Effects of Mixing Design Styles On The Synthesis of RTL Components

James R. Kipps and Daniel D. Gajski

Technical Report 91-42

Information and Computer Science
University of California, Irvine
Irvine, CA 92717

Abstract

By mixing design styles during synthesis of RTL components such as adders, multipliers, and ALUs, it is possible to generate a range of designs from small to fast, where intermediate designs make favorable and possibly desirable tradeoffs between area and delay. Although module generators can be written to reflect design styles that reduce either area or delay, the current approach to generator execution does not examine the effects of mixing different design styles. We have developed an approach to RTL component synthesis that searches the space of design alternatives, and we have implemented this approach with the DTAS Design Language. The significance of our approach is that it allows DTAS to generate designs use a combination of design styles and to compare the effects of mixing styles. In this paper, we outline the operation of DTAS and describe how DTAS expands and constrains the design space. We present results from applying DTAS to large RTL components using an MCNC benchmark library. We also present results of integrating DTAS with the MISII logic optimizer.

Effects of Mixing Design Styles in The Synthesis of RTL Components

Abstract

By mixing design styles during synthesis of RTL components such as adders, multipliers, and ALUs, it is possible to generate a range of designs from small to fast, where intermediate designs make favorable and possibly desirable tradeoffs between area and delay. Although module generators can be written to reflect design styles that reduce either area or delay, the current approach to generator execution does not examine the effects of mixing different design styles. We have developed an approach to RTL component synthesis that searches the space of design alternatives, and we have implemented this approach in a system called DTAS. The significance of our approach is that it allows DTAS to generate designs use a combination of design styles and to compare the effects of mixing styles. In this paper, we outline the operation of DTAS and describe how DTAS expands and constrains the design space. We present results from applying DTAS to large RTL components using an MCNC benchmark cell library. We also present results of integrating DTAS with the MISII logic optimization system.

1 Introduction

The design style used in mapping register-transfer level (RTL) components, such as adders, multipliers, and ALUs, into logic or layout impacts the quality of the resulting design with regard to area or delay. For example, a ripple-carry style generates a small but slow adder, while a carry look-ahead style generates a fast but large adder, and a matrix style generates a small but slow multiplier, while a tree style generates a fast but large multiplier. Design styles can be mixed to generate designs that tradeoff area and delay. A 16-bit adder designed by rippling four 4-bit carry look-ahead adders will be intermediate in area and delay between a full ripple-carry adder and a full carry look-

ahead adder. Whether the tradeoffs are actually desirable is difficult to determine in the absence of technology mapping.

RTL components are often synthesized using module generators, where a module generator is treated as procedure that generates logic or layout for a parameterized component specification (1; 2; 3). Module generators can be written to reflect alternative design styles. Such generators will include parameters with values such as **smallest** or **fastest** to select between competing styles. The problem with this approach is that it only finds a single point in the design space of an RTL component. There may exists alternative designs that make favorable and even desirable tradeoffs between area and delay. Without searching the design space, the "single design" approach to module generation will fail to find these alternatives.

We have developed a search-based approach to RTL component synthesis that explores the effects of mixing alternative design styles. In this approach, input component specifications are mapped into a given cell library by functional decomposition, where design styles are represented as decomposition methods. When multiple decomposition methods are applicable, each is tried. Search is controlled by constraining the size of the design space with performance filters and other means. We have implemented this approach in a system/language called DTAS.

Our objectives in developing DTAS are two-fold. One objective is to achieve a high-level of design quality for large RTL data path components, such as 64-bit ALUs. We are attempting to meet this objective in two ways. First, when possible, components are mapped into complex RTL library cells, i.e., cells that can provide highly optimized layout in comparison to functionally equivalent configurations of Boolean cells (4). Second, alternative design styles are explored dynamically in order to find designs that best meet

performance constraints. The other objective is to maintain design integrity against technology changes. We are attempting to meet this objective by automating the process of generating library-specific design methods (5).

In this paper, we focus on the use of alternative design styles and the effects mixing designs has on design quality. We present results obtained by applying DTAS to a 16-bit adder, 24-bit multiplier, and 64-bit, 16-function ALU. Designs are mapped into cells from an MCNC benchmark library. We also present a comparison of performance gains achieved by passing DTAS's designs through the MISII logic optimization system (6). These results indicate that by dynamically examining the effects of mixing design styles it is possible to find designs that make very favorable tradeoffs between area and delay. These results complement traditional approaches to logic synthesis. By further application of logic optimization techniques, such as those found in MISII, it is possible to proportionally refine the performance characteristics of DTAS-generated designs.

2 DTAS: System Overview

DTAS is a functional synthesis system for RTL data path components. This includes *combinational components*, such as decoders, multiplexers, parity checkers, and function generators, *arithmetic components*, such as adders, comparators, multipliers, and ALUs, and *sequential components*, such as shift registers and counters. Designs are hierarchically decomposed into netlists of cells from a given ASIC vendor's library. These can be simple Boolean cells or complex functional cells, from multiplexers, adders, and comparators up to n-bit ALUs, multipliers, and counters. DTAS compares alternative design styles to find candidate designs that best fit performance constraints. Designs can be output in structural VHDL and input to logic synthesis and layout tools.

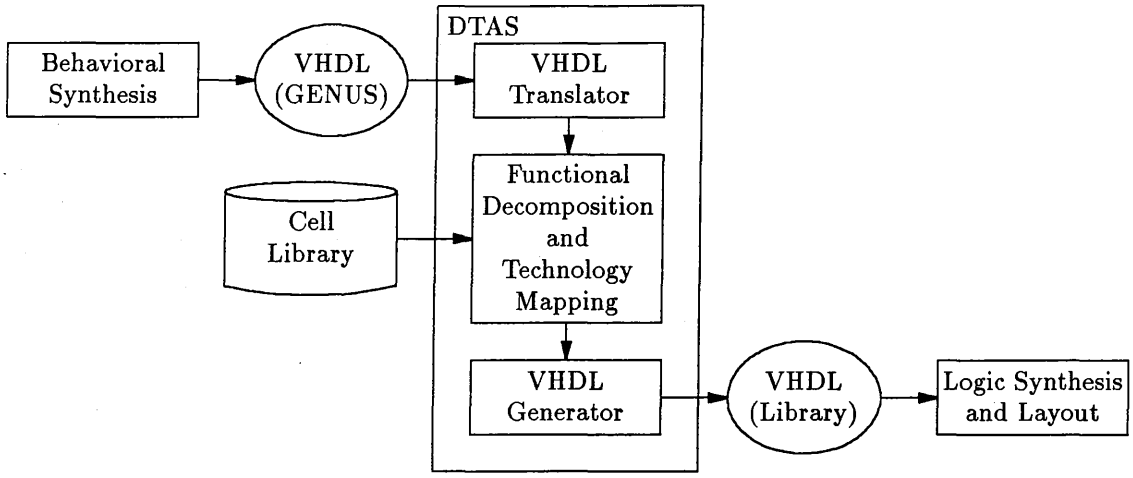

Figure 1: Top-level structure of DTAS

The top-level structure of DTAS is outlined in Fig. 1. The input to DTAS is a technology-independent netlist of RTL components, described using the GENUS component library (7). This netlist is passed through a phase of functional decomposition and technology mapping. The result is a set of hierarchical, library-specific netlists that represent alternative implementations of the components in the input netlist. Each output netlist traces the top-down design of the input netlist into subcomponents. Leaves implement the design with cells drawn from the given library. Netlists vary by the design styles and library cells used in their construction.

DTAS is implemented as a rule-based constructive language, the architecture of which is shown in Fig. 2. This includes a parser for reading and loading rules from text files and an interpreter for selecting and firing loaded rules. Each rule describes a decomposition method for a parameterized component specification. Decomposition can be described with a combination of Boolean description and connected subcomponents. While the DTAS Design Language is too complex to describe here, Fig. 3 presents three sample

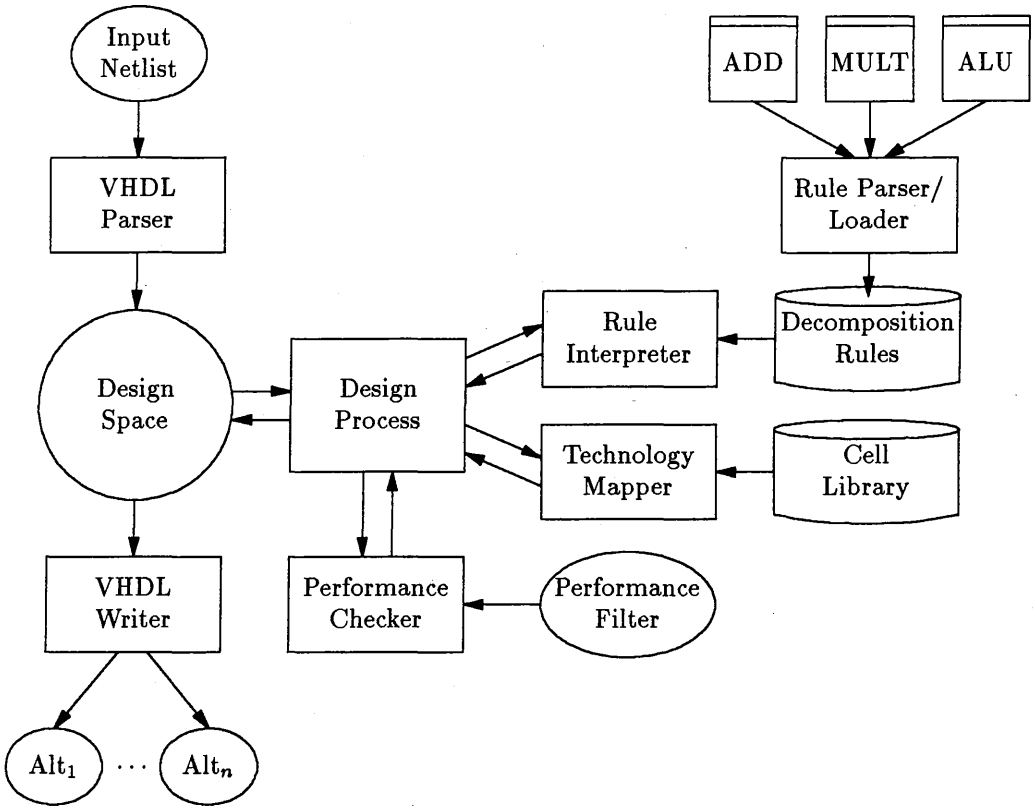

Figure 2: System architecture of DTAS design language

rules for decomposing adders (ADD). The rule in Fig. 3 (*a*) decomposes a 1-bit adder into Boolean logic; square brackets surrounding ports denote optionality. The other two rules depict alternative design styles for an n-bit adder. The rule in Fig. 3 (*b*) decomposes an n-bit adder into a netlist of n 1-bit adders connected with ripple carry. The rule in Fig. 3 (*c*) decomposes an n-bit adder, for $n \leq 4$, into n 1-bit adders, with carry propagate (P) and generate (G) outputs, and a carry look-ahead generator (CLA). There are additional rules, not shown here, for decomposing CLAs and for decomposing carry look-ahead adders for $n > 4$.

```
ADD(X Y [CI])
   (S [CO] [P G])
  ->
     let CI := 0 (default)
         P  := X(+)Y
         G  := X*Y
         S  := P(+)CI
         CO := CI*P + G
```

(*a*) 1-bit adder w/optional carry

```
ADD(X^?n Y^?n [CI])
   (S^?n [CO])
  :style RIPPLE
  :levels 0
  where ?n > 1
  ->
     for ?i from 1 to ?n
       ADD(X[?i] Y[?i] C.?i)
          (S[?i] C.?i+1)
     let C.1 := CI
         C.?n+1 := CO
```

(*b*) Ripple carry adder

```
ADD(X^?n Y^?n [CI])
   (S^?n [CO] [P G])
  :style CLA
  :levels 1
  where ?n > 1 & ?n < 4
  ->
     for ?i from 1 to ?n
       ADD(X[?i] Y[?i] C.?i)
          (S[?i] <> P.?i G.?i)
     CLA(P.1..?n G.1..?n C.1)
        (C.2..?n CO P G)
     let C.1 := CI
```

(*c*) Carry look-ahead adder

Figure 3: Sample DTAS decomposition rules

Internally, the input netlist is represented as a connection of *modules*, where a module maps the functional specification of a component to a particular implementation of the specification. There is a one-to-many mapping between component specifications and component implementations; a module is viewed as an instance of a single implementation. At the beginning of the design process, the modules of the input netlist are said to be *uninstantiated*; i.e., they are specified but not link to an implementation. At the end, the input netlist may have been duplicated several times. In each copy, the modules of the netlist will be linked to a different set of implementations, giving a set of functionally-equivalent designs with varying performance characteristics.

The input netlist is used to initialize the design space. The goal of the design process is to fully instantiate the design space from this seed. The design process uses the rule

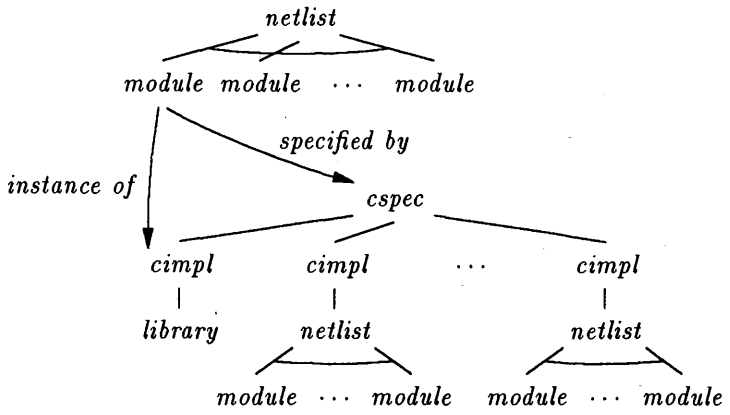

Figure 4: Design space representation

interpreter and the technology mapper to construct alternative implementations of each component specification of the input netlist. The design process then compares various combination of the implementations and instantiates those that satisfy a user-defined performance filter on area and delay. Once the design process has fully instantiated the design space, DTAS provides tools for examining and testing designs and for outputting the design in structural VHDL.

3 Expanding The Design Space

The design space is represented as an acyclic graph rooted at the input netlist of uninstantiated modules. As shown in Fig. 4, nodes of the graph alternate between netlists, modules, component specifications (*cspec*), and alternative component implementations (*cimpl*). Each component implementation corresponds to either a library cell or a netlist of modules. Each netlist represents one level of component decomposition into connected subcomponents. Each module represents an instance of a single implementation of a specified component. Component implementations corresponding to library cells are said to

```
ADD(X^4 Y^4 CI)
   (S^4 CO)
  :data X[] Y[] S[]
  :carry CI CO
  :style CLA
  :levels 1
```

Figure 5: Sample component specification

be *grounded*; grounded components constitute the leaves of the graph. Branches that do not eventually lead to grounded components are pruned from the graph. The design space is expanded by firing decomposition rules and grounded by matching component specifications to the specification of library cells.

The design process begins with the uninstantiated modules of the input netlist and recursively expands the design space for each distinct component specification. An example component specification for a 4-bit carry look-ahead adder is shown in Fig. 5. It identifies the function type of the component (ADD), as well as its input ports (A B CI) and output ports (S CO); it specifies the width and type of each port; and it associates attribute/value pairs with the component, such as STYLE/CLA and LEVELS/1. The ports, their type and width, and the attribute/value pairs can all be treated as parameters of a specification.

To expand the design space for a component specification, the design process first sends the specification to the technology mapper. The technology mapper matches the specification agains the component specifications describing each library cell. For every successful match, the technology mapper returns a component implementation mapping the component specification to the matched library cell.

Next, the design process sends the specification to the rule interpreter, regardless of whether the technology mapper found a match. The rule interpreter matches the

component specification against the head of each decomposition rule, where the head of a rule can be viewed as a parameterized component specification template. For each successful match, the matching rule is *fired.* Firing a decomposition rule generates a netlist of uninstantiated modules that implement the specified component to one level of decomposition. The design process is then applied recursively to this new netlist. The result is a set of fully grounded alternative implementations of the netlist. The rule interpreter returns a component implementation for each grounded netlist generated in this manner.

When the design process finally has all the alternative implementations for each of the distinct component specifications in the netlist on which it is working, it compares combinations of the implementations and constructs a netlist of each combination that satisfies the given performance filter. These netlists are added to the design space. (Performance filters are discussed further in Sec. 4.)

A sample design space for a 4-bit adder is shown in Fig. 6; assume that the cell library includes a variety of Boolean gates and a 1-bit full adder (FA1). The decomposition rule shown in Fig. 7 matches the component specification at the root of the design space. This rule generates a set of netlists that decompose the adder into a ripple-carry adder with varying levels of carry look-ahead, assuming a 4-bit CLA. The design rule seen earlier in Fig. 3 (*b*) is applicable to a 4-bit adder with zero levels of carry look-ahead and generates a netlist of four 1-bit adders. The specification for the 1-bit adders matches the specification of the FA1 library cell, so the design process can ground one implementation of the netlist. The specification also matches the design rule seen earlier in Fig. 3 (*a*), which decomposes the 1-bit adders into Boolean gates, which can also be grounded. A rule not shown here decomposes the RIPPLE-style adder with one level of look-ahead

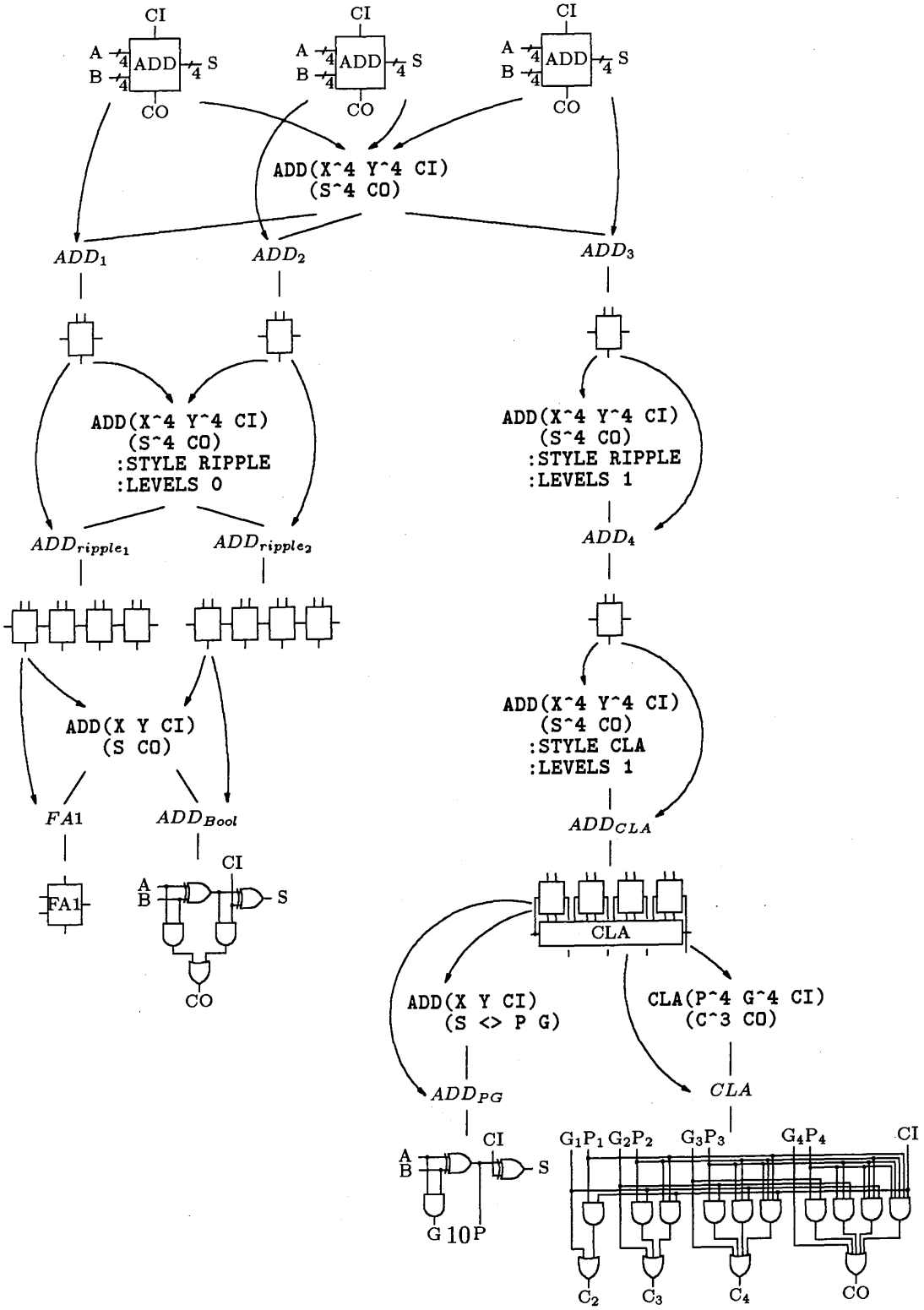

Figure 6: Design space for 4-bit adder

```
ADD(X^?n Y^?n [CI])
   (S^?n [CO])
  where ?n > 1
  varying ?l from 0 to ceiling(log(?n,4))
  ->
     ADD(X^?n Y^?n CI)
        (S^?n CO)
       :style RIPPLE
       :levels ?l
```

Figure 7: Decomposition rule for n-bit adder

into a CLA-style adder, and the rule seen in Fig. 3 (c) further decomposes the adder into four 1-bit adders with carry propagate (P) and generate (G) outputs and a 4-bit CLA. The rule from Fig. 3 (a) is also applicable to the 1-bit adders, decomposing them into Boolean gates; another rule not shown here decomposes the CLA into Boolean gates. The final result is that the input netlist for the 4-bit adder has three alternative implementations: ADD_1, ADD_2, and ADD_3.

4 Controlling Search

If fully expanded, the size of the design space for a given netlist is bounded by the product of the number of alternative implementations for each module in the netlist. Even for small components, such as a 16-bit adder, there can be several hundred thousand alternative designs, only a small percentage of which are of interest. For DTAS to become a viable synthesis tool, some form of search control is necessary. We use two fundamental controlling principles to limit the manner in which DTAS expands the design space.

The first principle is to ignore netlist implementations containing two or more identically specified modules that are *not* instances of the same component implementation. For instance, DTAS maps the 4-bit ripple-carry adder seen earlier in Fig. 6 into two implementations, one in which all four 1-bit adders are mapped into the FA1 library cell,

and the other in which all four 1-bit adders are mapped into Boolean gates. If our first principle were not used, the design space would include an additional 14 ripple-carry implementations that mix the 1-bit library adders with the 1-bit Boolean adders. This principle reduces the size of the design space for a given netlist to be the product of number of alternative implementations for each distinct component specification among the netlist's modules. This effectively reduces the design space for a 16-bit adder to the tens of thousands.

The second principle is to apply a *performance filter* during netlist construction. Construction is the bottom-up process in which the design process combines alternative implementations of the netlist's modules in order to generate alternative implementations of the netlist. Before returning this set of netlists, the design process passes it to a performance checker. The performance checker calculates the area and delay of each implementation, orders the implementations by area, from smallest to largest, and applies a user-defined performance filter to the ordered list. The performance filter is expected to delete undesirable implementations and return a new list that is a subset of the original. For instance, if delay is critical, the filter can discard all but some percentage of the fastest implementations; if area is critical, it can discard all but the smallest. The list returned by the performance filter is returned by the performance checker and added to the design space by the design process.

Because we are interested in examining the effects of mixing design styles on area and delay, our preferred filter discards implementations that do not make "favorable" tradeoffs between area and delay. We call this the *baseline range filter* and define "favorable" in the following manner. Plotting the smallest implementation and the fastest implementation within a two-dimensional graph of delay versus area, we imagine a line

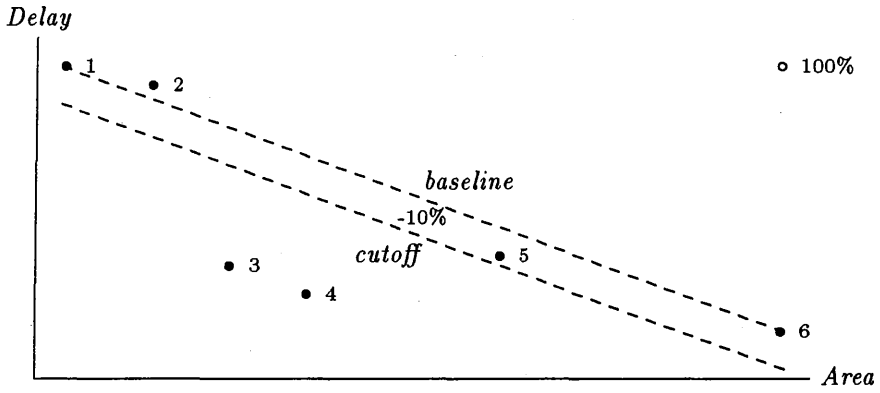

Figure 8: Example application of the baseline range filter

between these two points and call this the *baseline*. Any of the remaining implementations that falls between the origin of the graph and a parallel line, called the *cutoff*, some percentage distance from the baseline are considered to make a favorable tradeoff between area and delay; all other implementations are discarded.

To illustrate, consider the graph seen in Fig. 8, which plots six alternative designs by area and delay. Each alternative appears as a numbered dot. The *baseline* is shown as a dashed line connecting Design 1, which has the least area, to Design 6, which has the least delay. The *cutoff* is a distance of negative ten percent from the baseline, where a line at a distance of 100 percent from the baseline would intersect a point at the delay of the smallest design and the area of the fastest design. Given this cutoff, Designs 2 and 5 would be discarded by the perforance filter. The percentage distance of the cutoff from the baseline is a parameter of the baseline range filter and can be adjusted by the designer.

Using a performance filter can significantly constrains the size of the design space. For instance, even when the baseline range filter is applied only to Boolean gates, it still

reduces the design space for a 16-bit adder to less than a dozen alternative implementations. Examples of the baseline range filter will appear again in the next section, where we present our results.

5 Performance Results

To determine the effects of mixing design styles, we have applied DTAS to the task of designing three RTL components of increasing size and complexity: a 16-bit full adder (ADD16), a 24-by-24-to-48 bit multiplier (MULT24), and a 64-bit, 16-function arithmetic logic unit (ALU64/16). Generated designs were validated by simulation on randomly-selected inputs. We have also linked DTAS to the MISII logic optimization system (6) and report results of its applications to the ALU64/16 example. The results reported here are independent of mapping to MSI- and LSI-level RTL cells. We will present the results of using DTAS to map designs into a library of RTL in a future paper.

The cell library used in these experiments is the MCNC benchmark cell library "lib2.mis2lib" (8). This library contains a variety of one- and two-level Boolean gates. The performance filter used is the baseline range filter described in the last section. Area is computed as the sum of the area of the cells used in the design and is measured in 10^3 microns. Delay is computed using the intrinsic-plus-fanout delay model given in the library and is measured in nanoseconds. In our experiments, we vary the percentage distance of the cutoff to the baseline. All experiments were run using Common Lisp on a Sun-3/110 workstation. We also give the elapsed wall-clock time required to generate the entire set of designs for each example; this time is heavily dominated by the delay computations of the performance checker.

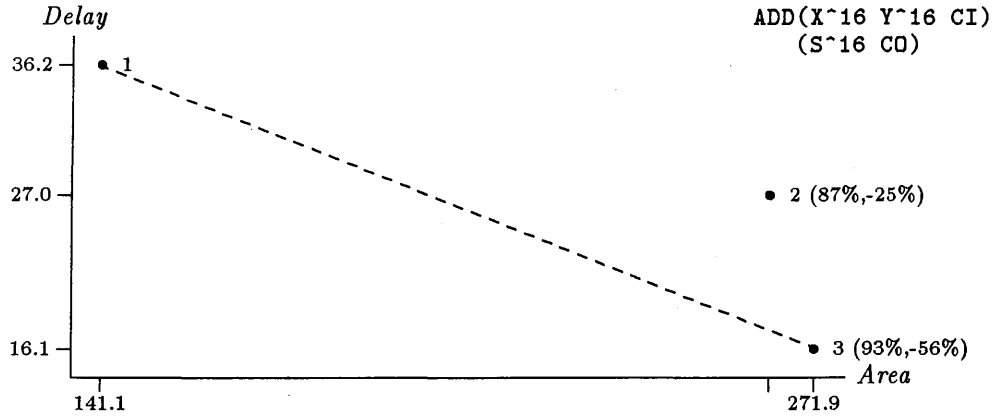

Figure 9: Design space for ADD16

The graph seen in Fig. 9 plots the results of applying DTAS to the component specification of ADD16, which appears in the upper left-hand corner. DTAS generated three alternative designs. Design 1 is the smallest, implementing ADD16 with full ripple carry. Design 3 is the fastest, implementing ADD16 with two levels of carry look-ahead. Design 2 implements ADD16 by rippling four carry look-ahead 4-bit adders. The dashed line between Design 1 and Design 3 represents the baseline. For this example, the performance filter was set to accepts implementations that are within 100 percent of the baseline, which is a very liberal interpretation of what constitutes a favorable tradeoff. In this case, Design 2 is not a necessarily desirable alternative. The wall-clock time required to generate this set of designs was 35 seconds.

The graph seen in Fig. 10 plots the results of applying DTAS to the component specification of MULT24. DTAS has rules that encode two multiplier design styles: matrix and tree. These rules encoding the tree style decomposes an n-bit multiplier into four smaller multipliers, a series of carry-save adders, and an n-bit adder. The partial-product multipliers can be decomposed further using either a tree or matrix style; the adder can be decomposed using any mix of adder design styles.

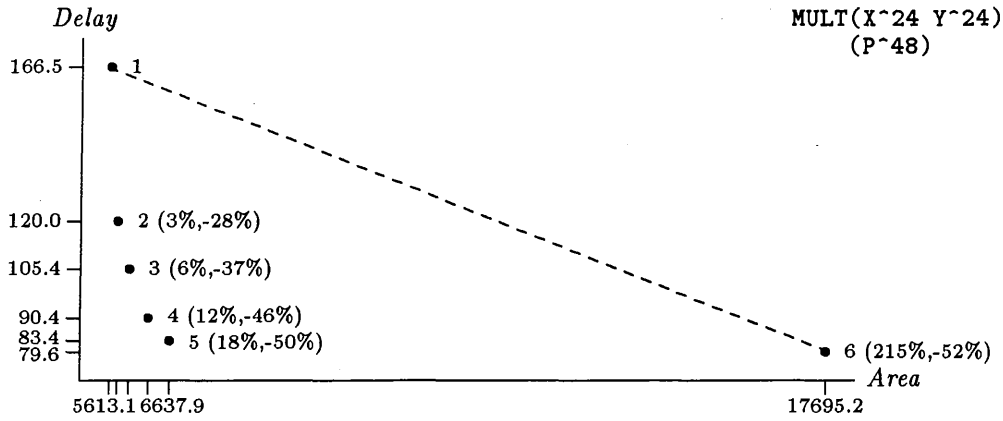

Figure 10: Design space for MULT24

Because of the range of possible designs, it was necessary to set the percentage cutoff to -10 percent of the baseline for this experiment. Nonetheless, DTAS still generated six alternative designs. At one end of the baseline, Design 1 implements MULT24 as a full matrix multiplier; at the other end, Design 6 implements MULT24 as a full tree multiplier with full carry look-ahead adders at each level of the tree. Design 6 reduces delay by 51 percent over Design 1, but only by increasing the area by 215 percent. In this case, the intermediate designs appear quite desirable. For a cost of 3 percent more area over Design 1, Design 2 reduces delay by 28 percent; for a cost of 2 percent in delay over Design 6, Design 5 reduces area by 73 percent. Designs 3 and 4 make similar tradeoffs. Design 2 implements MULT24 as a tree of matrix multipliers with a ripple carry adder; Design 3 as tree of matrix multipliers with a full carry look-ahead adder; Design 4 as tree of tree of matrix multipliers with ripple carry adders; and Design 5 as a tree of tree of matrix multipliers with full carry look-ahead adders. The wall-clock time required by DTAS was 5493 seconds.

The graph seen in Fig. 11 plots the results of applying DTAS to ALU64/16. DTAS rules encode two ALU styles: an integrated style, in which all operations are generated

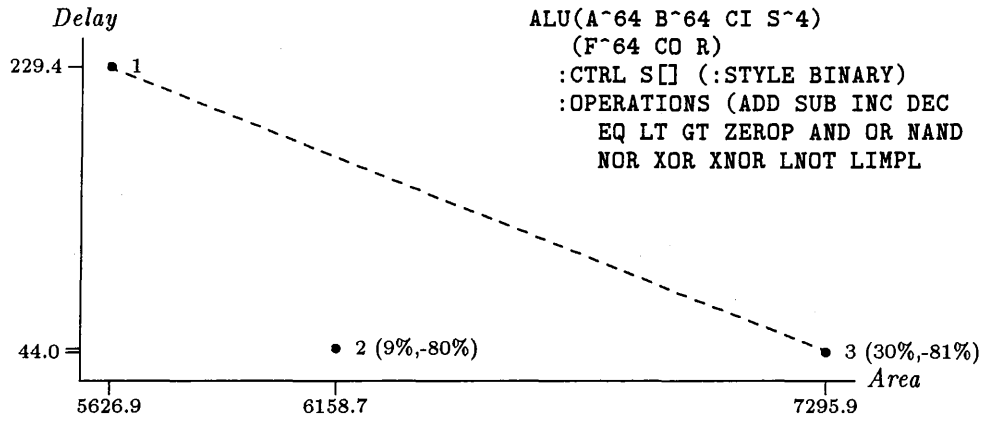

Figure 11: Design space for ALU64/16

by external logic around an adder with carry enable, and a segregated style, in which arithmetic and comparison operations are separated from logic operations, where the former are generated by an adder (without carry enable) and the latter by a function generator; outputs are combined through a multiplexer. Design 1 uses the integrated style with a ripple carry adder; Design 2 uses the integrated style with a full carry look-ahead adder; Design 3 uses the segregated style with a full carry look-ahead adder. Design 2 makes a very favorable tradeoff between Designs 1 and 3. In this experiment, the cutoff was set to the baseline. The wall-clock time required by DTAS was 3859 seconds.

Although DTAS performs technology mapping via functional decomposition and by comparison of functional specifications, we do not feel that this is in competition with the logic optimization and graph-matching techniques found in mainstream logic synthesis. Rather, we feel that it is complementary. DTAS can generate a range of designs for complex RTL components, such as MULT24 and ALU64/16, whose Boolean description would overwhelm the MISII logic optimizer. MISII can then be used to optimize subcomponents of these designs.

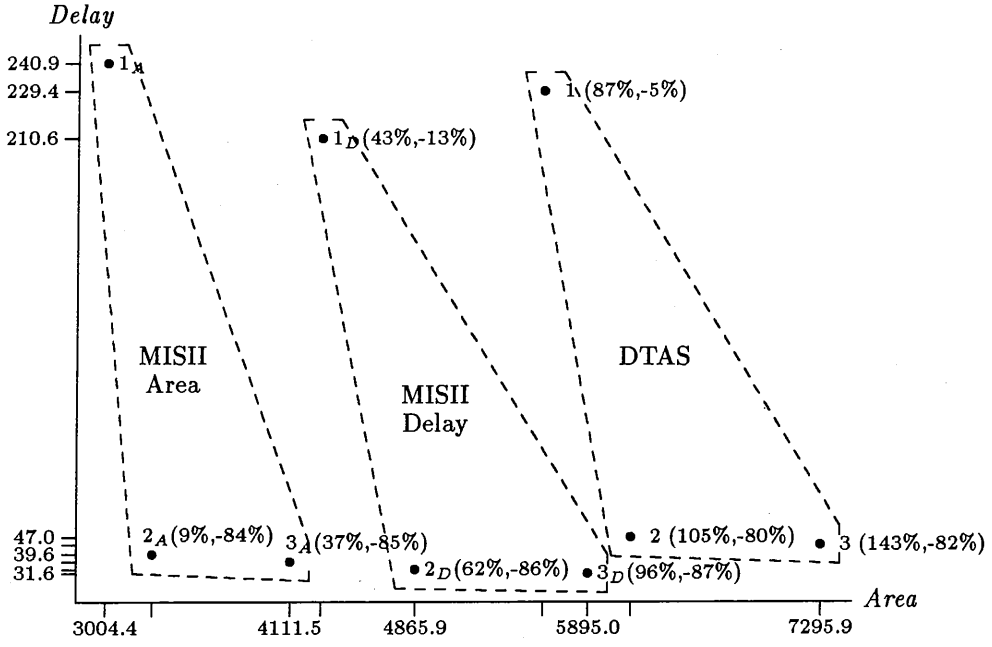

Figure 12: Passing ALU64/16 designs through MISII

The graph seen in Fig. 12 plots the results of applying MISII to ALU64/16. For Designs 1, 2, and 3 of ALU64/16 from Fig. 11, MISII was applied to those subcomponents whose netlists consisted entirely of configurations of Boolean gates. We used this approach twice: once to optimize for area, resulting in Designs 1_A, 2_A, and 3_A, and again to optimize for delay, resulting in Designs 1_D, 2_D, and 3_D. As indicated in Fig. 12, although MISII improved the delay and area characteristics of the DTAS-generated designs, the tradeoff between the three designs remain proportionally similar.

6 Conclusions

In this paper, we have introduced a functional-decomposition approach to technology mapping for large RTL data path components and described its implementation in DTAS.

Our claim is that by dynamically examining the effects of mixing design styles, we can generate a range of designs that make favorable and desirable tradeoffs between area and delay. We have presented results that validate this claim on two complex components, a 24-bit multiplier and a 64-bit ALU. We have also presented results that show how our approach can be integrated with mainstream logic synthesis technology to produce even better designs.

The DTAS Design Language/System is implemented in Common Lisp on a Sun-3 workstation environment. The designs generated for the three examples presented in Sec. 5 required 85 decomposition rules. Our future efforts will emphasize the use of RTL library cells and on techniques for maintaining technology independence within the DTAS framework.

References

[1] J. Rabaey, H. de Man, J. Vanhoof, G. Goossens, and F. Catthoor, "CATHEDRAL-II: A synthesis system for multiprocessor DSP systems," in *Silicon Compilers* (D. D. Gajski, ed.), pp. 311–360, Reading, MA: Addison-Wesley, 1988.

[2] R. K. Brayton, R. Camposano, G. De Micheli, R. H. J. M. Otten, and J. van Eijndhoven, "The yorktown silicon compiler system," in *Silicon Compilers* (D. D. Gajski, ed.), pp. 204–310, Reading, MA: Addison-Wesley, 1988.

[3] R. Camposano and L. H. Trevillyan, "The integration of logic synthesis and high-level synthesis," in *Proceedings of the International Symposium of Circuits and Systems*, pp. 744–747, 1989.

[4] N. D. Dutt and J. R. Kipps, "Bridging high-level synthesis to rlt technology libraries," in *Proceedings of the 28th Design Automation Conference*, 1991.

[5] J. R. Kipps and D. D. Gajski, "The role of learning in logic synthesis," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 4, pp. 167–180, June 1990.

[6] E. Detjens, G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "Technology mapping in mis," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD-87)*, pp. 116–119, November 1987.

[7] N. Dutt, "GENUS: A generic component library for high-level synthesis," Tech. Rep. 88-22, Department of Information and Computer Science, University of California, Irvine, September 1988.

[8] R. Lisanke, *Logic Synthesis and Optimization Benchmarks.* Research Triangle Park, NC: Microelectronics Center of North Carolina, 1988.