Copyright © 1995, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.
A METHODOLOGY TO APPLY OPTIMIZING
TRANSFORMATIONS

by

Shan-Hsi Huang and Jan M. Rabaey

Memorandum No. UCB/ERL M95/32

1 February 1995
A METHODOLOGY TO APPLY OPTIMIZING TRANSFORMATIONS

by

Shan-Hsi Huang and Jan M. Rabaey

Memorandum No. UCB/ERL M95/32

1 February 1995
A METHODOLOGY TO APPLY OPTIMIZING TRANSFORMATIONS

by

Shan-Hsi Huang and Jan M. Rabaey

Memorandum No. UCB/ERL M95/32

1 February 1995

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720
A Methodology to Apply Optimizing Transformations

Abstract - Transformations for algorithm optimization have shown to be effective in high-level synthesis. When a large number of transformations are available, it is always difficult to determine which transformations should be applied and in what order. In this report, we propose a methodology which clearly addresses these issues and organizes them in a systematic fashion. The proposed methodology is composed of a set of sub-tasks including bottleneck identification (why transformations should be applied), algorithm partitioning (which parts of an algorithm should be transformed), transformation prediction/selection (which transformations to apply), transformation ordering (the order in which the transformations are applied), and transformation execution (how to apply the selected transformations). A framework based on this methodology and aimed at the optimization of speed, area, or power consumption of custom DSP designs, is under development. Assisted by such a framework, designers can easily and quickly to apply a variety of transformations to explore the algorithmic design space to reach better designs.
A Methodology to Apply Optimizing Transformations

1 Introduction

Optimization is one of the most important tasks in the design process. This is especially true at the algorithmic level. An algorithm that only specifies the functionality of an application often has high degrees of abstraction which provides great freedom to improve the quality of the design implementation. Using transformations for algorithm optimization has proven to be effective and important in high level synthesis. The transformations include those explored in software compilers as well as specific ones for custom ASIC design. Some examples in the former set include common sub-expression elimination (CSE), strength reduction, loop invariant code motion, and a variety of for-loop transformations. The latter set contains retiming, pipelining, and time-loop transformations. In addition, algebraic transformations are of critical importance in DSP applications due to their computation-intensive nature.

In the past few years, many approaches have been developed for using transformations towards a variety of goals ranging from speed [1][2][3][4][5][6][7][8][9][10], area [11][12][13][14], power [16], and memory[17]. Most of these approaches only consider individual or small sets of transformations. Because an individual transformation usually has a limited application space, the reachable improvement range is often restricted. Integrating a large number of transformations can dramatically enhance their effectiveness. Unfortunately this also significantly increases the complexity of the optimization process.

When a large number of transformations are available, determining which parts of the algorithm (where) should be transformed, which transformations could be effective, and what is the appropriate order (when) to apply certain transformations is a non-trivial task. These issues are related to the designer’s algorithm, the constraints (and goals), and the considered transformations. In this report, we propose a methodology to address these issues (Section 2). This methodology easily translates into a generic framework, whose structure is discussed in Section 3. Such a framework allows a designer to readily and easily use a large set of transformations to explore the algorithmic design space. The report is concluded with a discussion of future development and a summary.
2 Methodology

The basic concepts of the methodology to apply transformations for algorithm optimization are illustrated in Figure 1. The inputs of the optimization process are the designer's algorithm represented as a control/data flowgraph (CDFG), the design constraints ranging from time, area, to power, and a predefined transformation set. This methodology is composed of a set of sub-tasks including bottleneck identification (why transformations should be applied), algorithm partitioning (which parts of the algorithm should be transformed), transformation ordering (the order in which certain transformations are applied), transformation prediction and selection (which transformations could be effective and should to be considered), and transformation execution (how to apply the selected transformations).

![Figure 1: Methodology for transformation-based optimization](image)

A design produced directly from a given algorithm instance might not meet all specified constraints or may have a prohibitive cost. The factors that violate the constraints or those components that yield the high cost are called the bottlenecks of the design. For example, if a design
cannot meet the timing constraint (sample period for DSP applications), the execution time is designated as the bottleneck. The resources that dominates the area are the bottlenecks when area constraints are not met. The bottleneck resources could be functional units (multipliers, ALUs, etc.), registers, interconnect, and memory. Similarly, the bottlenecks for power are those resources that consume significantly more power than others. Given the CDFG and the design constraints, the bottleneck identification module locates the potential bottlenecks. The goal of the optimization process is then to apply those transformations that specifically address the identified bottlenecks.

To achieve that goal, the optimization process can resort to a predefined set of transformations. The order in which these transformations are applied has an important impact on their effectiveness. The task of the transformation ordering module is to establish an appropriate ordering among the transformations. Because the transformation set determines the scope of the reachable design space and thus the potential improvement range, it is desirable that the set is sufficiently large. With a large transformation set, every algorithm is subject to a huge number of potential transformations-permutations. Reducing the search space and thus improving the efficiency of the optimization process is thus essential. One pruning technique to reduce the search space is algorithm partitioning — consider only the subgraph of the CDFG that are related to the currently selected bottleneck. The transformations that are not applicable in the subgraph are avoided because they have no chance to improve the bottleneck. In addition to the algorithm partitioning, the potential improvement of a transformation can also be used to prune the transformation space. Those transformations that have little potential to improve the bottleneck are avoided. This can be achieved with the help of the transformation prediction — predict the potential impact of a certain transformation. Based on the potential of the transformations, the transformation selection module selects a small set of transformations that are capable to optimize the given bottleneck.

After the ordering, partitioning, prediction, and selection, we have determined which part of the CDFG should be transformed, which transformations to apply, and the order in which they are applied. The last step is to execute the transformation task. After the execution, the transformed CDFG is sent back to evaluate the status. This optimization process is repeated until the constraints are satisfied or no further improvement can be obtained. In the following, we will discuss each module in more detail.
2.1 Transformation set

Since the scope of the design space to be explored is determined by the transformation set, it must be reasonably large and diverse. The possible transformations consist of algebraic transformations (associativity, distributivity, reverse distributivity, commutativity, algebraic identity, algebraic inverse, constant folding, constant multiplication expansion, and a few other specific ones), temporal transformations (retiming, pipelining, time-loop unfolding), loop transformations (loop unrolling, loop merging) and some generic transformations (common sub-expression replication/elimination, dead code elimination, loop invariant code motion).

2.2 Bottleneck identification

The prime responsibility of the bottleneck identification module is to identify the potential bottlenecks. It has been shown that there exist strong correlations between the performance metrics of a design and a number of structural properties of the algorithm [18][16]. For example, the length of critical paths is an accurate measure for the lower bound of the execution time, the concurrency is highly related to the chip area, and power consumption correlates to the number of access (count). These high-level properties can be used to derive a set of prediction models which can be used to identify the bottlenecks.

According to the design constraints, there could be a variety of bottlenecks, each of which may need different transformations. Simultaneously optimizing all bottlenecks in a design is difficult. A divide-and-conquer strategy is suggested. The bottleneck identification module picks the dominant one and defers others to later iterations. This allows the bottlenecks to be solved one by one. A general scenario to handle different bottlenecks is to assure a feasible solution first (e.g. satisfying the time constraints) and minimize the design cost (area or power, according to designer’s preference) next.

In the divide-and-conquer strategy, the identification module is also responsible for the control of the overall optimization flow and to ensure that all the potential bottlenecks are addressed. To accomplish this, the module must have the capability of memorizing the history of bottlenecks, actions taken for optimization, and improvements. The module evaluates the solution after each iteration — if a new version is not acceptable due to too much overhead (side effects), the module will either provide feedbacks to the transformation selection module to adjust the selections (e.g. avoid certain transformations) or step back to the previous CDFG.
2.3 Algorithm partitioning

Based on the identified bottleneck, the algorithm partitioning module extracts the trouble spots of a given CDFG. For instance, if execution time is the bottleneck of a design, those paths with the lengths longer than the sample period would be the targets for speed optimization (critical path reduction). The transformations will be applied only to the extracted subgraph. This reduces the transformation space in the sense that the transformations that are not applicable to the subgraph are avoided.

2.4 Transformation ordering

When a set of transformations are available, the order in which they are applied often affects their effectiveness. One approach to address the transformation ordering is the enabling principle [6]. There typically exist only a few transformations that can directly improve a given bottleneck. They are called kernel transformations [8]. However, those kernel transformations are often not sufficient due to their limited application space. Usually there exist some other transformations that can enable the applicability of a certain kernel transformation, and are therefore called the enabling transformations. The enabling relationship of transformations can be used to establish an ordering.

2.5 Transformation prediction

The transformation prediction module is to predict the potential improvement of transformations and their possible side effects in order to help to do the selection. The prediction module takes as input the bottleneck, the partitioned CDFG, and an ordered set of transformations. The ordered transformation set identifies kernel transformations as well as the dependency relationships of the transformations. Since only kernel transformations can directly affect the bottleneck, their potential improvements are of a major concern. The potential improvement of a kernel transformation depends on its applicability and capability. For example, constant multiplication expansion can be evaluated by the availability of linear multiplications. Associativity to improve the utilization of multipliers can be evaluated by the multiplication clusters in the partitioned CDFG. If a design has no loops (e.g. digital filters), all loop transformations are of no use.

Another useful information provided by the prediction module is the possible side effects of a transformation. This information can be used to degrade a kernel transformation or to unselect an enabling transformation. For example, the potential side effects of expanding a constant multipli-
cation are newly introduced additions/shifts and a longer computation time. These side-effects can be quickly predicted with the values of constant multiplicands (e.g. number of 1’s in the binary representation).

2.6 Transformation selection

Based on the predicted performance of the transformations, a small set of kernel transformations with high priority (high potential improvement plus low side effects) are selected for the final execution. Since enabling transformations are used to enable kernel transformations, they should be applied in a demand-driven fashion (to avoid the redundant enablers). There is no need to pre-select them. But if an enabling transformation potentially has negative side effects, it can be inactivated (unselected) to avoid the overhead. The goal of the selection module is to choose a small set of kernel transformations as well as unselect some enabling ones.

2.7 Transformation execution

After the bottleneck identification, algorithm partitioning, transformation ordering, and transformation prediction/selection, the bottleneck together with the partitioned CDFG and the selected transformation set are passed to the execution module to perform the transformation task. The execution module applies the selected transformations in the determined order onto the partitioned subgraph. The cost function in the optimization process is provided by the bottleneck. Generally speaking, two classes of transformation-application techniques can be discerned — global and local-move-based optimization techniques. The global approaches typically rely on analytical or heuristic approaches and tend to be more efficient and powerful. Local-move-based optimization techniques, on the other hand, are more generic. Examples of the latter are simulated annealing, exhaustive search, or steepest descent method. The execution module should allow both of them, but may give a preferential treatment to a global approach, if one exists in the library that matches the cost function and covers the transformations to be applied. The global approaches have a higher priority due to their efficiency and prowess. A framework with such an execution module provides an unified environment to integrate a variety of global approaches, and can help users to select the appropriate ones to apply.

On the other hand, if there are no global approaches available, the local-move-based optimization techniques are used instead. The selection of the techniques depends on the size of the search space, the features of the selected transformations, as well as user’s preferences. Although generic
optimization techniques may not be that efficient, the search space is expected to have been dra-
matically reduced by the partitioning and selection modules.

In order to avoid the redundant transformations, the enabling transformations should be applied
in a demand-driven mode: only if demanded by the kernel transformations. Their application is
somewhat more complex than that of the kernel ones in the sense that they depend on the applica-
bility of other transformations. There are at least two possible ways to approach this problem. One
is using the postponing principle proposed by [8]. The basic idea is to relax the conditions under
which a kernel transformation can be applied. Once a kernel transformation is selected but cannot
be applied by itself, the appropriate enabling transformations are invoked. Another approach is to
use composite moves such as described in [12].

3 Transformation Framework

Based on the methodology, the structure of a generic transformation framework is established
(Figure 2). The framework takes as input a CDFG and a number of user-defined design con-
straints. A set of structural-property-based prediction modes (P-models) and a transformation
library (T-lib) are predefined. The transformations in the T-lib are precharacterized into an ordered
set (manually), which identifies the kernel transformations for a certain P-model as well as the
dependency relationships of the transformations. The structural properties of a given CDFG are
extracted by the property extractor (P-extractor). The bottleneck analyzer (B-analyzer) uses the
predefined P-models and the structural properties to identify the prime bottleneck, which is then
passed to the transformation manager (T-manager) (Figure 3). The T-manager consists of the
algorithm partitioner (A-partitioner), transformation analyzer (T-analyzer), and transformation
selector (T-selector). The A-partitioner locates the trouble spots in the CDFG. The T-analyzer pre-
dicts the potentials of the transformations, and the T-selector suggests an appropriate set to apply.
Finally, the bottleneck, the partitioned CDFG, and the selected transformation set are passed to
the transformer. After execution, the transformed CDFG plus the executed actions are sent back to
the B-analyzer to verify the result. If nothing better can be achieved, the best solution is returned
to the user. Otherwise, the B-analyzer analyzes the history of bottlenecks/improvements and picks
up one CDFG to optimize further. (It is often the most recent solution unless it was rejected). If
possible, some feedback is provided to T-manager to adjust its selections. Of course, a new bottle-
neck is identified for the next iteration.
Figure 2: Structure of the transformation framework

Figure 3: Structure of the transformation manager
4 Future work

The proposed methodology is currently being used to develop a transformation framework for optimizing speed, chip area, or power consumption of custom DSP designs. This framework is named as TAO (transformations for algorithm optimization). One thing worth to mention is that the TAO system is not only intended to be an automatic environment, but also an analysis and guidance environment. The B-analyzer and T-manager provide all the key elements as a guidance for the optimization process. A designer can take the guidance as an assistance to his/her own knowledge and experiences for design optimization (the guidance is the same as used for automation). With such a framework, a user has the full accessibility to the environment and can manually override the decisions/actions made by the framework. This feature is very useful because designers may want to keep the control to their designs at such a high level.

5 Conclusion

We have proposed a new methodology to apply algorithmic transformations for design optimization. The proposed methodology clearly addresses the issues of which transformations to apply, when to apply certain transformations, as well as where to apply the selected transformations. With this methodology, the TAO system, a methodology-based transformation framework, is currently under development. This framework can systematically choose the appropriate transformations to apply at the right place and in the right order.

Since algorithm developers seldom take into account the implementation cost, and hardware designers rarely consider the merits of various algorithm instances, there often exists a gap between the algorithms conceived by the software developers and those that designers want to use. Moreover, an algorithm might be used by different designers for different specifications over a variety of implementation platforms. It is nearly impossible to have an algorithm well optimized for all situations. Our framework can bridge the gap by allowing hardware designers, under their specific design constraints, easily and quickly apply a variety of transformations to explore the algorithmic design space to reach better designs.
6 Acknowledgments

The authors would like to thank M. Potkonjak for valuable discussion, and Semiconductor Research Cooperation to sponsor this project (95-DC-324).

7 Reference


