TAO: A TRANSFORMATION FRAMEWORK FOR DSP ALGORITHM OPTIMIZATION

by

Shan-Hsi Huang and Jan M. Rabaey

Memorandum No. UCB/ERL M95/89

1 August 1995

TAO: A TRANSFORMATION FRAMEWORK FOR DSP ALGORITHM OPTIMIZATION

by

Shan-Hsi Huang and Jan M. Rabaey

Memorandum No. UCB/ERL M95/89

1 August 1995

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TAO: A Transformation Framework for DSP Algorithm Optimization

Abstract *- This report proposes a framework aimed at the optimization of speed, area, or power consumption of custom DSP designs through algorithmic transformations. The framework uses a generic methodology to select and order the transformations needed for the optimization of a given cost-function. This methodology combines bottleneck analysis (why the transformations should be applied), transformation ordering (the order in which the transformations are applied), algorithm partitioning (which parts of an algorithm should be transformed), transformation prediction/selection (which transformations to apply), and transformation execution (how to apply the selected transformations). Assisted by this framework, designers can easily and quickly exploit the optimizing transformations to explore the algorithmic design space to reach better designs.*

TAO: A Transformation Framework for DSP Algorithm Optimization

1 Introduction

In the past few years, numerous researchers in high-level synthesis have successfully exploited different transformations in optimizing a design with respect to a variety of objectives, including speed [1-9], area [10-16], power [17], and memory [18]. Most approaches consider only individual or small sets of transformations. Since the applicability of an individual transformation is often restricted, it is desirable to integrate a large number of transformations. While this enhances the effectiveness, it may, however, greatly increase the complexity of the optimization process.

Given a set of transformations, determining which transformations to apply, where to apply them, and in what order is a non-trivial task. A commonly used approach is to provide an interactive user interface which requires users to make all the decisions by themselves. This approach has the disadvantage of being ad hoc and might not work in cases where the cost-function is non-obvious, such as in area minimization. Another approach is to develop a dedicated algorithm (often with heuristics) which integrates all the transformations into one. This approach has little to no extensibility in terms of adding new transformations. As a large number of transformations exist, the above approaches are neither practical nor efficient.

The above issues have been studied for quite some time in the compiler world. Most of these studies have focused on loop transformations [19-21]. Whitfield [22] proposed an interesting idea to order the transformations by investigating the properties of transformations over a benchmark set. In high level synthesis, these issues were little addressed. Potkonjak [5] brought up the idea of enabling transformations for ordering. Huang [8] and Janssen [12] furthered the application of enabling transformations in a demand-driven fashion. In addition to the transformation ordering, predicting the cost and performance of transformations is another important task. Estimation helps to distinguish transformations and select appropriate ones for application. Rim [23] proposed a similar idea which employs some estimation techniques for loop transformations.

While many research efforts in high level synthesis are devoted to explore new individual transformations (or new algorithms for the known transformations), it is becoming more and more compelling to have a CAD environment that *integrates a large set of the existing transformations, has the flexibility of adding new ones,* and then *systematically makes use of them.* To our best knowledge, there exists no such environment. This report, instead of introducing new transformations, proposes one such framework, named *TAO* (Transformation for Algorithm Optimization), which systematically determines *which transformations to apply, where to apply them,* and *in what order.* This framework can help designers effectively exploit a variety of optimizing transformations to improve their designs.

To substantiate these concepts, we will use the area optimization for custom DSP chip design as an example. Area optimization using transformations has been little addressed in high level synthesis. The problem with area optimization is that the impact of a single transformation is often ambiguous; area (in contrast to time) is a global variable and local transformations often do not work. Typically, the interplay between many transformations is needed to get a substantial area improvement. For functional units, [10] and [12] proposed two different approaches - improving the resource utilization and reducing operation count. The fundamental concepts and differences will become apparent in the course of this paper. The proposed framework is also extensible to other cost-functions, e.g. speed and power.

In the next section we propose a generic methodology to apply optimizing transformations. This methodology translates into a transformation framework that is discussed in Section 3. The effectiveness of the approach is demonstrated in Section 4 with a number of examples. The paper concludes with a discussion of future developments and a summary.

2 Methodology

Figure 1 shows a generic methodology for applying optimizing transformations. The inputs to the optimization process are a designer's algorithm represented as a control/data flowgraph (*CDFG*), design constraints such as time, area, or 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 as a CDFG should be transformed), *transformation ordering* (the order in which certain transformations are applied), *transformation prediction and selection* (which transformations could be effective and should be considered), and *transformation execution* (how to apply the selected transformations).

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

To achieve this goal, the transformation module can resort to a predefined set of transformations. Since the order in which these transformations are applied has a significant impact on their effectiveness [22], an appropriate ordering among the transformations, based on the bottlenecks and the properties of transformations, is established in the *transformation ordering module*. As the transformation set determines the scope of the design space and hence the potential improvement range, it is desirable that the set is sufficiently large. This

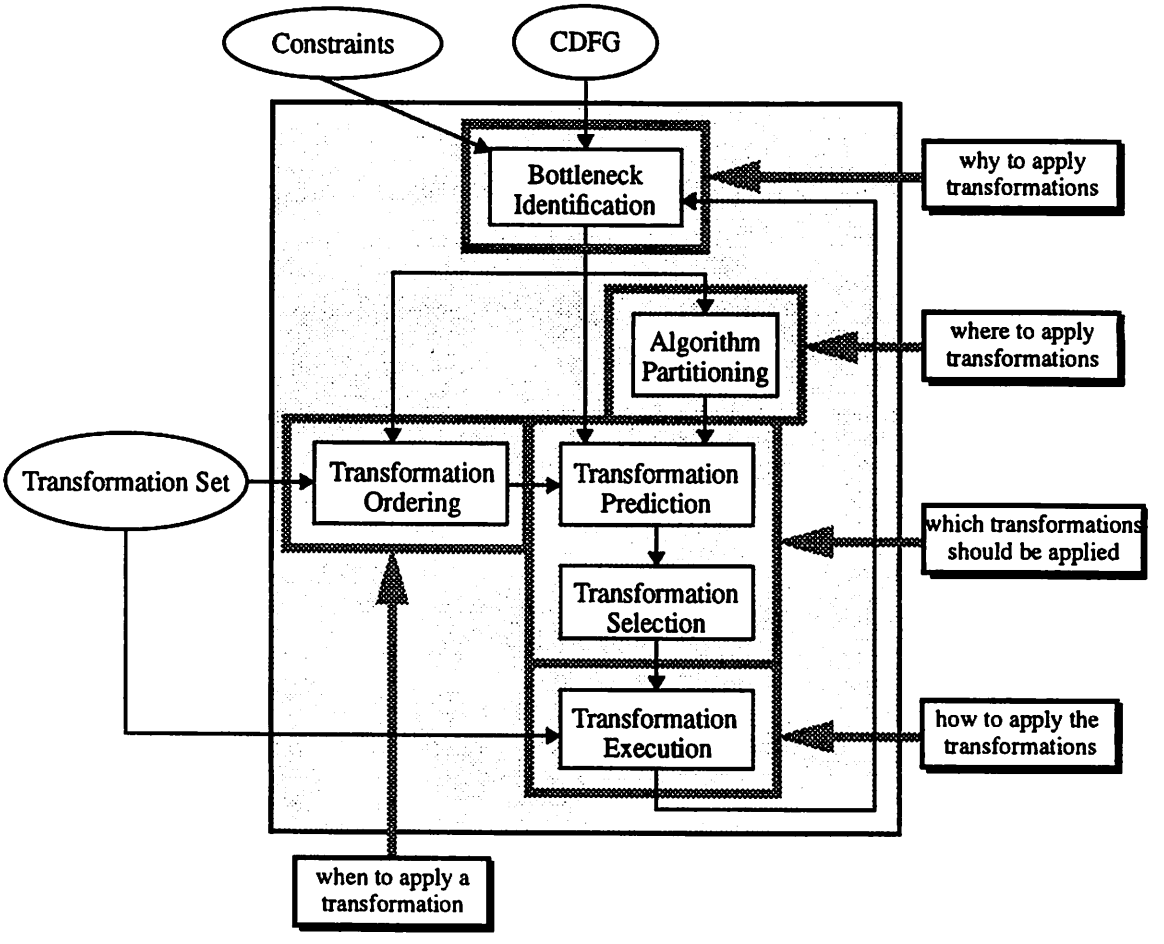

Figure 1: Methodology for transformation-based optimization

however leads to a large number of possible permutations among transformations. It is therefore important to reduce the search space so as to improve the efficiency. One technique for pruning the search space is *algorithm partitioning* — considering only the subgraph of the CDFG that is related to the currently selected bottleneck. Transformations which are not applicable in the subgraph are omitted because they cannot improve the bottleneck. Another technique for pruning a large transformation space is to predict the potential impact of transformations (*transformation prediction*) and select only the transformations that are capable of optimizing the given bottleneck (*transformation selection*). After *ordering*, *partitioning*, *prediction*, and *selection* are done, we have determined which transformations to apply, where to apply them, and in what order. The last step is executing the transformation task. After the selected transformations are applied, the transformed CDFG is fed back for re-evaluation. This optimization process is repeated until all constraints are satisfied or no further improvement can be achieved.

3 Transformation Framework

Based on the above methodology, a transformation framework, *TAO*, aimed at the optimization for speed, area, or power of DSP applications is established (Figure 2). In this framework, a set of prediction models (*P*-

models) and a transformation library (*T-lib*) are predefined (Section 3.1 & Section 3.3, respectively). The **bottleneck analyzer** takes these P-models to identify bottlenecks (Section 3.2). These P-models are also used to characterize transformations (Section 3.4). The characteristics of transformations, e.g. ordering among transformations, are represented as a set of *T-graphs* which will be discussed in Section 3.5. T-graphs are the primary object through which the modules in the framework communicate with each other. This representation enables our framework a flexibility of adding new transformations in the future.

With the identified bottlenecks and the set of T-graphs, the **transformation manager** performs *algorithm partitioning*, *transformation prediction*, and *transformation selection* (Section 3.6). It determines which transformations should be applied and where to apply transformations. The selected transformations are represented as a refined T-graph. Finally, the **transformer module** applies these transformations in an order dictated by the T-graph onto the subgraphs (Section 3.7). The transformed CDFG is then sent back to the bottleneck analyzer to evaluate the result. If nothing better can be achieved, the best solution achieved so far is returned to the user. Otherwise, a new bottleneck is identified for the next iteration.

In the rest of this section, we will discuss each module in more detail. To substantiate the concept, we use area optimization as an example. The experimental results on a number of real-life examples will follow in Section 4.

3.1 Prediction models (P-models)

Rabaey et al. [24] have shown that there exist strong correlations between the performance metrics of a design and a number of *structural properties* of the algorithm. For example, the length of critical paths is a good measure for the execution time, the concurrency is highly related to the chip area, and the number of

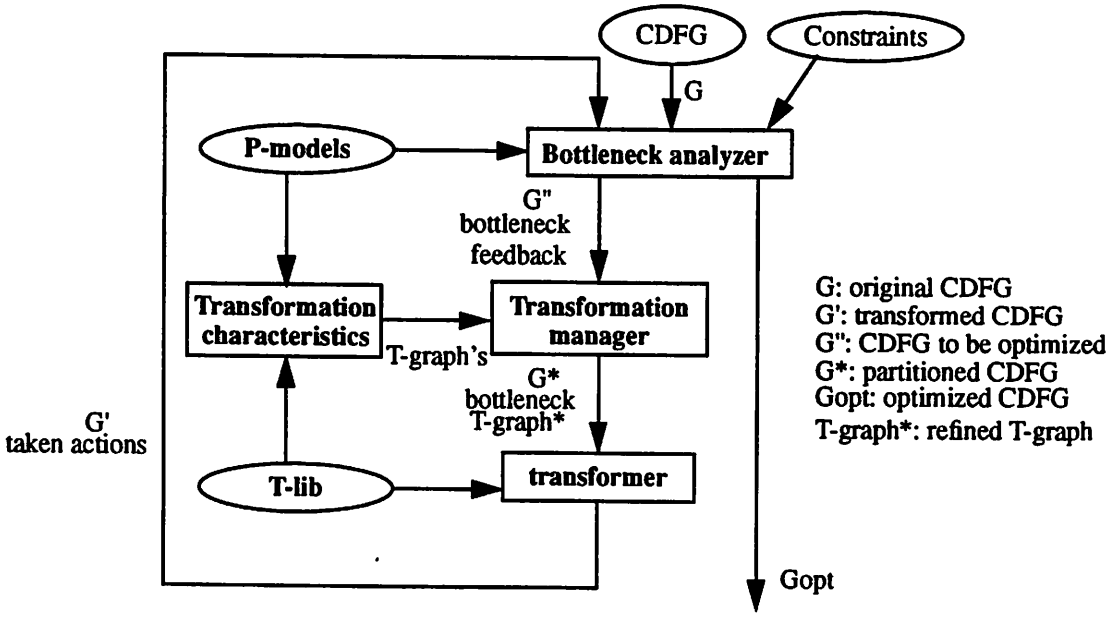

Figure 2: Structure of the TAO framework

accesses (count) correlates to power consumption. These high-level properties can be used to derive the P-models.

Our framework currently provides a set of structural-property-based P-models. These P-models are used to identify the bottlenecks as well as to characterize transformations. Besides, due to their low complexity, these models also serve as the cost functions in the transformer module. Although our framework provides the flexibility of using other more accurate P-models [27-30], this however will trade efficiency for accuracy.

For area optimization, a set of layered P-models are derived. These P-models predict the area costs of resources which can be functional units (FU's), registers, and interconnect busses. For simplicity, We will use FU's as an example. The P-model for FU's contains three submodels (Figure 3), each of which reveals different degree of information. The first layer is the simplest one. It only considers the *count* information (number of operations). This submodel determines the absolute min-bound[1] on the amount of resources. Since different types of operations usually have different costs, the second layer of the model considers both the number of operations as well as their weights. The corresponding sub-model is the *weighted sum.* A further refinement of the model is to account data dependencies as well. The submodel at this layer turns out to be *the maximal height of the distribution graph.* The concept of using distribution graphs for prediction has been proposed and discussed in [24-26, 11]. Figure 4 shows an example of a distribution graph. There are 3 multiplications in the flowgraph, and each of them takes 1 clock cycle to execute. Given the available time of 4 clock cycles, M_1 and M_2 must be executed at cycle 1, and M_3 could be in cycle 2 or 3. We assume that M_3 is *equally likely* to be executed in either cycle. It's obvious that both M_1 and M_2 need a multiplier at cycle 1. But M_3 potentially only needs 0.5 multiplier at cycle 2 and 3. This builds up a distribution graph as shown in Figure 4. The height of the distribution graph predicts the required amount of multipliers. In this example, 2 multipliers are needed.

Similar models can also be derived for other resources like registers and interconnect busses. Given the distribution graphs, the bottleneck analyzer can quickly predict the area cost of all resources and identify which resource should be optimized. The layered P-models later will be used to characterize transformations in Section 3.4.

Layered P-model		Effects
count	$\Sigma(\#op_i)$	abs-min-bound
weighted sum	$\Sigma(\#op_i * weight_i)$	weighted abs-min-bound
height of distribution graph	$\Sigma(\#\overline{op_i} * weight_i)$	utilization

Figure 3: Layered P-models

1. The absolute min-bound of a resource is defined as the minimum amount of the resource which is needed under the assumption that the full utilization can be achieved.

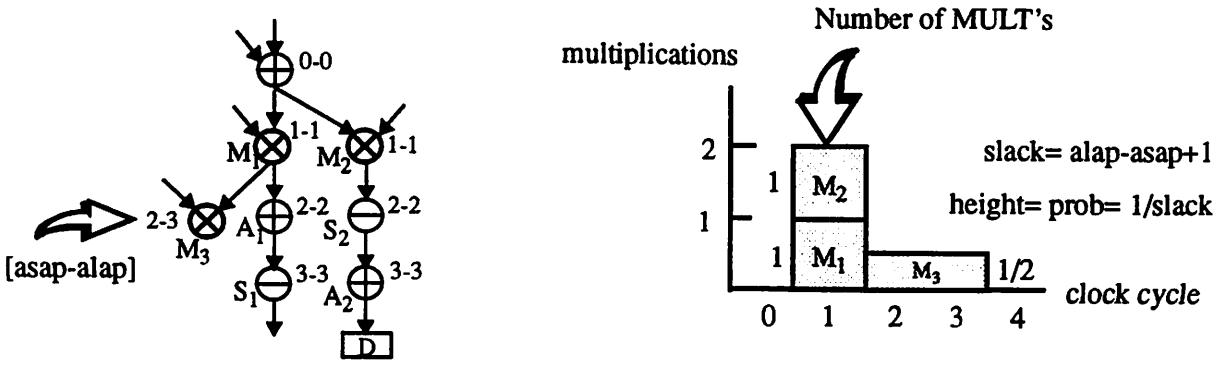

Figure 4: The distribution graph of multiplications (sample period = 4)

3.2 Bottleneck analyzer

To identify the bottlenecks, the bottleneck analyzer is associated with a set of P-models as discussed in Section 3.1. The identified bottlenecks will later be optimized using transformations. According to design constraints, there could be a variety of bottlenecks, each of which may need different transformations. Optimizing all bottlenecks in a design simultaneously is hardly possible. A *divide-and-conquer* strategy is used in our framework. The bottleneck analyzer picks the dominant one and defers others to later iterations. This allows the optimizer to eliminate the bottlenecks one by one. A general strategy to handle different bottlenecks is to assure a feasible solution first (e.g. satisfying the time constraints) and then minimize the design cost (area or power, according to designer's preference).

For area reduction, the resources that dominate the chip area are the bottlenecks of interest. Among these, the one which has the highest *potential* to get improved is selected to be the bottleneck and others are deferred to later iterations. The potential can be identified by analyzing the distribution graphs. For example, the distance between the height of the distribution graph and the absolute min-bound roughly shows the potential improvement range.

In the divide-and-conquer strategy, the bottleneck analyzer 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 analyzer must have the capability of memorizing the history of bottlenecks, actions taken for optimization, and improvements. The analyzer evaluates the solution after each iteration — if a new version is not acceptable due to too much overhead (side effects), the analyzer will either provide feedbacks to the transformation manager to adjust its selections (e.g. avoid certain transformations) or reject the new solution and step back to the previous CDFG.

One advantage of such a divide-and-conquer approach is that it supports *dynamic optimization flow.* Based on identified bottlenecks, different transformations might be applied in different orders through the optimization process. This benefit cannot be realized by any dedicated algorithm which has built-in static optimization flow.

3.3 Transformation library (T-lib)

The considered transformation set is defined in the T-Lib. Because the scope of the design space is determined by the transformation set, it should be reasonably large and versatile. Bacon [31] summarized dozens of existing optimizing transformations in software compilers. Some of them are of no use in our framework because our transformations are applied to a *CDFG* instead of *statements*. In addition, for the experimental purpose, we only consider the transformations which are most important in our application domain. For example, *algebraic transformations* and *retiming/pipelining* are of critical importance in DSP applications due to their computation-intensive nature and temporal property. The T-Lib in our framework consists 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 fusion, loop distribution*) and some generic transformations (*common sub-expression replication/elimination, dead code elimination, loop invariant code motion*).

3.4 Transformation characteristics

With such a large set of transformations, the *characteristics* of transformations can help us understand their capabilities and interactions. This is important for transformation ordering and selection. In order to characterize transformations, we relate their effects on a CDFG (algorithm) to the layered P-models proposed in Section 3.1. In the case of FU's being the bottleneck, we can classify transformations into the following categories:

(1) *operation reduction* [count] — This is the most often used technique for area reduction. Transformations in this category can reduce the number of operations, therefore the absolute min-bound. Possible transformations are common sub-expression elimination (CSE), reverse distributivity (also known as factoring), constant folding, algebraic identities, and loop invariant code motion.

(2) *operation conversion* (or *strength reduction*) [weighted sum] — This technique trades expensive operations for cheaper ones so as to reduce the overall weighted sum. This however may increase the total operation count. The weight of an operation is the module area given in the hardware library. A representative transformation for this category is constant multiplication expansion (CM).

Layered P-model		Behavior of T's	Effects
count	$\Sigma(\#op_i)$	op reduction	reducing abs-min-bound
weighted sum	$\Sigma(\#op_i * weight_i)$	op conversion	reducing weighted abs-min-bound
height of distribution	$\Sigma(\#\overline{op_i} * weight_i)$	op reordering	improving utilization

Figure 5: Layered P-models vs. Behavior of transformations (for FU's)

(3) *operation reordering* [height of a distribution graph] — This class of transformations improves the utilization of hardware resources by changing the computational order of the operations and therefore their distribution. Possible transformations in this category are associativity, distributivity, retiming, pipelining, loop fusion/distribution and loop unrolling.

(4) *none of the above*: There are also transformations that cannot directly improve the area of FU's, for example, commutativity and common sub-expression replication (CSR). They might however enable the application of transformations of classes 1-3.

With the above characterization, a simple bound on the potential improvement of each transformation can be obtained. Transformations of the operation-reduction class can reduce the operation count, therefore the absolute min-bound. Operation conversion directly reduces the weighted sum, but the total operation count is bounded. The operation-reordering transformations improve the resource utilization by reducing the height of a distribution graph, but their potential improvements are bounded by (weighted) absolute min-bound. The transformations at the lower layer have a more direct impact on the area, but those at the higher layer may lead to a better solution. Transformations should therefore be ordered in a top-to-bottom fashion. At first, operation reduction is used to reduce the absolute min-bound. If possible, operation conversion follows to trade expensive operations for cheaper ones such that the weighted min-bound is reduced. This is the best possible solution we can eventually achieve. Ultimately, operation reordering achieves the final resource requirement as close as possible to the absolute min-bound by improving the resource utilization. If the height of the distribution graph is already close to the min-bound (i.e. the utilization is very good), operation reordering is of no use and could be skipped. Similarly, if all operations have the same weight in area, e.g. ALU for all operations, operation conversion could be disregarded.

Based on the layered P-models, we characterize transformations into different categories. This helps us distinguish transformations. This layered characterization is one of the major contributions in this paper. With this characterization, we are able to order transformations in a much cleaner way and effectively predict potential improvement of a transformation.

3.5 Transformation Ordering

When a set of transformations are available, the order in which they are applied often affects their effectiveness [32, 33]. In our framework, the ordering among transformations is represented as a *transformation graph (T-graph)*, where each node stands for a transformation and a directed edge represents dependency relation between input and output node (Figure 6). Because transformation ordering varies with different objectives (P-models or submodels), different T-graph's are provided for each of them. These T-graphs are predefined in a generic fashion, and will be refined by the transformation manager, according to the design context. The final T-graph is used to drive the transformer module. T-graphs are the primary object in our framework through

which the modules communicate with the others. This representation enables our framework a flexibility of adding new transformations later.

For deriving ordering among transformations, one common approach is using the *enabling* principle [5]. However the enabling principle by itself is not sufficient because many transformations *mutually* enable the others. Furthermore, different parts of a CDFG may resort to different orderings [22].

In Section 3.2, we presented a bottleneck analyzer that dynamically determines a bottleneck to optimize. The associated P-models are used to characterize transformations. As in Section 3.4, we characterized transformations for area reduction (FU's) into three categories. In each category, there typically exist only a few transformations that can directly address the designated goal. They are called *kernel* transformations. Those kernel transformations may not be 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. With this classification, each node in a T-graph is tagged as an either kernel or enabling transformation.

Since a kernel transformation may also enable other kernel ones, we can derive the primary (*partial*) ordering among kernel transformations. The primary ordering determines the order in which kernel transformations are applied. As to enabling transformations, they will be invoked as demanded by the kernel ones (demand-driven). With this approach, we can avoid the *infinite transformation loop*; for example, CSE vs. CSR. CSE is the kernel transformation for count reduction and CSR is an enabler for other kernel ones. CSE does *not* enable CSR in the strict sense, and CSR will be invoked only if it can enable other kernel ones to reduce the cost. Thus in our framework, CSR and CSE do not enable each other. A similar idea applies to the distributivity and reverse distributivity.

For (FU's) area optimization, we have derived the T-graphs for count reduction, operation conversion, and operation re-ordering, respectively. Due to the lack of space, the details of the T-graphs are not elaborated here.

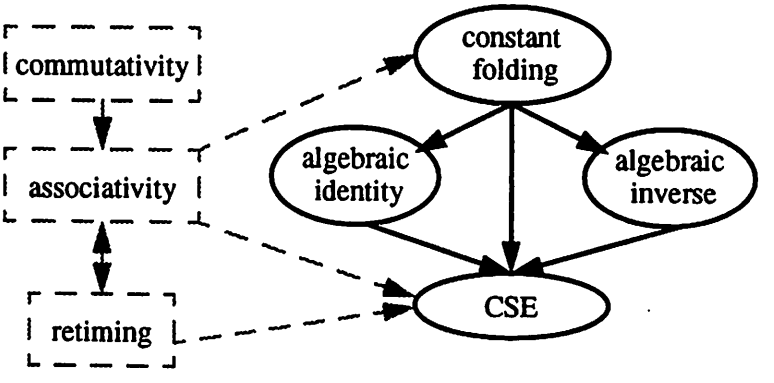

Figure 6: Partial T-graph for operation reduction

A partial T-graph for operation reduction is shown in Figure 6. Constant folding, algebraic identity and the inverse, CSE, and reverse distributivity are the kernel transformations. Commutativity, associativity, retiming, and CSR are only enabling transformations.

3.6 Transformation manager

Given the identified bottleneck and T-graphs, the transformation manager is to determine *which* transformations should be applied and *where* to apply them. The manager takes as input the CDFG, a set of T-graphs, and the bottleneck. According to the identified bottleneck, the corresponding T-graph is selected. In the case of area optimization, the selected T-graph may contain three sub-Tgraphs; one for count reduction, one for weighted-sum reduction, and one for utilization enhancement, in that order. The transformation manager will process the sub-Tgraphs one by one.

The transformation manager in our framework consists of three sub-modules: the algorithm partitioner, transformation analyzer, and transformation selector (Figure 7). These sub-modules will be briefly discussed in the following sections. The primary responsibility of the T-manager is to reduce the search space and thus improve the efficiency.

3.6.1 Algorithm partitioner

Based on a given bottleneck, the partitioner extracts the trouble spots of a given CDFG. Transformations will be applied only to these subgraphs. These transformations which are not applicable in these subgraphs are omitted because they cannot improve the bottleneck. This reduces the transformation space. For reducing the

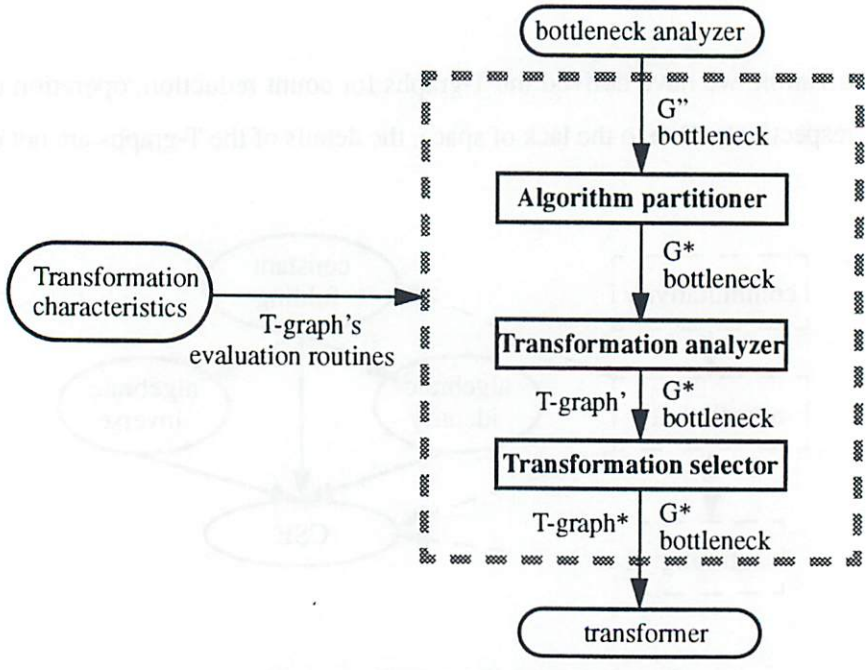

Figure 7: Structure of the transformation manager

count or the weighted sum of a certain resource, the subgraphs are composed of operations which impact the resource. Alternatively, if the goal is to improve the utilization of a certain resource, the operations contributing to the maximal height of the distribution graph are the candidates. Among these, the operations with small *slacks* are of special interest because they have higher contribution and their movement may produce larger gain. In this case, the subgraphs basically consist of the paths covering those candidates that have relatively small slacks.

3.6.2 Transformation analyzer

The transformation analyzer predicts the *potential improvements (benefits)* of transformations and their *possible side effects (costs)*. This will be used by the transformation selector to do the selection. The purpose is to avoid the redundant transformations and thus improve the efficiency. Since only kernel transformations directly affect the bottleneck, potential improvements from them are of a major concern. In our framework, each kernel transformation in a T-graph has a model (as an associated *evaluation routine*).

Rim [23] presented some models to predict performance of some loop transformations. We also have established models for algebraic transformations. For the *always-win* transformations, like constant folding or dead code elimination, the associated evaluation routines are trivial. We are currently working on models for other transformations. Due to the lack of space, we cannot enumerate the obtained models here. One example is constant multiplication expansion. It can be evaluated with linear multiplications and the associated coefficients.

For a kernel transformation, if the associated *evaluation routine* is not yet defined, this transformation will always be selected by the transformation selector. The application of such a transformation is controlled only by the transformer module. This may create redundancy and reduce the efficiency of the transformer module.

3.6.3 Transformation selector

Based on the predicted performance of the transformations, a 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 will be applied in a demand-driven fashion (to avoid the redundant enablers). There is no need to preselect them. This selection results in a refined T-graph. The heuristic currently used by the transformation selector in our framework is rather straightforward; a kernel transformation is selected if its potential improvement is greater than some (dynamic) threshold.

3.7 Transformer

After the transformation manager has done its job, the bottleneck together with the partitioned CDFG and the refined T-graph are passed to the transformer module to guide the transformation task. The transformer applies selected transformations onto the subgraphs. These transformations are applied in the order dictated by the T-

graph. For those "*mutually enabled*" kernel transformations, they will be applied together. The cost function is provided by the associated P-model.

The transformer in our framework exploits two classes of techniques. One is using a dedicated approach. Our framework gives a preferential treatment to such an approach, if one exists in the T-lib that matches the cost function and covers the transformations to be applied. The dedicated approaches have a higher priority due to their efficiency and prowess. On the other hand, if there exists no matched dedicated approach, generic local-move-based optimization techniques, e.g. simulated annealing, steepest descent method, etc., are then used.

A generic technique requires that each kernel transformation in a T-graph has an associated *identification routine* and *action routine. Identification routine* is used to set up the search space of local moves, and *action routine* is to execute the certain transformation on a selected move. To avoid the redundant enablers, enabling transformations are handled in a *demand-driven* mode. The idea of the *postponing principles* has been integrated into the *identification routines,* which relax the conditions under which a kernel transformation can be applied. Whenever an enabler is needed, the *action routine* will automatically invoke it for enabling.

3.8 Flow of the transformation-based optimization process

In summary, our framework takes as input a *CDFG* and user-defined design constraints. A set of structural-property-based P-models and a transformation library (*T-lib*) are predefined. The transformations in the T-lib are characterized into a set of T-graphs, each of which is associated with a certain P-model (or a sub-model).

The bottleneck analyzer takes the set of P-models to identify the prime bottleneck, which is then passed to the transformation manager. The manager selects the appropriate T-graph and locates the trouble spots in the CDFG. After that, it predicts the potentials of transformations and suggests an appropriate set to apply. The selected set of transformations is represented as a refined T-graph. Finally, the bottleneck, the partitioned CDFG, and the refined T-graph are passed to the transformer. The transformer module applies the selected transformations in an order dictated by the T-graph onto the partitioned subgraph. After execution, the transformed CDFG plus the actions taken for optimization (as the refined T-graph) are sent back to the bottleneck analyzer to evaluate the result. If nothing better can be achieved, the best solution is returned to the user. Otherwise, the bottleneck analyzer inspects the history of bottlenecks/improvements and picks up one CDFG for further optimization. This is often the most recent one unless it was rejected. If possible, some feedback is provided to the transformation manager to adjust its selections. Of course, a new bottleneck is identified for the next iteration.

4 Experimental Results

So far, we have formulated the transformation-based optimization process in a systematic fashion, and developed an integrated framework. In this section, we present preliminary results on several examples. The

assumed objective is to minimize design cost (area) of functional units without violating given throughput constraint.

4.1 Second-order Volterra filter

The first example is a 2nd order Volterra filter whose structure is shown in Figure 8(a). We assume that a fast and big multiplier is used. Both multiplications and additions are unit-cycle operations, but the cost (area) of a multiplier is 16 times larger than that of an adder. We also assume that all the constant coefficients are different.

The critical path of the original structure is 12 clock cycles. Given a sample period of 12 clock cycles, any solution needs at least 2 multipliers and 1 adders (absolute min-bound). Since the multipliers are expensive, the common approach is to trade big multipliers for cheaper adders/shifters by using constant multiplication expansion (CM). After the expansion, the critical path is longer than the sample period, and thus speed optimization techniques have to be invoked. Since a Volterra filter has variable multiplications, at least one multiplier is needed. In addition, extra adders and shifters will be needed due to the expansion. It is therefore by no means clear that this approach is a desirable one, especially since no other transformations have been considered yet. With our framework, we can systematically resolve this area optimization problem. Figure 9 shows the working flow of our framework. It illustrates one iteration of the optimization flow as described in Section 3.

With the P-models discussed in Section 3.1, the bottleneck analyzer predicts 5 multipliers and 1 adder may be required (the distribution graph for multiplications is shown in Figure 8d). Multipliers are thus identified as the bottleneck resource in area. According to the layered P-models, the optimization process iterates through 3 rounds. The first round is operation reduction. The only applicable kernel transformation, reverse distributivity, enabled by the associativity and the commutativity, successfully reduces the number of multiplications from 17 to 11 as shown in Figure 8(b). The absolute min-bound of multipliers is reduced from 2 to 1 (Figure 8e).

The next round is operation conversion, which trades multipliers for adders/shifters. Due to the existence of variable multiplications, at least one multiplier is needed. Because the current absolute min-bound of multipliers is already one, CM cannot reduce it any further. The potential improvement of CM predicted by transformation analyzer is *zero* (ignoring the extra overhead of the additional adders/shifters). Since no good transformation is available at this level, this round is skipped.

The third round is utilization improvement. The distribution graph predicts 2 multipliers (the height is 1.51) might be needed, but the absolute min-bound shows that only 1 multiplier is definitely required. Transformations of operation reordering to improve the resource utilization are considered. In this case, the algorithm partitioner has no effect because the whole structure should be considered. Applicable kernel transformations are retiming, associativity, and distributivity. Among them, distributivity is undesirable because it may increase the absolute min-bound from 1 to 2. Also, the potential improvement of associativity is very small due to the

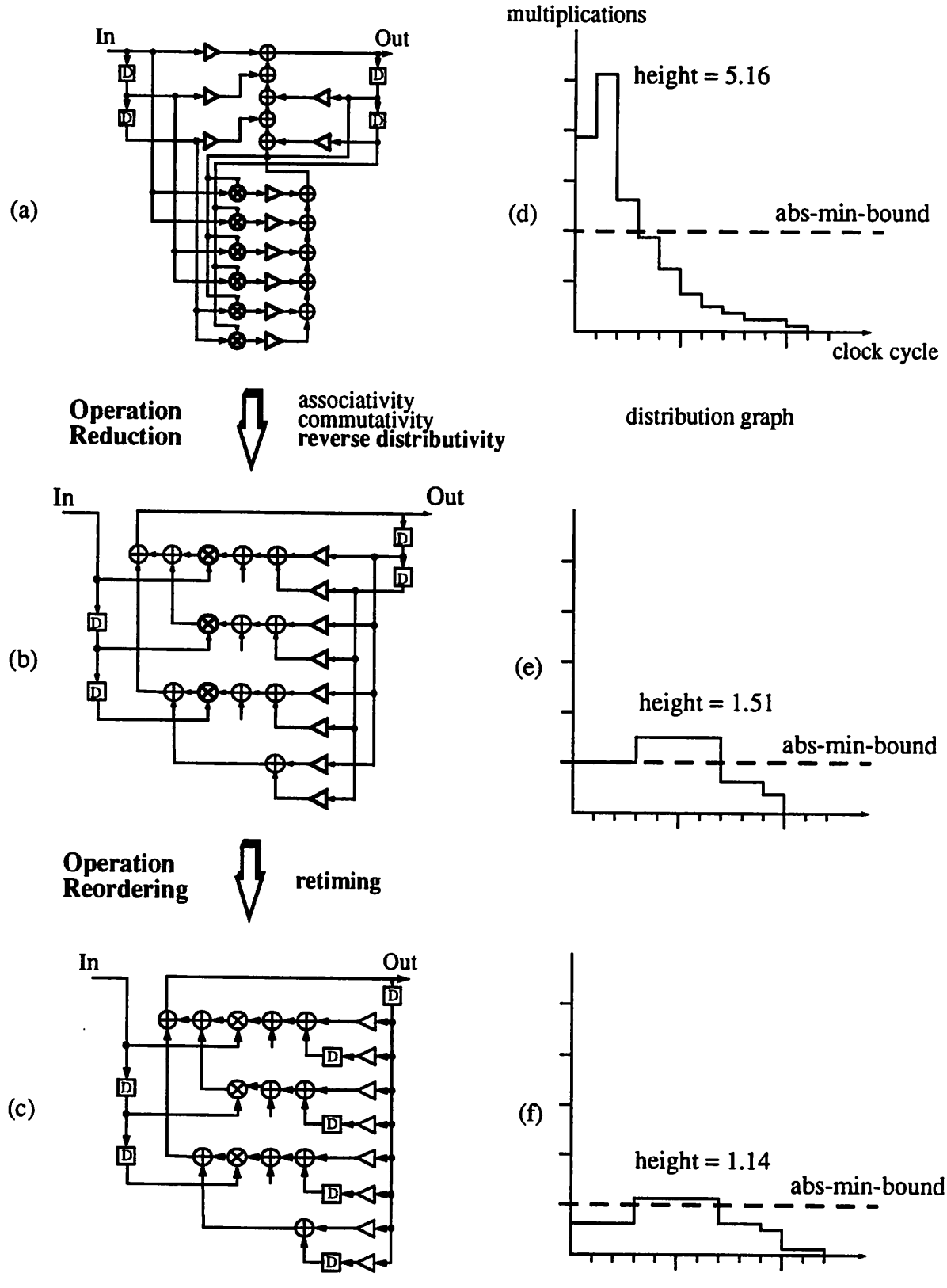

Figure 8: Structures of a 2nd order Volterra filter with the distribution graphs

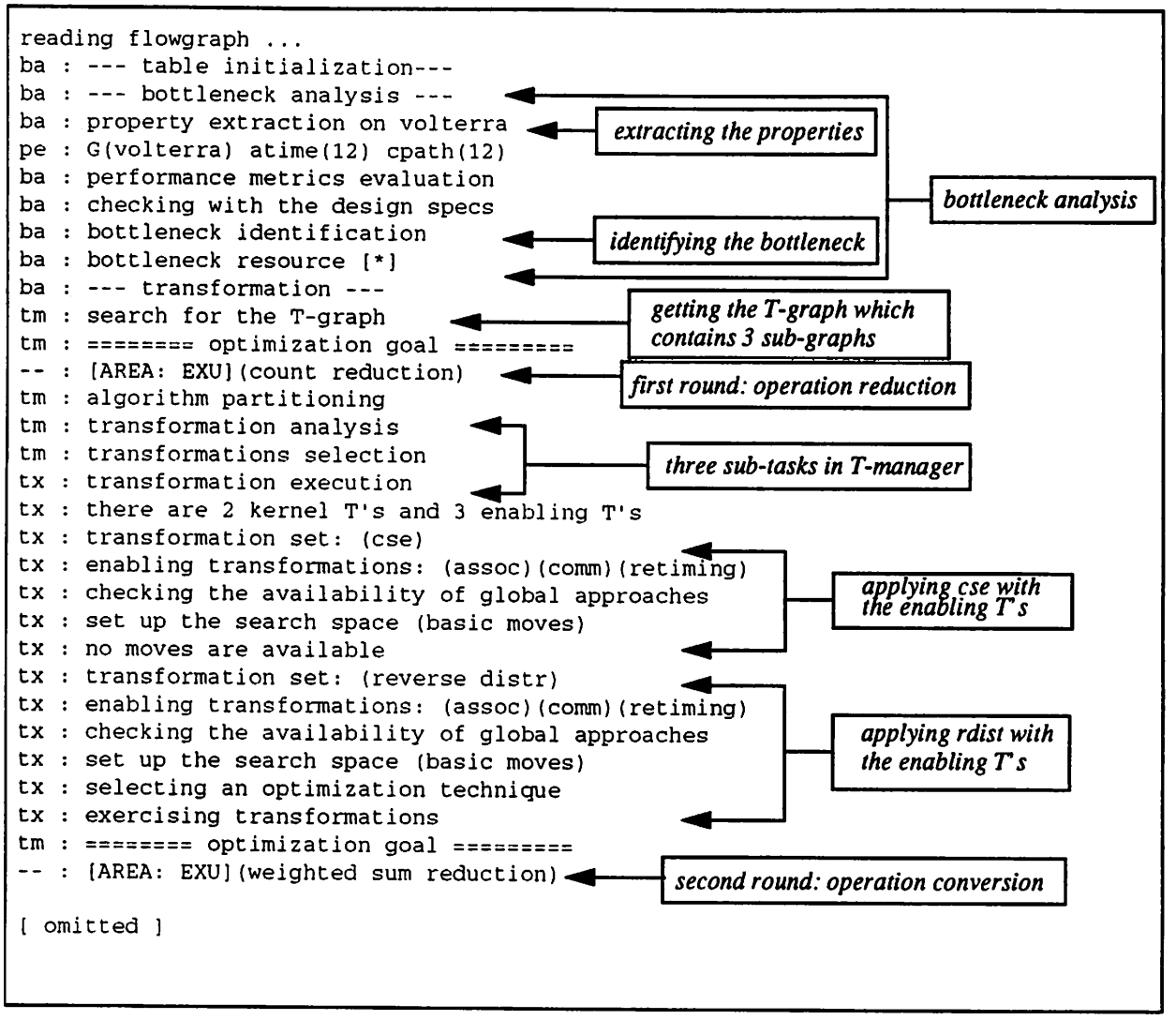

Figure 9: Working flow of the TAO framework

lack of a long multiplication (addition) chain. The transformation manager suggests that only retiming has good potential. The transformer follows and successfully reduces the predicted number of multipliers from 2 to 1 (the height is 1.14 as shown in Figure 8f) with retiming. The final structure shown in Figure 8(c) only needs 1 multiplier and 1 adder.

This optimization process saves 1 multiplier without adding extra units, which is about 50% reduction in the datapath. This result is superior to any of the existing approaches. Speed optimization techniques are not needed either. Throughout this process, only capable and applicable transformations are considered at each stage and only the appropriate transformations are applied. This methodology systematically determines what to optimize (with the *layered P-models*) and which transformations to use (based on the *characteristics of transformations*), and quickly reaches a better solution. One thing worth to notice is that CM is not needed in this case at all, although constant multiplications still exist in the final structure. This example also shows the effectiveness of the transformation ordering. If we apply CM first, we are hardly able reach the best solution.

Applying the transformations in an appropriate order not only can reach a good solution faster, but also may achieve a better one.

4.2 5th order elliptic wave filter

The second example is a 5th order elliptic wave filter. This example has been used as a benchmark for high level synthesis [34]. The general assumption is that a multiplication takes 2 clock cycles and an addition takes 1 clock cycle. Given the sample period of 17 clock cycles, the optimal solution needs 2 multipliers and 3 adders. The existing best solution by using transformations is 2 multipliers and 2 adders [36]. To compare the result with other approaches, we intentionally disable the constant multiplication expansion by assuming that a multiplier is only 2 times larger than an adder. (Note that different results might be obtained with different sample period and resource costs.)

The flow of the optimization process is very similar to the previous example. With the associativity and retiming (enabled by CSR), the predicted multipliers are reduced from 4 to 2. In the final design, 1 multiplier and 2 adders are needed, which is about 43% reduction in the area of datapath.

4.3 Tree search vector quantizer (distance computation)

The third example is TSVQ. Figure 10 repeats a manual transformation process as described in [35]. The objective is to reduce the number of accesses to multipliers and memory units in order to lower the power consumptions.

$$
\begin{aligned}
MSE_{ab} &= \sum_{i=0}^{15} (A_i - X_i)^2 - \sum_{i=0}^{15} (B_i - X_i)^2 \\
&= \sum_{i=0}^{15} \left[A_i^2 - 2X_iA_i + X_i^2 - \left(B_i^2 - 2X_iB_i + X_i^2 \right) \right] && \text{expansion} \\
&= \left(\sum_{i=0}^{15} \left(A_i^2 - B_i^2 \right) + \sum_{i=0}^{15} 2X_i(B_i - A_i) \right) && \text{regrouping}
\end{aligned}
$$

Figure 10: Transformations (TSVQ-1)

This calculation is to get the difference between the distances of the input vector X from two other constant vectors A and B. In the optimization process, loop fusion is first applied to combine the two summations. It is then followed by the expansion of the quadratics. The last step is to simplify and regroup the computations. The number of multiplications is reduced from 32 to 16. However, this gain is not clear until the final expression is obtain. It is nearly impossible to predict the improvement during the process. Also, regrouping after the expansion is often not an easy task to automate.

Figure 11 shows a similar result obtained from our framework. A new transformation is introduced here: $(x^2-y^2) = (x+y)(x-y)$, denoted by T_1. This transformation trades one multiplication for one addition (*operation conversion*). The bottleneck analyzer first identifies the number of accesses to multipliers is the dominant factor (we assume the access to multipliers is more expensive). The transformation manager suggests T_1 to apply. Following this, the transformer attempts to execute this decision. However, it is not applicable because of the summations (loops). The enable transformation, loop fusion, is invoked for enabling. (In our system, a loop is represented as a hierarchy node in the CDFG. A legal loop fusion can be recognized by matching the loop variables of two loops and checking the array dependencies. A kernel transformation which can be enabled by loop fusion can be easily recognized by ignoring the loop boundaries and just checking the loop bodies of two loops.)

After the conversion, the number of multiplication is reduced to 16. If the number of access to multipliers is the only concern, the optimal solution is reached and the process is done. Otherwise, second iteration is invoked to reduce the access to adders (or memory units). Because (A_i+B_i), (A_i-B_i), $(A_i+B_i)*(A_i-B_i)$, and $2(A_i-B_i)$ can be precomputed, the constant folding enabled by distributivity is applied. Finally, a similar result as obtained in [35] is reached. During the whole process, the gain at each step can be predicted because only capable transformations are considered and enabling transformations are applied only if they are demanded by the kernel ones.

$$MSE_{ab} = \sum_{i=0}^{15} (A_i - X_i)^2 - \sum_{i=0}^{15} (B_i - X_i)^2$$

loop fusion(enabling)

$$= \sum_{i=0}^{15} [\,(A_i - X_i)^2 - (B_i - X_i)^2]$$

conversion

$$= \sum_{i=0}^{15} [\,(A_i - X_i) + (B_i - X_i)\,] \cdot (A_i - B_i)$$

constant folding
enabled by distributivity

$$= \sum_{i=0}^{15} \left[(A_i + B_i) \cdot (A_i - B_i) - 2X_i (A_i - B_i) \right]$$

constant folding (loop)

$$= \left(\sum_{i=0}^{15} (A_i + B_i) \cdot (A_i - B_i) - \sum_{i=0}^{15} 2X_i (A_i - B_i) \right)$$

Figure 11: Transformations (TSVQ-2)

4.4 Summary

In summary, we have systematically gone through a few simple examples and demonstrated some promising results. In addition to these examples, we also have done some other experiments on a few digital filters. The optimization process on these simple examples took from negligible up to 30 seconds on a SUN SPARC 2. The actual run-time complexity depends on the number of applicable transformations and the number of iterations. Usually a good solution can be obtained within 1 second. We are currently working on more experiments on larger examples.

At last, we conclude this section with an observation. It is clear that all the optimization processes on the above examples can be translated into simple but dedicated global approaches. This is actually the beauty of our framework. Since all these shown examples require different set of transformations with different orderings, it is hardly possible to develop dedicated approaches by enumerating all the possibilities. With our framework, however, an appropriate transformation set can be easily detected and applied in a generic fashion.

5 Future work

We have developed a transformation framework for area optimization. Promising results have been obtained. To make this framework more effective and efficient, more efforts need to be devoted to enhance the evaluation routines for the transformation analyzer. We also plan to incorporate additional transformations into the framework. Some experiments on larger examples will be done as well. In addition, our main attention so far has been focused on the optimization of datapath resources such as FU's, registers, and interconnect. Since many real-time DSP applications are memory-intensive, memory-related issues should also be considered. This extension will require the introduction of a larger set of loop transformations.

6 Conclusion

In this report, we have constructed the *TAO* system, an integrated framework for optimizing transformations. This framework, based on a generic methodology, can *systematically* choose appropriate transformations to apply at the right place and in the right order. We have proposed a layered prediction model for area, and classified transformations into three different categories (for FU's). Some techniques, like the enabling/postponing principles, are integrated to enhance the framework. The proposed framework is *extensible*, in that new cost functions and new transformations can be easily integrated, and supports *dynamic optimization flow.*

Since algorithm developers seldom take into account the implementation cost, and hardware designers rarely consider the merits of various algorithm instances, there exists a gap between the algorithms conceived by the software developer and those used by hardware designers. The *TAO* framework can bridge the gap by assisting hardware designers, under their specific design constraints, to exploit optimizing transformations to easily and quickly explore the algorithmic design space and attain better designs.

7 References

[1] R. Hartley, A. Casavant: "Tree-height minimization in pipelined architectures," *Proc. of ICCAD*, pp. 112-115, Nov. 1989.

[2] C.E. Leiserson, J.B. Saxe: "Retiming Synchronous Circuitry," *Algorithmica*, Vol. 6, pp. 5-35, 1991.

[3] K.K. Parhi, D.G. Messerschmitt: "Pipeline interleaving and parallelism in recursive digital filters - part I: pipelining using scattered look-ahead and decomposition," *IEEE Trans. on ASSP*, vol. 37, no. 7, pp. 1099-1117, July 1989.

[4] K.K. Parhi, D.G. Messerschmitt: "Static Rate-Optimal Scheduling of Iterative Data-Flow Programs via Optimum Unfolding," *IEEE Trans. on Computers*, vol. 40, no. 2, pp. 178-191, Feb 1991.

[5] M. Potkonjak, J. Rabaey: "Maximally fast and arbitrarily fast implementation of linear computations," *Proc. of ICCAD*, pp. 304-308, Nov. 1992

[6] M. Potkonjak, J. Rabaey: "Pipelining: Just another transformation," *Int'l Conf. on Application Specific Array Processors*," pp. 163-175, 1992.

[7] Z. Iqbal, M. Potkonjak, S. Dey, A. Parker, "Critical path minimization using retiming and algebraic speed-up," *Proc. of DAC*, pp. 573-577, 1993.

[8] S.-H. Huang, J.M. Rabaey, "Maximizing the Throughput of High Performance DSP Applications Using Behavioral Transformations," *Proc. of EDAC-ETC-EUROASIC 94*, pp. 25-30, March 1994.

[9] D.J. Kolson, A. Nicolau, N. Dutt, "Integrating program transformations in the memory-based synthesis of image and video algorithms," *Proc. of ICCAD*, pp. 27-34, Nov 1994.

[10] M. Potkonjak, J. Rabaey: "Optimizing the resource utilization using transformations," *Proc. of ICCAD*, Nov 1991.

[11] R. Karri, A. Orailoglu, "Transformation-based register optimization in high-level synthesis," *Conf. Record of Asilomar Conf. on Signals, Systems and Computers*, pp. 894-898, 1992.

[12] M. Janssen, F. Catthoor, H. D. Man, "A specification invariant technique for operation cost minimization in flow-graphs," *Int'l Sym. High Level Synthesis*, pp. 146-151, 1994.

[13] M. Sheliga, E. H.-M. Sha, "Global node reduction of linear systems using ratio analysis," *Int'l Sym. High Level Synthesis*, pp. 140-145, 1994.

[14] V. Chaiyakul, D.D. Gajski, L. Ramachandran, "High-level transformations for minimizing syntactic variables," *Proc. of DAC*, pp. 413-418, June 1994.

[15] W.-K. Cheng, Y,-L, Lin, "A transformation-based approach for storage optimization," *Proc. of DAC*, pp. 158-163, June 1995.

[16] M. Potkonjak, M.B. Srivastava, A. Chandrakasan, "Efficient substitution of multiple constant multiplications by shifts and additions using iterative pairwise matching," *Proc. of DAC*, pp. 189-194, 1994.

[17] A.P. Chandrakasan, M. Potkonjak, J.M. Rabaey, R.W. Broderson, "HYPER-LP: A System for Power Minimization Using Architectural Transformations," *Proc. of ICCAD*, pp. 300-303, Nov. 1992

[18] M.F.X.B. van Swaaij, F.H.M. Franssen, F.V.M. Catthoor, H.J. De Man, "Automating High Level Control Flow Transformations for DSP Memory Management," in *VLSI Signal Processing*, Vol. 5, edited by K. Yao *et al.*, pp. 397-406, IEEE Special Publications, 1992.

[19] M.E. Wolf, M.S. Lam: "A loop transformation theory and an algorithm to maximize parallelism," *IEEE Trans. on Parallel and Distributed Systems*, vol. 2, no. 4, pp. 452-471, Oct. 1991.

[20] V. Sarkar, R. Thekkath, "A general framework for iteration-reordering loop transformations," *Proc. of the ACM SIGPLAN '92 Conf. on Programming Language Design and Implementation*, pp.175-187, June 1992.

[21] W. Kelly, W. Pugh, "Finding legal reordering transformations using mappings," *Proc. of 7th Int'l Workshop on Languages and Compilers for Parallel Computing*, pp. 107-124, Aug 1994.

[22] D. Whitfield, M.L. Soffa, "Investigating properties of code transformations," *Proc. of Int'l Conference on Parallel Processing*, pp. II-156-160, Aug 1993.

[23] M. Rim, R. Jain, "Estimating performance characteristics of loop transformations," *Int'l Symp. on Circuits and Systems*, pp. 249-252, June 1994.

[24] J.M. Rabaey, L.M. Guerra, "Exploring the architecture and algorithmic space for signal processing applications,"

[25] P.G. Paulin, J.P. Knight, "Force-directed scheduling for the behavioral synthesis of ASIC," *IEEE Trans. on CAD*, vol. 8, no. 6, pp. 661-679, June 1989.

[26] L. Guerra, M. Potkonjak, J.M. Rabaey, "System-level design guidance using algorithm properties," *VLSI Signal Processing VII*, IEEE Press, NY, pp. 73-82, 1994.

[27] R. Jain, A.C. Parker, N. Park, "Predicting system-level area and delay for pipelined and nonpipelined designs," IEEE Transactions on computer-aided design, pp. 955-965, vol. 11, no. 8, Aug 1992.

[28] A. Sharma, R. Jain, "Estimating architectural resources and performance for high-level synthesis applications," *Proc. of DAC*, pp. 355-360, June 1993.

[29] S. Chaudhuri, R.A. Walker, "Computing lower bounds on functional units before scheduling," *Proc. Int'l Symp. on High-Level Synthesis*, pp. 36-41, May 1994.

[30] S.Y.Ohm, F.J.Kurdahi, Nikil Dutt, "Comprehensive lower bound estimation from behavioral descriptions," *Proc. of ICCAD*, pp. 182-187, Nov 1994.

[31] D.F. Bacon, S.L. Graham, O.J. Sharp, "Compiler transformations for high-performance computing," *ACM Computing Survey*, pp. 345-420, Dec 1994.

[32] D. Whitfield, M.L. Soffa, "An approach to ordering optimizing transformations," *2nd ACM Symposium on Principles and Practice of Parallel Programming*, pp. 137-147, March 1990.

[33] D. Whitfield, M.L. Soffa, "Automatic generation of global optimizers," *Proc. of the ACM SIGPLAN '91 Conf. on Programming Language Design and Implementation*, pp. 120-129, June 26-28, 1991.

[34] K.S. Huang *et al.*, "Workshop on high-level synthesis," Orcas Island, WA, Jan 1988.

[35] D.B. Lidsky, J.M. Rabaey, "Low power design of memory intensive functions case study: vector quantization," *VLSI Signal Processing VII*, IEEE Press, NY, pp. 378-387, 1994.

[36] M. Potkonjak, "Algorithms for high level synthesis: resource utilization based approach," *Ph.D. dissertation*. UCB/ERL M92/10, 1992.