A scripting engine for combining design transformations

Tim Todman, Qiang Liu, Wayne Luk
Department of Computing
Imperial College London
United Kingdom
{*timothy.todman, qiang.liu205, w.luk*}*@imperial.ac.uk*

George Constantinides
Department of Electrical Engineering
Imperial College London
United Kingdom
g.constantinides@ic.ac.uk

Abstract

This paper descrribes a scripting engine based on the Python language and the ROSE compiler framework. Our scripting engine supports hardware design involving both syntax-directed transformatins such as loop coalescing, and goal-directed transformations such as geometric programming. We show how customizing the composition and parameterization of design transformations can lead to designs with different trade-offs in performance and resource usage.

1. Introduction

To implement complex designs quickly, designers increasingly turn to high-level design descriptions, which ease design capture and design space exploration, and allow rapid prototyping and fast time to market. However, to meet design goals, designers must apply multiple transformations to optimize their design while maintaining the intended behaviour.

Many current hardware compilation tools support a fixed recipe of applying multiple transformations, but designers have few options to adapt the recipe without re-writing the tools. In addition, complex transformations based on linear programming and geometric programming are rarely included.

This paper describes a novel approach combining customization of transformations, and their automated application. The key innovation of this approach is to enable users to customize the composition and parameterization of transformations from a library of transformations including both syntax-directed and goal-directed transformations as shown in Fig. 1. Syntax-directed transformations match and transform syntax patterns if some Boolean conditions are met. Designers can specify target patterns, the conditions and how to transform the pattern. Goal-directed transformations target more complex transformations, where all possible transformation options are modeled in an optimization problem such that the transformed designs meet the goals specified by designers. Although our approach could apply to software compilation, we focus oc compiling for Field Programmable Gate Arrays (FPGAs).

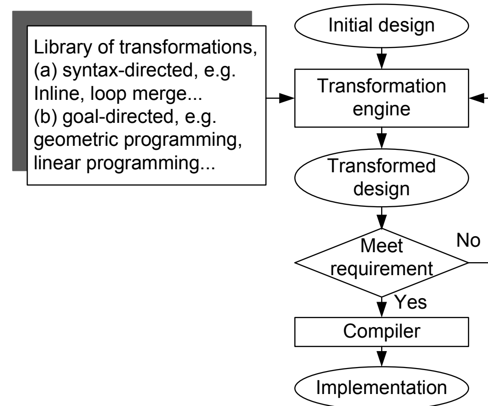

Figure 1. Proposed approach for customizing design transformations.

This paper makes the following contributions:

- We define requirements for a customizable transformation engine.
- We design and implement an engine meeting our requirements, showing how we provide appropriate levels of abstraction for composing and parameterizing transformations.
- We evaluate our approach, showing how it can be used to optimize for speed and resource usage.

The rest of the paper runs as follows. Section 2 gives background and related work. Section 3 shows requirements for a transformation engine. Section 4 shows the design and implementation meeting the requirements. Section 5 shows results from applying the approach to several benchmarks. Finally, Section 6 concludes and gives ideas for future work.

2. Background

Hardware compilers: Commercial hardware compilers include Catapult C Synthesis [1], CoDevel-

oper [2] and Celoxica DK Design Suite [3]. Each targets a different C-like language. These compilers perform several optimizations, such as retiming, common subexpression optimization, memory pipelining, *etc.*, mainly achieved by syntax-directed transformations. Rewriting a design from C/C++ to Impulse C or Handel-C is not trivial.

Several academic efforts optimize high level hardware design. Syntax-directed transformations, such as code motion, loop transformation, dynamic renaming and scalar replacement, are used in [4] and [5]. Goal-directed transformations are implemented in [6] and [7]. Liu *et al.* [6] propose a geometric programming approach to explore the data reuse and loop-level parallelization design space for FPGA targeted hardware compilation. An integer linear programming model in [7] pipelines outer loops in FPGA hardware coprocessors. However, all these approaches need external support for transformation to complete the optimization.

Liu *et al.* [8] identify two kinds of transformation, goal-directed and syntax-directed, and combine them in one approach. Using syntax-directed and goal-directed transformations together simplifies goal-directed transformations, as they can rely on syntax-directed transformations to render their input into a suitable form. Liu *et al.* integrate the two approaches, but do not automate the compile process.

Compiler frameworks: Compiler frameworks SUIF [9] and ROSE [10] provide a means of building compilers from components, but choosing the next transformation is left to the user. CoSy [11] has less common ways of composing transformations: competitive and parallel, but with similar limitations to other approaches. Compiler frameworks allow multiple patterns to be used together, but do not support goal-directed transformations. All three approaches target software compilation; we primarily target hardware compilation.

Syntax-directed transforms for hardware compilation have been explored by di Martino *et al*. [12] on data-parallel loops, as part of a synthesis method from C to hardware. Unlike our method they do not allow user-written transforms. Pattern matching and transforming can also be done in tree rewriting systems such as TXL [13], but the generality of such systems makes them difficult to incorporate hardware-specific knowledge into the transformation process.

Build tools: The Make tool [14] is fine for building an executable from a dependence graph, but iteration requires outside tools, such as shell scripts. Make supports building in parallel, allowing independent parts of a build to run on different processors. This speeds up large builds rather than comparing the results of different transforms on the same inputs. Many other tools exist to correct features of make, including Ant, Bjam, Aegis, but have no specific support for iteration or hardware compilation.

Scripting languages such as Python [15] have good support for embedding in programs written in C. Our approach extends scripting into a domain-specific language for hardware compilation. Many other scripting languages exist, but may lack proper type systems or be overly complex.

This paper builds on previous work [16] combining syntax-directed and goal-directed transformations in one approach. We focus on the transformation engine, allowing users to customize the compiler and to adapt the transformations to applications. We aim to combine the strengths of existing approaches with more user control.

3. Requirements

We derive a list of requirements for a customizable transformation engine.

- Allow multiple transformations.
- Support common and domain-specific transformations.
- Compose and parameterize transformations.
- Playback for verification and reuse.

We now explain the requirements in detail.

Allow multiple transformations: Experience with previous approaches such as SUIF and ROSE shows that optimization can be usefully decomposed into multiple optimization passes. Allowing multiple optimizations is thus a minimum requirement.

Support common and domain-specific transformations: Research in compilers identifies many useful optimizations. Previous work [8] on domain-specific transformations shows that combining standard and domain-specific transformations allows users to achieve design goals by choosing a sequence of suitable transformations.

Compose transformations: *Sequence*: calls several transformations with output of one being input to another. Sequence composition does not check for success or failure of the composed transformations; it is useful for initial patterns for goal-directed transformations.

The engine supports other ways of composing transformations, such as conditional sequence and competitive, which are beyond the scope of this paper.

Playback for verification and reuse: Playback requires recording into a logfile at each step: design and transformation parameters used, and the choice of next step. The recording can then be replayed, either on the same design (verification), or on other designs (reuse).

Verification: transformations must preserve intended design behaviour. Playback allows verification by finding which step deviates from initial behavior. The user can replay the recorded sequence to find which step caused the failure, much like single-stepping a debugger but with transformations instead of program

lines. This step could then be removed from the script, debugged separately, different parameters used or a pre-requisite added.

Reuse: successful compilation compositions may be reused for other designs. For example, in DSP filter implementation, designs with different numbers of taps can benefit from same transformation sequence. Other sequences might optimise for particular targets.

4. Design and Implementation

To meet the requirements, we propose the following design for the transformation engine: a customizable approach for source-to-source compilation, with fixed components for common tasks such as front end (parser), back end (unparser) and utilities such as a symbol table and means of composing transforms. Each input design is first parsed into an abstract syntax tree (AST) and symbol table. The designer applies different transformations to the AST: standard transformations such as loop unrolling, or custom transformations for particular domains or applications, to optimize for their design goals. When the goals have been met, the back end renders the transformed design back into C source code.

Each transformation has input parameters:

- *Current design*: including AST and symbol table.
- *Platform constraints*: number of multipliers, memories, DSP resources, and so on.
- *Design goals*: for now restricted to the overall optimization goal – speed, size, power.
- *Transform-specific parameters*: for example, unroll factor.

and produces as output:

- *Transformed design*: if successfully applied.
- *Output parameters*: showing how the design was transformed and allowing later transformations to benefit from any analysis results.

Both the approach as a whole, and each individual transformation, are source-to-source. Keeping the design at the source level avoids compiling to hardware description languages.

Our design meets the requirements as follows:

Allow multiple transformations: Separating each transformation into independent units allows multiple transformations.

Support common and domain-specific transformations: Previous work shows that combining common and domain-specific transformations together can optimize realistic benchmarks. We provide a library and allow custom transformations to extend the library.

Compose transformations: any of the ways of composing transformations may be used.

Implementation: Our implementation uses the ROSE framework with a domain-specific language for syntax-directed transformation. We choose ROSE

Table 1. Transformations from our library used in experiments.

Goal-directed	Description
GP1	Speed optimization exploiting data reuse and MapReduce
GP2	Speed optimization exploiting MapReduce and pipeline

Syntax-directed	Description
LM	Loop merging
DE	Decompose expressions with the 3-address rule
scriptsize RF	Reduce fanout of variables
Par	Parallelize independent statements

Table 2. Some example transform parameters

Params	Type	Description
numLoop	int	number of loop levels
loopBound	int Array	upper bound of each loop after normalization
numArray	int	the number of arrays
numRAMBlock	int	number of on-chip RAM blocks
memBandwidth	int	memory bandwidth
numMultiplier	int	number of on-chip multipliers

because it is mature, has comprehensive C / C++ support, and contains existing features for program transformation including function inlining.

We script ROSE with the Python scripting language [15], using the Boost Python library, a quasi-standard way of reflecting C++ objects and classes into Python [17]. Using a scripting language to control compilation allows designers to use language facilities for control and iteration. Furthermore, the transformation engine can extend the scripting language command-line interpreter, allowing transformations to interactively apply to the program. If a good sequence of transformations is found, it can be saved in a script and applied to other designs.

Our implementation meets the remaining requirement – *playback for verification and reuse*. Before each step, the engine records into a log file: the current design, the next step, and the parameters to use. This is all the information needed to replay the step. Each custom transformation must record its own parameters and anything else needed for replay.

Example: We show an example script for matrix multiplication. Table 1 shows the library of available transformations; table 2 shows transformation parameters. The sequence we apply is: GP1 - DE - RF - Par - GP2. Python pseudocode is:

```
1: InDesign=parse("matmult.c")
2: Ini_design_params=[("numLoop1",3),...]
3: Platform_params=[("numRAMBlock1",168),
  ...]
4: (des1, Out_design_params)=GP1(InDesign,
  Speed, Ini_design_params, Platform_params)
5: des2=decomposeExpressions(des1)
6: des3=reduceFanout(des2)
8: des4=parallelize(des3)
9: Ini_design_params=[("loopBound2",[64]),
 ...]
10: Platform_params=[("memBandwidth2",32),
 ...]
11: (OutDesign, Out_design_params)=GP2(des4,
  Speed, Ini_design_params,
```

Table 3. Linear reduction vs tree reduction for FIR

Designs	Exe time (ms)		# Slices	
	Linear	Tree	Linear	Tree
3-tap	0.68	0.69	70	89
5-tap	0.68	0.7	96	164
11-tap	1.32	1.35	155	369

```
  Platform_params, innermostLoop, tree)
12: unparse(OutDesign, "out.c")
```

where numbered source lines work as follows:

1) Parse input file into variable `InDesign`.
2) store GP1 model parameters in variable `Ini_design_params`; we omit some parameters for space reasons.
3) platform parameters are stored in another variable.
4) apply GP1 transform to InDesign, yielding des1.
5) three syntax-directed transformations apply to des1, yielding des4.
9) parameters for the second goal-directed transform.
10) store platform parameters for GP2.
11) run GP2 transform.
12) finally, unparse the design to output file.

5. Results

We show results for applying syntax-directed and the GP2 model-directed transformations to Finite Impulse Response (FIR) filtering. The GP2 goal-directed transformation allows designers to choose which reduce structure—tree or linear—is used in the reduce phase of MapReduce [16]. Different structures result in different system latency, throughput and resource usage for different applications. Table 3 shows the results of applying GP2 transformation to filters with 3, 5 and 11 taps. In all three cases, GP2 with tree reduce structures results in designs with similar execution time, but using a linear structure can reduce the slices used. Hence designers would choose the transformation with the linear reduce structure. Other applications may need different parameters and transformation sequences.

6. Conclusion

This paper proposes a novel approach that enables designers to customize the composition and parameterization of design transformations. Our approach is implemented by a scripting engine based on the Python language and the ROSE compiler framework, which supports both syntax-directed transformatins such as loop coalescing, and goal-directed transformations such as geometric programming. Current and future work includes automating transformation choice and profile-driven optimization, as well as verifying transformation correctness.

Acknowledgment: The support of the FP6 hArtes project, EPSRC and Xilinx is gratefully acknowleged. The authors would also like to thank Tobias Becker, Gabriel Coutinho, David Thomas and Brittle Tsoi for help and encouragment.

References

[1] http://www.mentor.com/products/esl/high_level_synthesis/catapult_synthesis/, accessed Jan. 2010.

[2] http://www.impulsec.com/C_to_fpga_overview.htm, accessed Oct. 2005.

[3] http://www.agilityds.com, accessed May 2008.

[4] S. Gupta et al., "SPARK: a high-level synthesis framework for applying parallelizing compiler transformations," in *Proc. Int. Conf. on VLSI Design*, 2003, pp. 461–466.

[5] Z. Guo et al., "Input data reuse in compiling window operations onto reconfigurable hardware," in *ACM SIGPLAN/SIGBED Conf. LCTES*. ACM, 2004, pp. 249–256.

[6] Q. Liu et al., "Combining data reuse with data-level parallelization for FPGA-targeted hardware compilation: A geometric programming framework," *IEEE Trans. CAD.*, vol. 28:3, pp. 305–215, 2009.

[7] K. Turkington et al., "Outer loop pipelining for application specific datapaths in FPGAs," *IEEE Trans. VLSI.*, vol. 16:10, pp. 1268–1280, 2008.

[8] Q. Liu et al., "Optimising designs by combining model-based and pattern-based transformations," in *Proc. Int. Conf. on FPL*, 2009, pp. 308–313.

[9] M. W. Hall et al., "Maximizing multiprocessor performance with the SUIF compiler," *IEEE Computer*, December 1996.

[10] M. Schordan and D. Quinlan, "A source-to-source architecture for user-defined optimizations," in *JMLC'03: Joint Modular Languages Conference*, ser. LNCS, vol. 2789. Springer Verlag, Aug. 2003, pp. 214–223.

[11] ACE, "CoSy Compilers: Overview of Construction and Operation," http://www.ace.nl/compiler/paper-construct.pdf.

[12] B. di Martino et al., "A technique for FPGA synthesis driven by automatic source code synthesis and transformations," *Proc. FPL*, 2002.

[13] "The TXL programming language," http://www.txl.ca/.

[14] R. Stallman et al., *GNU Make: A Program for Directing Recompilation, for version 3.81*. Free Software Foundation, 2004.

[15] M. Lutz, *Programming Python: Object-Oriented Scripting*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2001.

[16] Q. Liu et al., "Automatic optimisation of mapreduce designs by geometric programming," in *Proc. Int. Conf. on FPT*, 2009, pp. 215–222.

[17] http://www.boost.org/doc/libs/1_41_0/libs/python/doc/index.html, "Boost.python documentation," accessed January 2010.