)	`deHeap` (46%)	`evalExpr` 32%)

Figure 7: The three most dominating functions for the HOAS evaluator on the benchmark designs for n=5000. In the previous results (Figure 5) `lookupVar` was a dominant function for the RegFile, Shifter, and Cache designs. After the HOAS transformation this call is eliminated completely. Although total execution time has decreased, heap manipulations for values not affected by the HOAS transformation (such as incoming wire ports to methods) remain the majority of the execution time. Future work should focus on improving the representation of the heap data structure to speed these operations.

provement from the transformation. Contrary to our expectations, the HOAS version of the Shifter design had its performance degraded. Previously the compiler could elaborate an 5000-bit wide shifter in 147 seconds. After the HOAS transformation the compiler takes 213 seconds to elaborate a 19-bit wide shifter. Larger values than 19 cause the elaborator to run out of memory after over an hour of execution. We now analyze this case in detail.

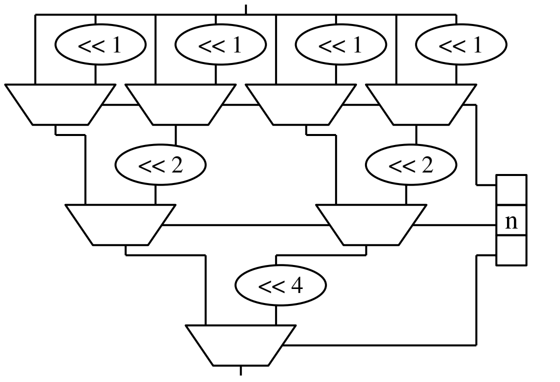
4.1 The Effect of HOAS on the Shifter Design

The Shifter benchmark design is similar to the one described in Section 2.1, except that it uses recursive functions rather than while-loops to describe the shift/mux pairs. For a shift amount of *n*, we would expect *n* function calls or loop iterations. If each iteration contains *k* program expressions, we would expect *kn* calls to the `evalExpr` function. Profiling information revealed the following behavior:

	time	**evalExpr calls**	**Memory use**
Original (n=19)	0.24s	543	51MB
Original (n=20)	0.22s	571	51MB
HOAS (n=19)	17.64s	23,068,544	3.9GB
HOAS (n=20)	60.86s	46,137,211	7.9GB
HOAS (n=21)	DNF > 1 hour	-	Out of Memory

The original version clearly demonstrates linear behavior, whereas the HOAS version results in an exponential number of calls to `evalExpr` (23 million is approximately 43×2^{19}), and exponential memory use. The other benchmarks in our suite exhibit no such behavior, nor increased memory use.

Investigation revealed that the HOAS shifter was generating unexpected, but functionally correct circuitry. For *n*=3 we would expect elaboration to generate the circuit diagrammed in Section 2.1. Instead it turned out that the HOAS evaluator generated the following structure:



Note that this circuit is functionally equivalent to the circuit we had intended to describe. Thus the testbench for this design identified it as a correctly functioning shifter circuit, and we did not examine the hardware structure in detail until we investigated the elaboration time. Although functionally equivalent, this circuit has the same critical path as the intended circuit, but exponentially larger area, and therefore is strictly worse and would not be desired by the designer under any circumstances.

But why was this circuit being generated? Consider the original *n*=3 shifter description:

```
method v_width shift(v_width v,
                        bit[3] n) {
     v_width v1, v2, v3;
     v2 = (n[0] == 0) ? v : v << 1;
     v3 = (n[1] == 0) ? v2 : v2 << 2;
     v4 = (n[2] == 0) ? v3 : v3 << 4;
     return v4;
   }
```

If we naively substitute the variable `v2` into the definition of `v3`:

```
     v3 = (n[1] == 0) ?
            ((n[0] == 0) ? v : v << 1)
          : ((n[0] == 0) ? v : v << 1)
            << 2;
```

then we have duplicated hardware. This is an example where a transformation that is safe to perform in software is undesirable for a hardware compiler.

We are currently investigating how to reconcile this behavior with Haskell's non-strict semantics. Given a function such as:

```
\v -> (b) ? v : (f v)
```

we would expect the argument *v* to be to be evaluated only once, and the result to be re-used when needed. Therefore our elaborator should not exhibit exponential behavior, although it clearly does. We are currently investigating whether some of the GHC Haskell compiler's *eagerness* optimizations could cause a bad interaction in this case. In the meanwhile we are attempting to implement explicit sharing via memoization as a stopgap solution.

5. CONCLUSIONS

As hardware designs become more and more complex, hardware description languages are continually adding new features to compensate. We believe that hardware description languages should not reinvent the wheel, but rather leverage techniques that have been developed in the mature field of software programming languages. In this paper we demonstrated that the Higher-Order Abstract Syntax technique can be used to speed up the static elaboration phase of a hardware compiler. Additionally we demonstrated that HOAS can be used as a framework to transmit information from static analyses to the elaborator. In the future we hope to apply this work to the actual Bluespec compiler, and to expand our framework to include analyses which perform optimizations of the generated circuit structures rather than improving compiler performance.

6. ACKNOWLEDGEMENTS

We would like to thank Ravi Nanavati and the rest of Bluespec, Inc. for their ideas and support.

REFERENCES

[1] Arvind, R. S. Nikhil, D. L. Rosenband and N. Dave, High-Level Synthesis: An Essential Ingredient for Designing Complex ASICs. In *Proceedings of ICCAD '04*, pages 775-782, 2004.

[2] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware Design in Haskell. In *Proceedings of International Conference on Functional Programming (ICFP) '98*, pages 174-184, 1998.

[3] N. Dave, M. Pellauer and Arvind. The UNUM Microprocessor Framework. Unpublished.

[4] J.C. Hoe and Arvind. Synthesis of Operation-Centric Hardware Descriptions. I*n Proceedings of ICCAD '00*, pages 511-518, 2000.

[5] F. Pfenning and C. Elliot. Higher-Order Abstract Syntax. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI) '88*, pages 199-208, 1988.

[6] D. Rosenband. A Performance-Driven Approach to Hardware Synthesis of Guarded Atomic Actions. PhD Thesis, MIT, 2005.

[7] R. Sharp and A. Mycroft. A Higher-Level Language for Hardware Synthesis. *Lecture Notes on Computer Science*, 2144:228-240, 2001

[8] M. Sheeran. Finding Regularity: Describing and Analyzing Circuits that are not Quite Regular. In *Proceedings of the Conference on Correct Hardware Design and Verification Methods (CHARME) '03*, pages 4-18, 2003.

7. APPENDIX

Abstract Syntax Tree for FSpec in Haskell, and ex− amples of concrete syntax.

```
data FModule = FModule                   -- module nm < Types > { stmts }
  {
     modname :: Id,
     modtype :: FType,
     params :: [(FType, Id)],
     decls :: [FStmt],
     act_methods :: [FActionMethod],
     val_methods :: [FValueMethod],
     rules   :: [FRule]
  }

data FRule = FRule Id FExpr [FStmt]      -- rule nm when (p) { stmts }

data FActionMethod = FActionMethod       -- method Action nm ( params )
     FType Id FExpr [Id] [FStmt]         --        when ( expr ) { stmts }

data FValueMethod = FValueMethod         -- method Type nm ( params )
     FType Id FExpr [Id] FExpr           --        when ( expr ) expr

data FStmt =
   FSRegUpdate FExpr FExpr               -- r <= expr ;
   | FSDecl FType Id FExpr               -- Type var = expr ;
   | FSArrayUpdate Id FExpr FExpr        -- var[expr] = expr ;
   | FSVarUpdate Id FExpr                -- var = expr ;
   | FSIf FExpr [FStmt]                  -- if ( expr ) { stmts } ;
   | FSMethodCall FExpr Id [FExpr]       -- m.actmeth ( params ) ;
   | FSWhile FExpr [FStmt]               -- while ( expr ) { stmts };

data FExpr =
   FERegRead FExpr                       -- r
   | FEConst Value                       -- 1'b0, 1'b1, ...
   | FEInt Int                           -- 0, 1, 2, ...
   | FEVar Id                            -- var
   | FELam FType Id FExpr                -- \ x -> expr
   | FEFuncApp FExpr FExpr               -- f expr
   | FETri FExpr FExpr FExpr             -- ( expr ) ? expr : expr
   | FEArrayRead FExpr FExpr             -- var[expr]
   | FEArray (Array Int FExpr)           -- { expr, expr,... }
   | FEMethodCall FExpr Id [FExpr]       -- m.valmeth( params )
```