Customizable Composition and Parameterization of Hardware 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*—A promising approach to high-level design is to start initially with an obvious but possibly inefficient design, and apply multiple transformations to meet design goals. Many hardware compilation tools support a fixed recipe of applying design transformations, but designers have few options to adapt the recipe without re-writing the tools themselves. In addition, complex transformations based on linear programming and geometric programming are often not included. This paper proposes a new approach that enables designers to customize the composition and parameterization of different types of design transformations in a unified framework, using a high-level language to control a transformation engine to automate the application of design transformations. Our approach is implemented by a tool based on the Python language and the ROSE compiler framework, which supports both syntax-directed transformations such as loop coalescing, and goal-directed transformations such as geometric programming. We illustrate how customizing the composition and parameterization of design transformations can lead to designs with different trade-offs in performance, resource usage, and energy efficiency. We evaluate our approach on benchmarks including matrix multiplication, Monte Carlo simulation of Asian options, edge detection, FIR filtering, and motion estimation.**

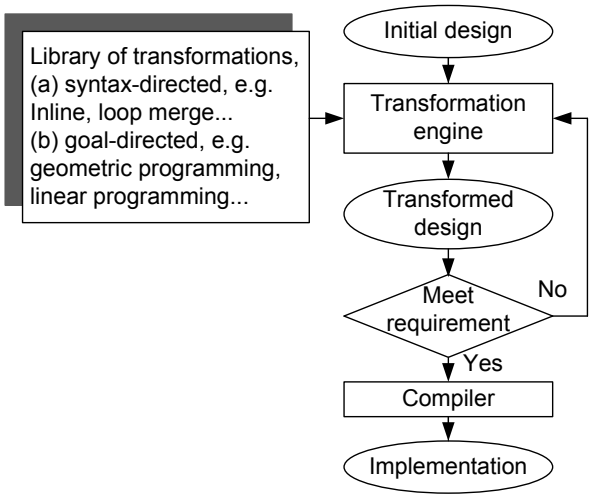

Fig. 1. Proposed approach for customizing design transformations.

I. 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, in order to meet design goals, designers must apply multiple transformations to optimize their design while maintaining the intended behavior.

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

This paper describes a novel approach that combines 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 involve matching and transforming syntax patterns if some Boolean conditions are met. Designers are allowed to specify target patterns, the conditions and how to transform the pattern. Goal-directed transformations target complex transformations, where all possible transformation options are modeled in an optimization problem such that the transformed designs meet the goals specified by designers.

This paper makes the following contributions:

- We define requirements for a customizable transformation engine combining multiple optimizations in a simple composition where syntax-directed and one goal-directed transformation can be sequenced together, and show how our design and implementation meet the requirements.
- We refine our requirements, design and implementation to allow more complex compositions and multiple goal-directed transformations.
- We evaluate our approach on several benchmarks, showing how it can be used to optimize for speed, resource usage and power efficiency.

The rest of the paper is structured as follows. Section II gives background and details related work. Section III shows requirements for simple composition, showing how our design and implementation meet them, while section IV shows how

we extend and modify the requirements, design and implementation for complex composition, where transformations can be composed in several ways and multiple goal-directed transformations can be used. Section V shows results from applying the approach to several benchmarks. Finally, section VI concludes and gives ideas for future work.

II. Background

A. Hardware compilers

There are several commercial hardware compilers, such as Catapult C Synthesis [1], CoDeveloper [2], and Celoxica DK Design Suite [3]. Each targets different C-like languages. These compilers perform several optimizations, such as retiming, common subexpression optimization, memory pipelining, *etc.*, which are mainly achieved by syntax-directed transformations. Rewriting a design from C/C++ to C-like languages, such as Impulse C and Handel-C, is not trivial.

In the academic community, several 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 in the context of FPGA targeted hardware compilation. An integer linear programming model is proposed in [7] for pipelining outer loops in FPGA hardware coprocessors. However, all these approaches need external support for transformations 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 allows goal-directed transformations to be simplified, as they can rely on syntax-directed transformations to render their input into a suitable form. Liu *et al.* concentrate on integrating the two approaches, but do not automate the compile process.

B. Compiler frameworks

Previous compilation approaches, 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 it has the same limitations as other approaches. All three approaches aim at software compilation, whereas we primarily aim at hardware compilation.

Syntax-directed transformations for hardware compilation have been explored by researchers like di Martino *et al*. [12] on data-parallel loops written in C-source code, as part of a synthesis method from C to hardware. Unlike our method they do not allow user-written transformations. Compiler toolkits such as SUIF and CoSy [11] allow multiple patterns to be used together, but they do not support including goal-directed transformations. 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.

C. Build tools

The Make tool [14] is fine for building an executable from a dependence graph, but does not support iteration; iteration requires use of outside tools, such as shell scripts. Make allows parts of a build to run in parallel, allowing independent parts of a build to run on different processors. This is intended to speed up large builds rather than comparing the results of multiple transformations on the same inputs. Many other tools exist to correct features of make, including Ant, Bjam, Aegis, but these have no specific support for iteration or hardware compilation.

Scripting languages such as Python [15] and Lua [16] have good support for embedding in programs written in C. Our approach extends scripting into a domain-specific language for compilation. Many other scripting languages exist, but may lack proper type systems or be overly complex.

III. Simple composition

Simple composition means combining a single goal-directed transformation with multiple syntax-directed transformations.

A. Requirements

Common requirements:

- Allow multiple transformations: syntax-directed and one goal-directed.
- 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 the optimization problem can be usefully broken down into multiple optimization passes. Allowing multiple optimizations is thus a minimum requirement for an optimizing compiler approach. For simple composition, we only allow one goal-directed transformation, whose goal should correspond to the overall design goal; syntax-directed transformations can prepare the goal-directed transformation's input and refine its output.

Support common and domain-specific transformations: Research in compilers has identified many useful optimizations; we call transformations that have proved useful in many application domains *common transformations*. By contrast, *domain-specific* transformations are only useful in one application domain. Previous work [8] shows that combining common and domain-specific transformations allows users to achieve design goals by choosing a sequence of suitable transformations.

Compose transformations: For simple composition, we only allow *sequence* composition: calling several transformations, with output of one being input to another. Sequence is useful for providing initial patterns for goal-directed transformations, or when the design input is in a known form.

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

Verification: a key requirement of any optimization is that the intended design behavior is preserved; clearly, a fast but incorrect design is useless. Playback allows verification by finding which step deviates from initial behavior.

If the final design fails to preserve initial design behavior, the user can replay the script to step through the recorded transformation 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.

While we do not consider formal verification in this paper, the same stepwise approach could also be used. The framework could formally verify that the design after a step is the same, or has the same properties as the design before that step.

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 optimize for particular targets: for example, we have found that one compiler can get a good speedup from inlining, merging, normalization and scalar replacement.

B. Design

To meet the requirements described above, 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 transformations together. Each input design is first parsed resulting in an abstract syntax tree (AST) and symbol table. The designer can then apply different transformations to the AST: either standard transformations such as loop unrolling, or custom transformations for a particular domain or application, attempting to optimize for their design goals. When the goals have been met, the back end renders the transformed design back into C source code. To implement the final design, we translate from C to Handel-C, then use the Handel-C compiler to generate a hardware description that can be compiled using FPGA vendor tools. Currently, the C to Handel-C translation is manual, but this could be mechanized or a tool such as C-to-Verilog [17] used.

We consider each transformation as a function that takes as 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, loop tiling might specify what tile size or range of tile sizes to use.

and produces as output:

- *Transformed design*: if the transformation was successfully applied.
- *Output parameters*: as a key-value list, such as loop partition scheme and operation pipelining scheme, showing how the design was transformed and allowing later transformations to benefit from any analysis results from the current transform.

Goal-directed transformations may have several specific parameters:

- *Objective*: optimization goal
- *Pattern*: required design characteristics for designs targeted by the model.
- *Input design parameters*: these include the properties or specifications of a design, such as loop structure parameters (number of loops, loop bounds, parallelizability), array variable parameters (number of array references, data reuse options of each reference), data flow information (number of data flow levels, computation types), and so on.
- *Platform parameters*: target hardware platform parameters give the physical constraints, including target device (on-chip computation resources, on-chip memory size, on-chip memory bandwidth), off-chip memory (bandwidth, latency, ...) and so on.
- *Constraints*: optional constraints, which could be specified by designers.

We provide a transformation library, containing multiple syntax-directed and goal-directed transformations, and a resource monitor, showing current available resources.

Both the approach as a whole, and each individual transformation, are source-to-source. Keeping the design at the source level has the advantage of avoiding complications of compiling to hardware description languages; however, we cannot take advantage of any optimizations that can be done at lower levels and rely on downstream tools to provide them. The purpose of our approach is to complement lower-level tools by customizing optimizations at a high level, rather than replacing the lower-level tools.

Our design meets the requirements as follows:

Allow multiple transformations: Separating each transformation into independent units allows multiple transformations to be used. By requiring as little as possible from each transformation, we allow multiple kinds of transformation to be used.

Support common and domain-specific transformations: Previous work shows that combining common and domain-specific transformations together is sufficient to optimize realistic benchmarks. ROSE already provides a library of optimizations including loop optimizations; we reflect these into our tool. Our tool allows custom transformations to extend those in the library.

C. Implementation

Our implementation is based on the ROSE framework with a domain-specific language for syntax-directed transformation.

TABLE I
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
GP3	Power optimization
Syntax-directed	**Description**
LM	Loop merging
LC	Loop coalescing
DE	Decompose expressions with the 3-address rule
RMD	Remove "$\times/\div$" operations
RF	Reduce fanout of variables
Par	Parallelize independent statements
Pipe	Pipeline innermost loop

We use the ROSE framework because it is fairly mature, has comprehensive support for C / C++, and contains many existing features for program transformation including features for function inlining and loop unrolling.

ROSE is written in C++, requiring developers to use C++ to build their compiler and add custom passes written in C++. This approach is powerful, because all the facilities of ROSE are available, and efficient, because the approach is compiled using a conventional optimizing C++ compiler. However, writing a custom compiler and adding custom passes require extensive knowledge of ROSE.

In our approach, we adapt ROSE to be scripted by the Python scripting language [15]; other scripting languages like Lua could equally be used. We choose Python in particular because it is well supported by the Boost Python library, a quasi-standard way of reflecting C++ objects and classes into Python [18]. Using a scripting language to control compilation allows designers to take advantage of all the language facilities for control and iteration. A further advantage of scripting languages is that the transformation engine can be used to extend the scripting language command-line interpreter, meaning that the user can experiment with interactively applying transformations to the program. If a good sequence of transformations is found, the user can save the sequence in a script, allowing the sequence to apply to other designs without interaction.

While the resulting compiler is slower than one developed using ROSE directly, most of the compiler run time will be taken up by transformations, rather than the transformation engine choosing which transformation to apply. Should the transformation engine logic prove to be too slow, we note that scripting languages are slowly progressing to just-in-time compilation, which will improve the speed of compiled code while retaining the convenience of an interpreter.

Our implementation meets the remaining requirement – *playback for verification and reuse*. Before each step, the engine records the following information into a log file: the current design after preceding transformations, the name of next step, and the parameters to be used. This is all the information needed to play the step back later, and is done transparently and automatically. Each custom transformation is responsible for recording its own parameters and anything else needed to recreate the step later. We design our tool so that as much of this as possible is done by the transformation engine by default.

TABLE II
TRANSFORM PARAMETERS USED IN EXAMPLE.

Params	**Type**	**Description**
numLoop	int	number of loop levels
loopBound	int Array	upper bound of each loop after normalization
loopParallelizability	bool Array	1: loop parallelizable, 0: otherwise
numArray	int	the number of arrays
numRAMBlock	int	number of on-chip RAM blocks
blockSize	int	size of each RAM block
memBandwidth	int	memory bandwidth
numMultiplier	int	number of on-chip multipliers
numCompSteps	int	number of computation steps
stepResource	int Array	resources required in each computation step
notAlign	bool	1: data are not aligned between storage and computation, 0: otherwise
recII	int	dependence constraint on initiation interval of pipelining

D. Example

We show an example script for the matrix multiplication benchmark; the results section shows the performance of the resulting designs for this and other benchmarks. Table I shows the library of available transformations; table II shows transformation parameters. The sequence we apply is: GP1 - DE - RF - Par - GP2. Python pseudocode for this sequence is:

```
1: InDesign=parse("matmult.c")
2: Ini_design_params=[("numLoop1", 3),
  ("loopBound1", [64, 64, 64]),
  ("loopParallelizability1", [1, 1, 1]),
  ("numArray1", 3), ...]
3: Platform_params=[("numRAMBlock1", 168),
  ("blockSize1", 16384), ("memBandwidth1", 32),
  ("numMultiplier1", 168)]
4: (des1, Out_design_params)=GP1(InDesign,
  Speed, Ini_design_params, Platform_params)
5: des2=decomposeExpressions(des1)
6: des3=reduceFanout(des2)
7: des4=parallelize(des3)
8: Ini_design_params=[("loopBound2", [64]),
  ("loopParallelizability2", [1]),
  ("numCompSteps2", 1), ("stepResource2", 1),
  ("numArray2", 2),  ("notAlign", 0),
  ("recII", 1)]
9: Platform_params=[("memBandwidth2", 32),
  ("numMultiplier2",
  168-param("numMultiplier1"))]
10: (OutDesign, Out_design_params)=GP2(des4,
  Speed, Ini_design_params,
  Platform_params, innermostLoop, tree)
11: unparse(OutDesign, "out.c")
```

where numbered source lines work as follows:

1) The input file is parsed into a variable `InDesign`.
2) parameters for the GP1 model are loaded into a variable `Ini_design_params` as a list of key-value pairs. Each transformation is responsible for checking its parameters are correct. Note that values can be integers, booleans, strings or arrays of other values; we omit some of the parameters for space reasons.
3) platform parameters are similarly put in another variable.
4) the GP1 transform is applied to `InDesign`, yielding

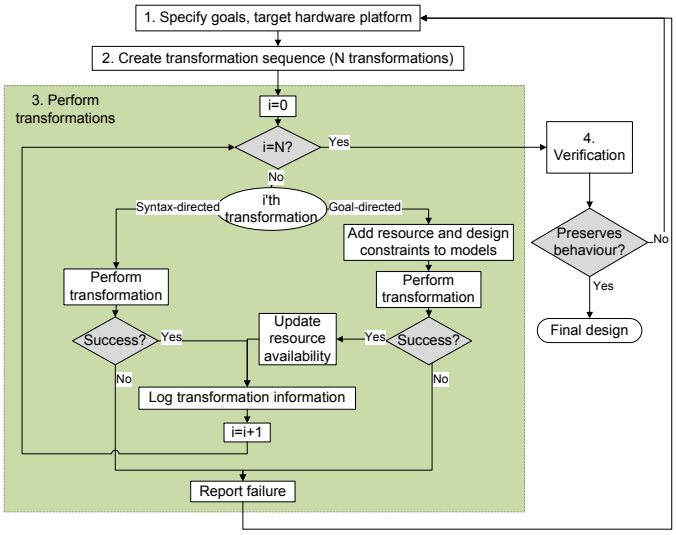

Fig. 2. Flowchart for applying goal-directed and syntax-directed transformations. The counter `i` can be used by the transformation engine to limit the number of transformations applied, to limit the total compilation time. It is used internally in the logfile to record which transformation applies at each step.

`des1`.

5) three syntax-directed transformations are applied to `des1`, yielding `des4`; none of these has any parameters.
8) parameters for the second goal-directed transform are put in a key-value list.
9) platform parameters for GP2 are stored. Note that the value for `numMultiplier2` is calculated by using the `param` function to retrieve the value of parameter `numMultiplier1`.
10) the GP2 transform is run.
11) finally, the design is unparsed to an output file.

Each source code line corresponds to one iteration of the loop in step 3 of Fig. 2.

IV. Complex composition

A. Requirements

Complex composition extends and refines the requirements for simple composition to:

- Allow multiple goal-directed transformations.
- Allow further ways to compose transformations.

In more detail:

Allow multiple goal-directed transformations: we restrict simple composition to one goal-directed transformation, which effectively restricts the overall optimization goal to that of the goal-directed transformation. With multiple goal-directed transformations, multiple optimization goals can be pursued; for example, sequencing goals for speed then power gives the lowest power design meeting a speed constraint. The goals achieved by the former transformation become the constraints for the current one, so transformations optimize designs without affecting former achievements.

Allow more ways to compose transformations: using multiple goal-directed transformations requires more ways to compose transformations together. To allow this, we require each transform to have a notion of success or failure: success means the transformation has been applied without errors; failure means it could not match pattern, or the design is not in the correct format (syntax-directed), or it could not achieve the goal (goal-directed). This allows more ways of composing transformations:

Conditional sequence: this requires each transformation to succeed before the next is applied. If a transform fails, the whole sequence could fail (where the design is left unchanged), or the design after the last success could stand. Two kinds of conditional may be used: *and-then* and *or-else*; and-then requires that every transformation in the sequence succeeds; or-else continues until the first transformation that succeeds.

Competitive: given the same input, run two transformations and compare their output; the one with better metrics is chosen.

Parallel: apply transformations simultaneously to separate, independent parts of design, enabling the use of computing clusters to speed up design space exploration.

Parameterization: some transformations can be parameterized to give designers finer control. The parameters could be values, such as design parameters and platform parameters, or other transformations.

Pre- and post-requisites: one transformation can use another to pre-transform its input, *e.g* loop normalization before many loop optimizations. This simplifies many transformations.

Joining transformations into single unit: analogous to joining expressions and statements into a procedure or function; for example, 2D loop tiling can be implemented using 1D loop tiling with loop interchange. Using one transformation to implement another saves development effort and eases transform verification. This can combine common sequences for particular domains or inputs together.

B. Design

To meet the first requirement, allow multiple goal-directed transformations, we extend the parameters of goal-directed transformations to allow finer control:

- *Force transformation*: (optional) designers specify transformation to perform, even though characteristics targeted by the models are not present in the current design. For example, the geometric programming model [19] for mapping the MapReduce pattern can apply to a design not exhibiting its required pattern, in order to pipeline the design.
- *Output parameters*: for conditional sequences, each transformation should update a parameter indicating whether it successfully applied or not.
- *Constraints*: for a sequence of goal-directed transformations, the transformations may be combined into one if their models are compatible; for example, two geometric

programming (GP) models may be combined into another which is also a GP model.

- *Output design parameters*: outputs of models are the parameters of the optimized design found, for example, chosen data reuse options, loop partition scheme, operation pipelining scheme; these show how the design was transformed.

To meet the second requirement, we extend the design of our framework to allow complex composition of transformations. Our transformation engine uses the following strategy for sequence, conditional sequence and competitive compositions of goal-directed and syntax-directed transformations:

(1) Specify design goals and the target hardware platform.
(2) Choose transformations from the library, specify transformation parameters, and determine the transformation composition.
(3) Perform transformations.
  (a) **if** Sequence is chosen (designers have good understanding of the input code), **then**
    (i) Before each goal-directed transformation, add resource constraints to the model based on the information shown by the resource monitor, and if this transformation will work on the same code part where the previous goal-directed transformation ran, then the design goal achieved in the previous goal-directed transformation is added to the model as constraints.
    (ii) After each transformation, record the transformation status (success or failure) and the transformation scheme in the log file and the resource monitor updates the availability of resources after each goal-directed transformation, and go to the next transformation.
    (iii) Pass the final transformed code and the log file to the verification step (4).
  end if
  (b) **if** Conditional is chosen (designers have basic understanding of the input code), **then**
    (i) As step (i) in (a).
    (ii) After each transformation, check the status of the transformation.
    if success **then** as step (ii) in (a).
    else stop the transformations, report the failure to designers, and go back to step (1).
    end if
    (iii) As step (iii) in (a)
  end if
  (c) **if** Competitive is chosen (designers have multiple transformation sequences in mind), **then**
    (i) Execute each sequence as Conditional, but if failure, assign an extreme value (maximum for minimization or minimum for maximization) to the objective of the transformation sequence.
    (ii) Compare the design objectives of the transformation sequences and choose the best sequence when all sequences finish.
    (iii) As step (iii) in (a).
  end if
(4) Verify transformed design.
  if all transformations are successful and legal **then**
    Continue the compilation
  else
    Suspend the compilation, report to designers and go back to step (1), where designers have chance to look through the log file and rearrange transformations, probably removing problematic transformations.
  end if

This can be readily extended for further ways of composing transformations. Fig. 2 shows the flowchart for this strategy.

C. Implementation

In addition to sequential composition, advanced composition also supports conditional sequence (and then and or else), competitive, parallel (CoSy), parameterized, pre- and post-requisites, and joining transformations into a single unit (our goal-directed transformations can combine the goals into a joint goal if mathematical models are compatible.

D. Example

An example of joining goal-directed transformations is the goal-directed transformation GP2, which is obtained by combining two separate models for optimizing pipelining and MapReduce. The advantage of this combination over the separate approaches is that we could find more optimal designs which may not be reached in the separate approaches.

V. Results

We follow the control strategy described above and apply several transform compositions, with different input parameters, to five benchmarks:

- multiplication of two 64 $\times$ 64 matrices (MAT64),
- Sobel edge detection (Sobel) [20],
- the full search motion estimation algorithm (FSME) [21],
- finite impulse response (FIR) filters of 3, 5 and 11 taps,
- Monte Carlo simulation (MCS) for Asian Option pricing [22].

We apply three transformation sequences to the first three benchmarks respectively; we choose each sequence for its benchmark. For MAT64 and FIR, we also use competitive composition to investigate the impact of different transformation orders and different input parameters. Moreover, Sobel is transformed to achieve the most power-efficient design meeting the speed constraints specified by designers. The goal-directed transformations used in our approach also allow a special case of composition: joining multiple goal-directed transformations into a single mathematical model. We compare joined and separate transformations applied to MCS. All experimental results are obtained on our target platform, containing a Xilinx XC2V8000 FPGA with four external static RAM banks.

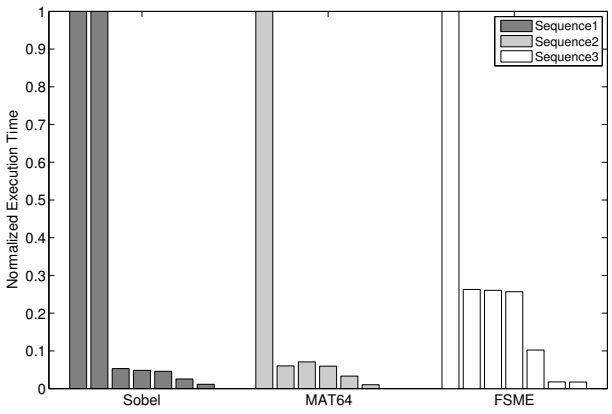

Fig. 3. Execution times on the target platform (Xilinx XC2V8000) after successive transformations, normalized so initial design takes unit time.

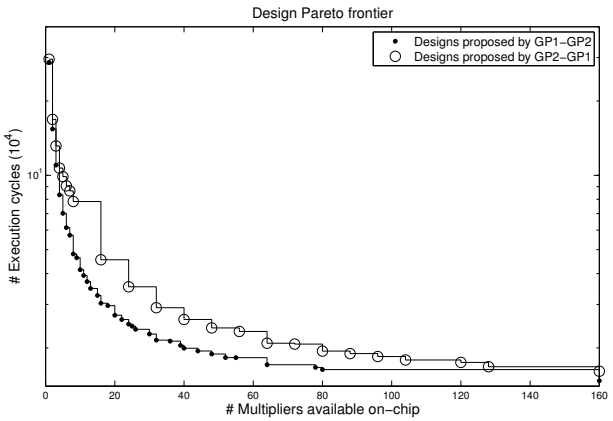

Fig. 4. Results for MAT64.

TABLE III
LINEAR REDUCTION VS TREE REDUCTION FOR FIR

Designs	Exe time (ms)		# FPGA 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

We optimize the speed of Sobel, MAT64 and FSME algorithms on the target platformby respectively applying three transformation sequences from the transformation library (table I):

- **Sequence**1: LM–GP1–DE–RF–Par–GP2;
- **Sequence**2: GP1–DE–RF–Par–GP2;
- **Sequence**3: GP1–DE–RMD–Par–Pipe–LC.

Each transformation sequence is customized for a specific benchmark. After verifying that the functionality is preserved, we translate the design into Handel-C, synthesize, map and place and route the transformed designs onto the target platform. We currently translate the designs manually; this could be done mechanically, using a tool such as C-to-Verilog [17]. Fig. 3 shows execution times, normalized so the initial design takes unit time. For each benchmark, the first bar corresponds to the initial design, the second bar to the design after the first transformation in the sequence, and so on. For example, the second bar for Sobel is the design after loop merging; the third bar is the design after loop merging and GP1 goal-directed transformation. We see that LM does not improve the speed of Sobel; however, LM simplifies the Sobel loop structure and thus facilitates the following GP1 transformation [8]. Fig. 3 shows similar cases. Overall, the transformation sequences reduce the execution times on our target platform for Sobel, MAT64 and FSME by factors of 88 times, 98 times and 57 times respectively.

Competitive transformation composition allows designers to experiment with different transformations. In Sequence2, the goal-directed transformation GP1 can run before GP2 (GP1-GP2), or in the reverse order (GP2-GP1). To see the impact, we apply Sequence2 with the two transformations in both orders to MAT64; the results are shown in Fig. 4, where the y-axis is in logarithm scale. This figure shows the speed-optimized design Pareto frontiers generated by Sequence2 with GP1-GP2 and GP2-GP1 respectively, under different multiplier resource constraints. After the transformation sequence with GP1-GP2 the total execution cycles of the MAT64 designs is up to 2.7 times less (in the worst case) than after the sequence with GP2-GP1 under the same multiplier constraint. However, for another design metric, required on-chip RAM blocks (not shown), designs chosen by the GP2-GP1 transformation sequence use about 5 times, on average, fewer RAM blocks to achieve the same design speed as ones generated by the GP1-GP2 sequence. Therefore, if there are enough memory resources, then the transformation sequence with GP1-GP2 would be chosen; otherwise the sequence with GP2-GP1 may be better, depending on the design goal and parameters specified by designers.

Moreover, the GP2 goal-directed transformation allows designers to specify which reduce structure – tree or linear – to use in the reduce phase of the MapReduce computation [19]. Different structures result in different system latency, throughput and resource usage for different applications. Designers can set transformation-specific parameters to specify which reduce structure to use. Table III shows the results of applying the GP2 transformation to FIR filters with 3, 5 and 11 taps. We can see that in all three cases, GP2 results in designs with similar execution times for either reduce structure, but using a linear structure can reduce the number of slices by up to 2.4 times. Hence designers would choose the transformation with the linear reduce structure. For other applications, this may not be true, so designers would need to apply competitive composition.

In the above examples, designers may specify the design objective, choose transformations and specify the target hardware platform. Our approach lets designers provide additional constraints to some transformations.

The goal-directed transformation GP3, shown in Table I, allows designers to specify target design speed when the transformation objective is power optimization. We apply the GP3 transformation to Sobel together with the transformations

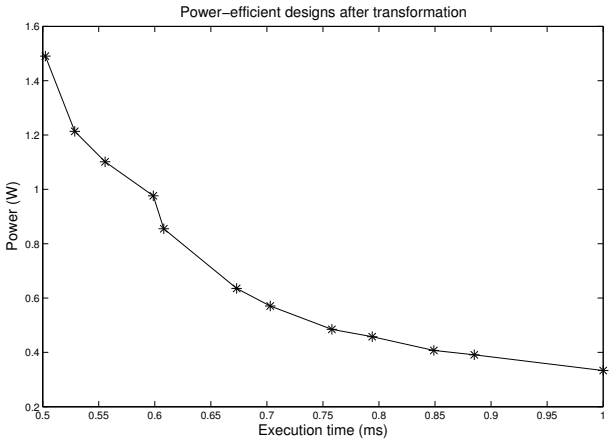

Fig. 5. Power reduction transformation for the Sobel benchmark.

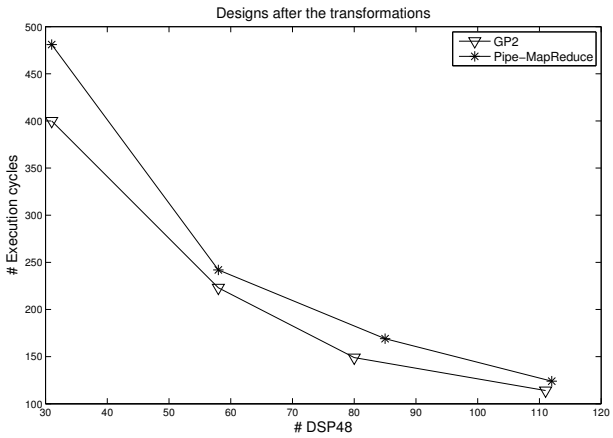

Fig. 6. Results for the MCS benchmark for Asian option pricing.

in Sequence1, but without GP1 and GP2. Fig. 5 shows the resulting power-efficient designs, for different execution time constraints. When the execution time constraint is tight, more parallelism is needed and thus more resources used, resulting in larger power consumption. When the target execution time is less than 0.5 ms, the transformations cannot achieve the goal at all, and report this to designers. If designers still want to achieve this goal, then they must add other transformations or choose a platform with more resources.

Our approach also supports joining goal-directed transformations with compatible mathematical models. Fig. 6 shows the results when we transform the MCS algorithm for Asian Option pricing by optimizing pipeline and MapReduce separately (Pipe-MapReduce) and at the same time (GP2). It shows that the joined transformation GP2 finds more optimal designs under the same constraints, resulting in up to 1.22 times speed improvement, or up to 1.27 times reduction in DSP48 block usage.

VI. 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 tool based on the Python language and the ROSE compiler framework, which supports both syntax-directed transformations such as loop coalescing, and goal-directed transformations such as geometric programming. Current and future work includes supporting further automation, such as automatic choice of transformations and profile-driven optimization, as well as verifying the correctness of the transformations.

Acknowledgment

The support of the FP6 hArtes project, EPSRC and Xilinx is gratefully acknowleged.

References

[1] Mentor Graphics, "Catapult C synthesis datasheet," http://www.mentor.com/products/esl/high_level_synthesis/catapult_synthesis/upload/Catapult_DS.pdf, accessed March 2010.
[2] Impulse Accelerated Technologies, "Impulse CoDeveloper C-to-FPGA Tools," http://www.impulseaccelerated.com/products_universal.htm, accessed March 2010.
[3] Agility Design Solutions, http://www.agilityds.com, accessed March 2010.
[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] J. R. Cordy et. al, "The TXL programming language," November 2007, version 10.5, http://www.txl.ca/docs/TXL105ProgLang.pdf.
[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] R. Ierusalimschy, *Programming in Lua, Second Edition*. Lua.org, 2006.
[17] C to Verilog, "http://c-to-verilog.com/," accessed May 2010.
[18] "Boost.python documentation," version 1.42, accessed January 2010, http://www.boost.org/doc/libs/1_41_0/libs/python/doc/index.html.
[19] Q. Liu et al., "Automatic optimisation of mapreduce designs by geometric programming," in *Proc. Int. Conf. on FPT*, 2009, pp. 215–222.
[20] http://www.pages.drexel.edu/~weg22/edge.html, accessed 2006.
[21] V. Bhaskaran and K. Konstantinides, *Image and Video Compression Standards: Algorithms and Architectures*, Norwell, MA, USA, 1997.
[22] http://www.interactivesupercomputing.com/success/pdf/caseStudy_financialmodeling.pdf, "Financial Modeling – Monte Carlo Analysis," accessed 2009.