

Visualizing the Behavior of Logic Synthesis Algorithms

Howard A. Landman

Toshiba America Electronic Components, Inc.
San Jose, California

LandmH@taec.com

ABSTRACT

We present a system for generating high-resolution graphs of the behavior of synthesis tools as their input parameters are varied. The behavior with respect to changing delay constraint is studied, first theoretically, and then based on data from actual synthesis. The results are somewhat surprising, and shed light on why it seems to take some “luck” to get a good synthesis result.

.

1.0 Introduction: The Unpredictability of Synthesis

One of the biggest sources of frustration for users of synthesis tools is that the tools often seem to behave unpredictably. A small change to an RTL design may produce a large change in gates. Asking for a faster result may produce a slower one. Setting a particular synthesis flag may give better results at one time, and then later give worse results. There seems to be a certain amount of “luck” involved.

This problem is perhaps most acute during benchmarking. We would like to have precise measures of whether one setting of synthesis flags is better than another, whether one RTL architectural gives a faster netlist than another, and so on. But if there is a large random component in the results, and we can’t quantify it, then we can’t be certain that any of our conclusions are true, or even estimate the probability that they are true.

Many people try to limit this problem by using a large number of different test cases. This can help because the expected total error only goes up as the square root of the number of test cases; so, the average error per case goes down as the number of test cases increases. However, there are still problems with this approach. In order to reduce the average error by one half, you need to run four times as many test cases, and so on (the square root works against you). More importantly, most people don’t even have any idea of what their error is! They’re hoping that they’ve run enough test cases to make it “negligible”, but they don’t actually try to measure it.

This paper takes a different approach. Rather than simply taking more test cases, we investigate how to understand a single test case in great detail, with a reasonable expenditure of synthesis resources.

2.0 The Behavior of Synthesis Tools

2.1 Input Parameters for Synthesis

Synthesis is a complex process with many input parameters that can be varied. Some of these parameters are discrete, and others are continuous.

Discrete parameters include such things as RTL architecture, logic function, choice of synthesis library, settings of various synthesis switches, and choice of tool or algorithm within tool.

Continuous parameters are those which are represented by floating point numbers, such as input and output delays, capacitive loading, and time constraints. It’s amusing to note that, because FP numbers are actually stored as patterns of bits, they also have only a finite number of discrete values that they can take on! In a digital computer program, nothing is really continuous. But we think of these parameters as continuous, so it’s still worthwhile to distinguish them.

2.2 Simplifying the Problem

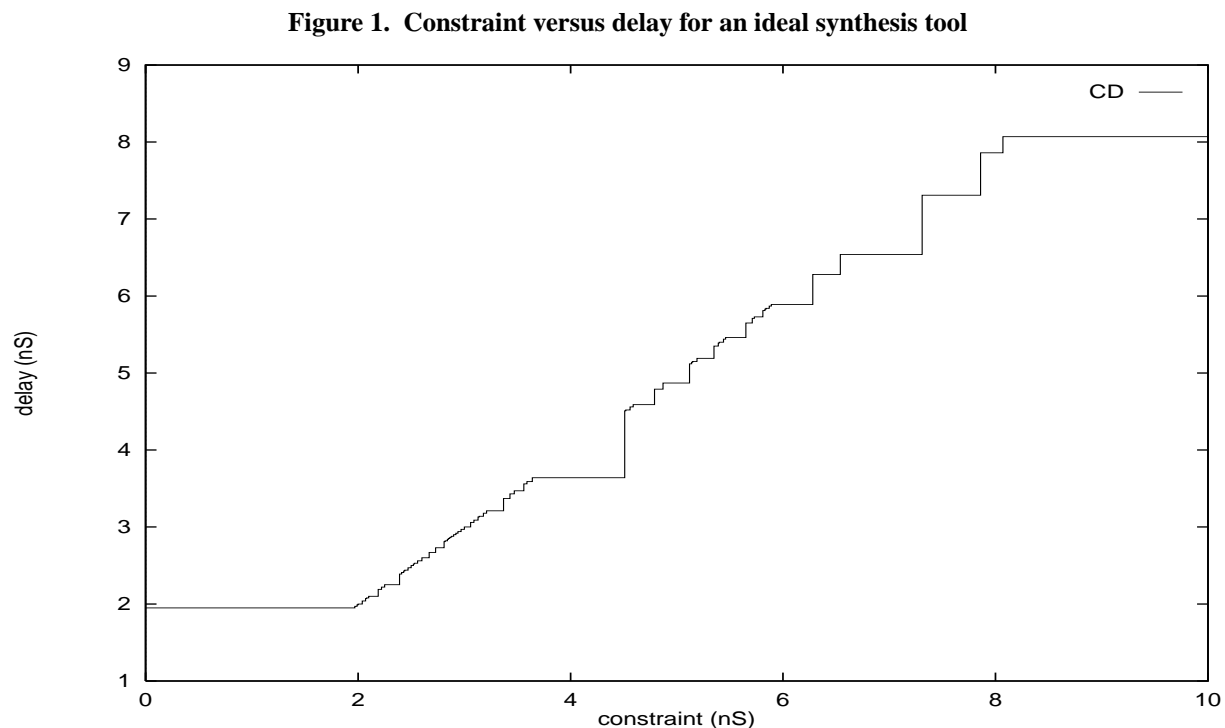
Now, if we allow all of those parameters to vary at once, we have a very messy multi-dimensional problem. So, to keep things simple, let’s assume we’re synthesizing a given RTL, and that we’re mainly interested in the speed and area of the resulting netlist. The question then becomes, how can we see and understand the behavior of the synthesis algorithm for that RTL?

It's not enough to try one synthesis, because of the "luck" factor. What we'd really like to see is the whole range of solutions produced under different time constraints. Of course, time constraints are multidimensional too; up to one dimension for each pin. But often there's one master time constraint, such as clock period, which controls all the others. In this case we can parameterize all other time constraints so that they vary as a function of the master constraint.

2.3 CDA Space

Once we accept this restriction, then the set of synthesis results (netlists) can be mapped into a 3-dimensional space, few enough to visualize. We have one continuous input parameter (the constraint), which gives one dimension, and two output values (delay and area), which give two more. I call this space CDA Space (for Constraint-Delay-Area). We can also plot any two of these dimensions on a screen or piece of paper. Each of these plots is a projection of the 3D data set into 2D. There are three such pairs.

The CD graph plots Constraint versus Delay. This shows you how the *actual* delay of the synthesized netlist changes as you vary the *desired* delay (the constraint). The following figure shows how an ideal synthesis tool might behave. (The "ideal" was made by collecting data from multiple synthesis tools and option settings for the same function, and throwing out bad points.)

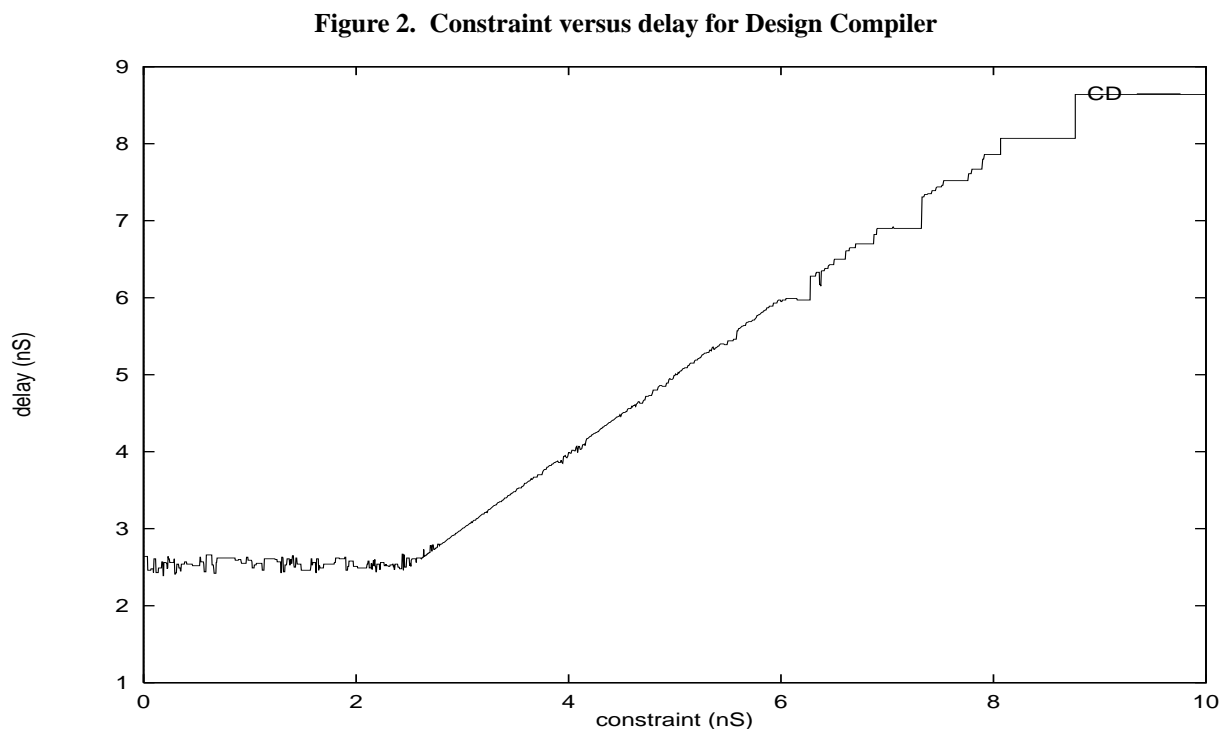


The 45-degree line on this plot indicates where the delay equals the constraint. If you're below the line, you're meeting your constraint; if above, you have a timing violation. The horizontal line segment in the upper right represents the netlist with the smallest possible area for the given function. This line extends off to infinity at the right; once your constraint is loose enough that the smallest possible netlist meets timing anyway, making it looser shouldn't have any effect.

As we follow this line left, at some point it hits the 45-degree line. At this constraint, the smallest netlist is no longer fast enough. So, our ideal synthesis tool will switch to the next smallest netlist which meets the given timing. This new netlist will be faster than the previous one (because it meets timing and the other doesn't). So, the CD curve makes a discontinuous jump at this point (a *breakpoint*) where the netlist changes. But after that, it stays constant again until we reach a constraint where the new netlist becomes too slow. At that point, we jump to an even faster (and larger) netlist. This continues until we finally reach the fastest possible netlist. At that point, the tool can't do any better, and (for the first time) it fails to meet timing. The fastest netlist gives a horizontal line that reaches all the way to constraint zero.

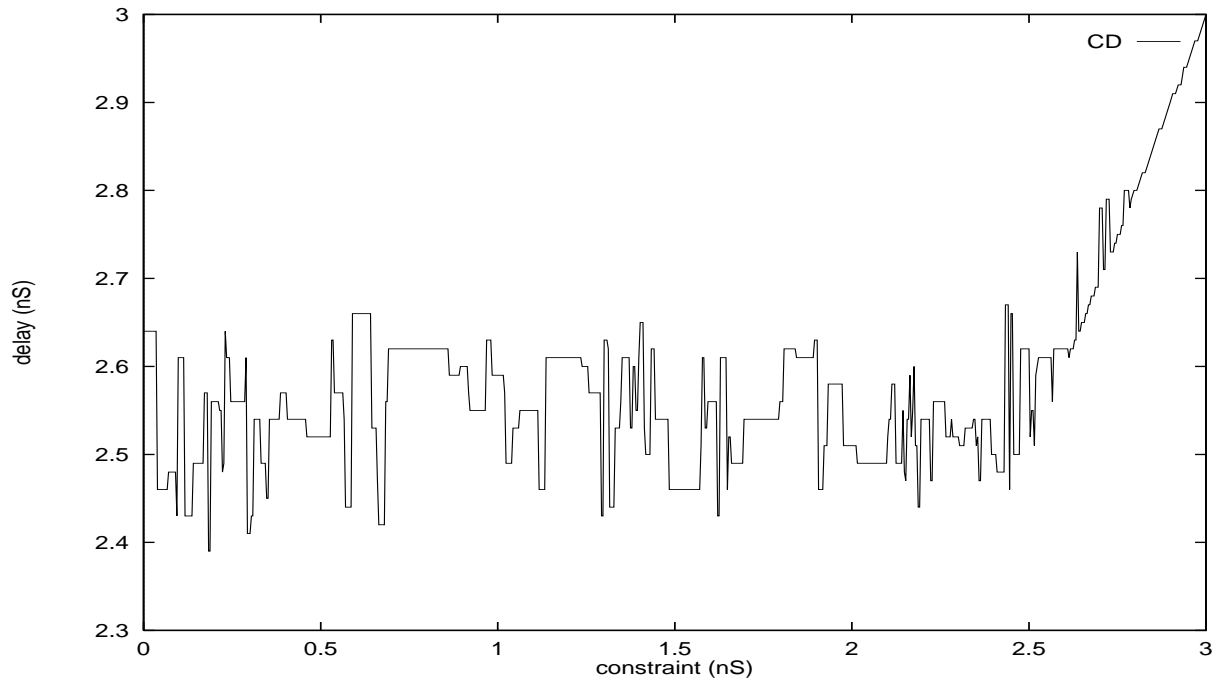
In summary, the CD curve should consist of a set of horizontal line segments (in math terms it's "piecewise constant"), with each segment corresponding to a different netlist that was optimal for some range of time constraints. The segments should form a kind of irregular staircase following the 45-degree line, always just under it, except that the rightmost segment (for the smallest netlist) continues flat out to infinity, and the leftmost segment (for the fastest netlist) crosses above the 45 degree line and goes flat to zero.

But what happens when we synthesize using a real tool like Design Compiler (all data presented here is from DC 3.4b)? The curve is similar in general shape, but noisier. Particularly noticeable is the noise in the "fastest design" region at the lower left, which should theoretically be flat.



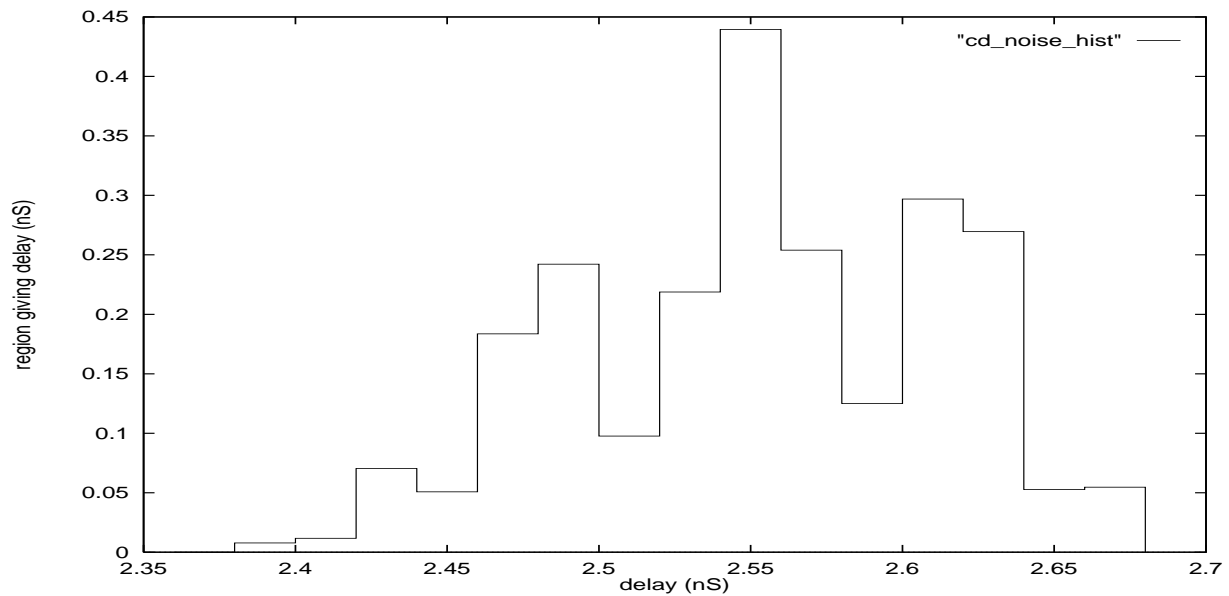
Let's take a closer look at that noise. Figure 3 shows a blowup of the previous plot.

Figure 3. Detail of previous figure



DC's delay behavior is not stable or monotonic in this region. There's a "noise band" of roughly 10% of the total delay. This explains the luck factor in timing optimization when constraints are hard or impossible to meet. We can also compute the mean and standard deviation of the "noise", and plot a histogram or distribution of it. Here's such a histogram for the above noise band:

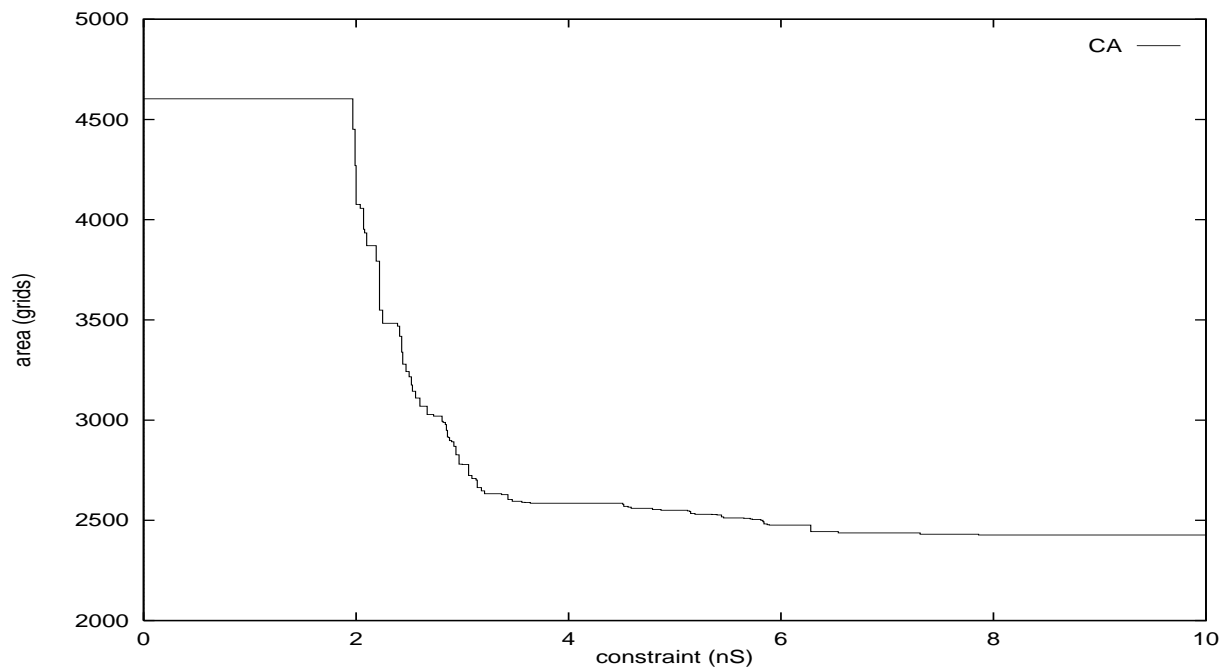
Figure 4. Histogram of delays in noisy region of previous figure



This lets us get answers to questions like "How often are synthesis results within 1% of the best the tool can do?" or "How many runs do I need to get a result better than X nS?"

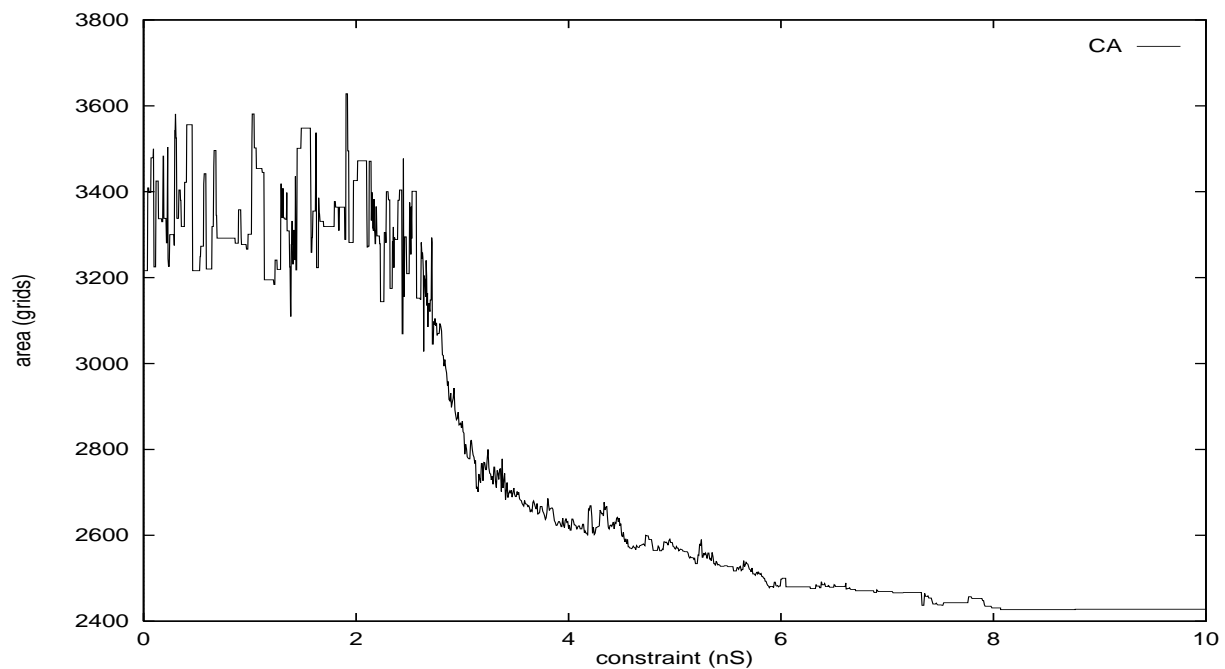
The CA graph plots Constraint versus Area. This shows you how the area of the synthesized netlist changes as you vary the constraint.

Figure 5. Constraint versus area for an ideal synthesis tool



This ideal graph too is piecewise constant. It decreases from left to right, because a looser timing constraint lets smaller netlists meet timing. The leftmost segment is again the fastest netlist, and the rightmost segment the smallest netlist. But area in Design Compiler also behaves chaotically:

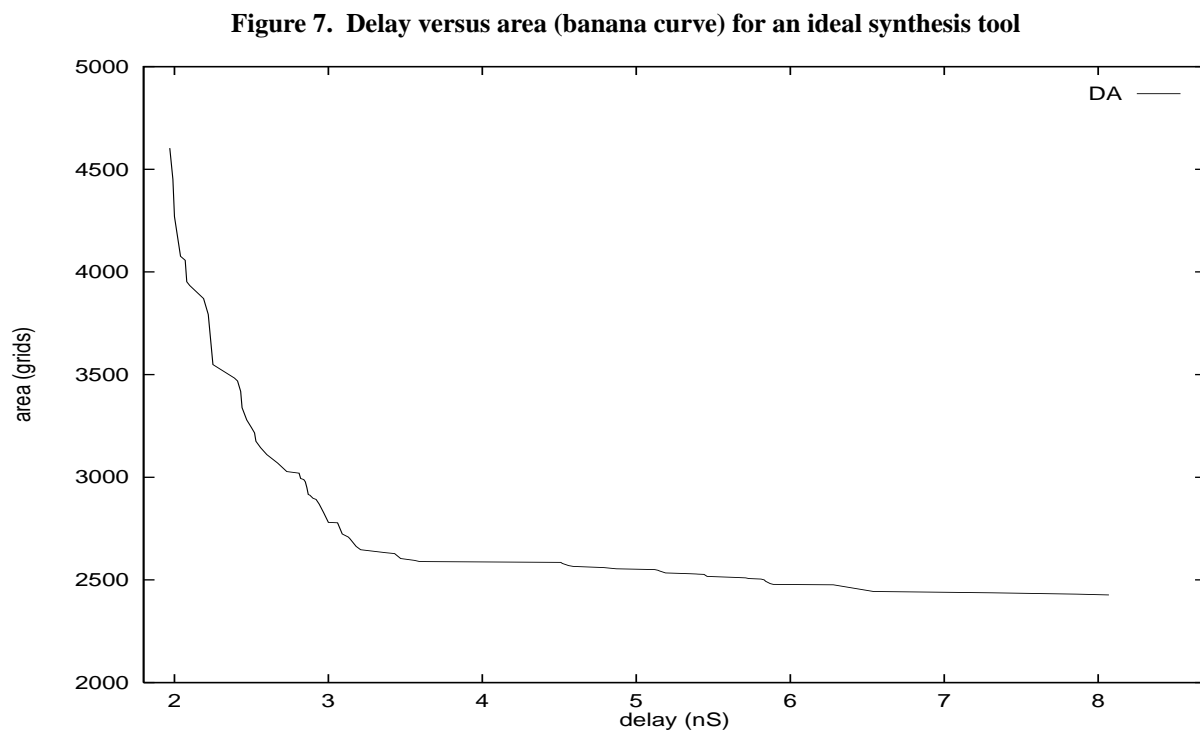
Figure 6. Constraint versus area for Design Compiler



The DA graph plots Delay versus Area. This shows you how the areas and delays of the synthesized netlists relate. This is commonly known as the “banana curve” or “area versus delay trade-off curve”, and every major synthesis vendor has marketing slides showing it.

This concept has survived years of exposure, and probably ranks as one of the dominant paradigms of modern logic synthesis. What’s surprising about this is that there are at least three ideas in it which are fundamentally wrong. It doesn’t take long to see that they must be wrong, but most people still believe them. I call them myths.

The three myths of the banana curve are: it’s a curve (i.e. it’s continuous), it’s monotonically decreasing, and it’s convex. But the reality is very different.

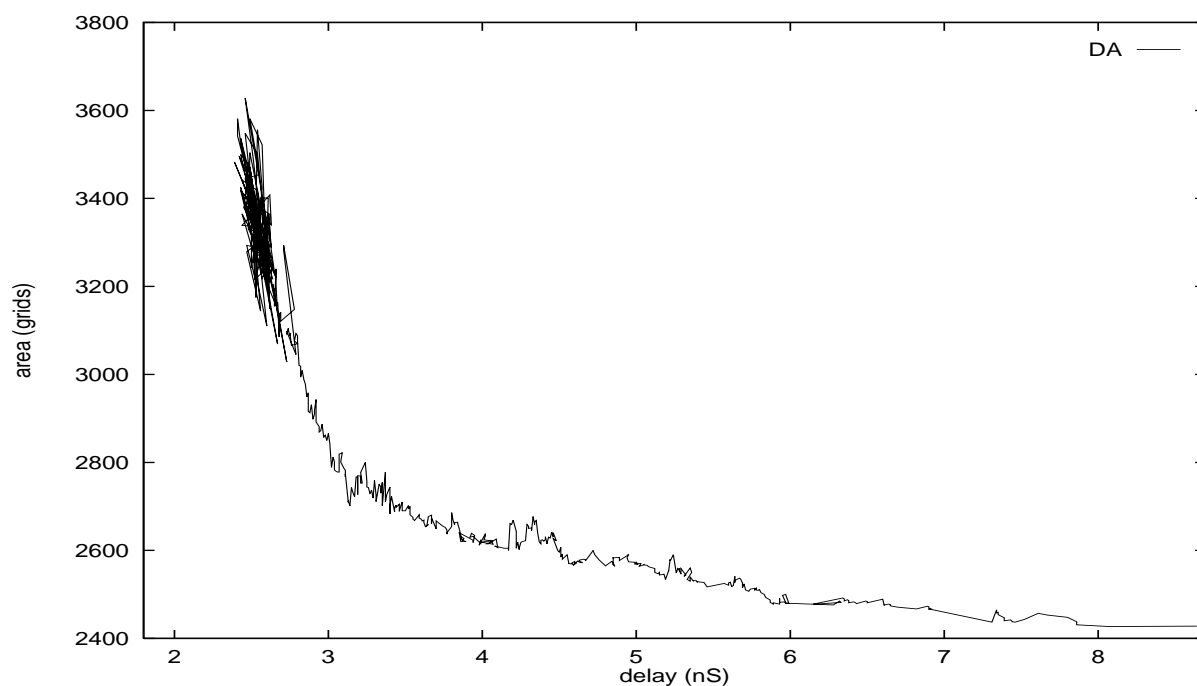


First of all, it’s not really a curve, but a collection of discrete points, each of them corresponding to a different netlist. In the above I connect each pair of points which correspond to the two netlists on each side of a breakpoint. But those lines have no meaning in terms of actual solutions.

In theory, a perfect synthesis tool should produce a set of points each of which is monotonically slower and smaller than the previous one. So, that myth has some basis in reality. However, real-world synthesis tools are not perfect, and may behave more strangely.

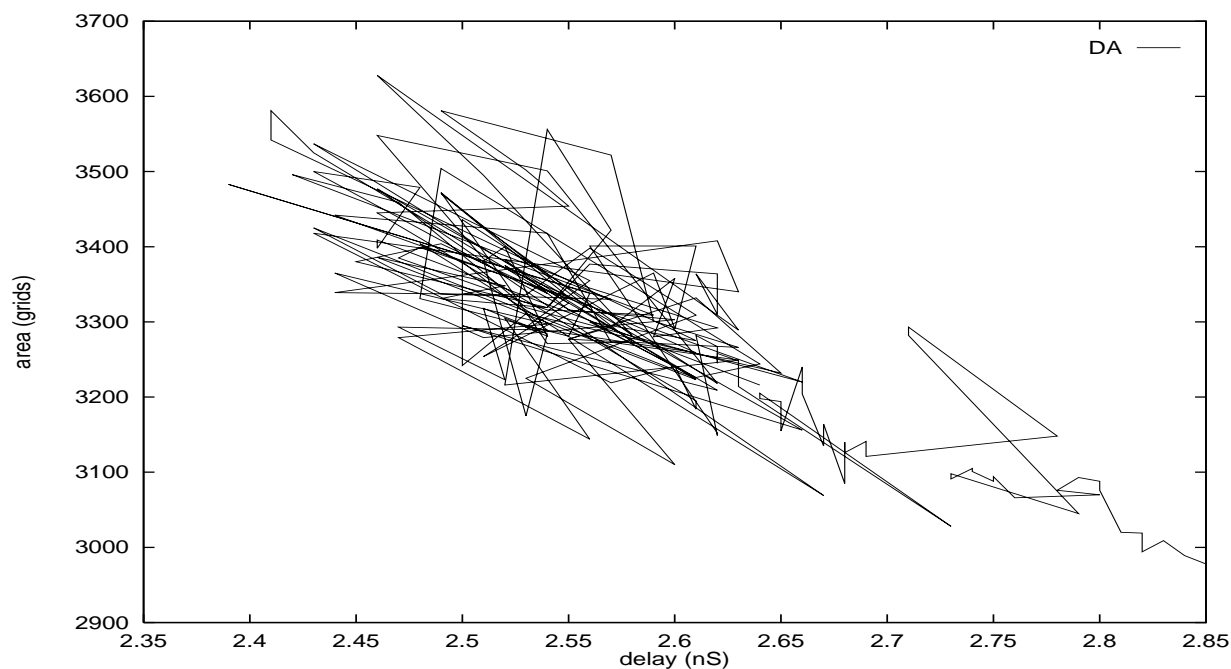
And finally, it’s not convex. There isn’t even any reason to think it should be. Convexity of a set of solutions is forced when any linear combination of two solutions is also a solution. That is, if we draw a line connecting two solutions, all the points on the line must also represent solutions. But you can’t average two netlists! The lines connecting the points here only indicate sequence of constraints, and have no other meaning.

Figure 8. Delay versus area for Design Compiler



This graph of actual DC results looks much bumpier. The curious “puff-ball” at the left shows the combined chaos in both area and delay. Once you try to push timing beyond a certain level, DC bounces around in a cloud of solutions, some of which are very good and some of which aren’t. And it isn’t just DC; other synthesis tools show qualitatively similar behavior. Perhaps this is not unusual for heuristic algorithms trying to solve an NP-complete problem.

Figure 9. Detail of previous figure



3.0 Gathering the Data

3.1 The Computational Dilemma

Gathering the data for the above graphs posed a difficult problem. In order to get a complete picture of how the algorithm behaves, we want to have data points over the whole constraint range, from zero to slower than the smallest netlist. To get a resolution of $1/N$ th of a nS by just synthesizing at every point would require $W*N$ synthesis runs, where W is the width (in nS) of the area to be studied and N is the number of points per nS . Each halving of the space between points appears to require a doubling of the number of runs. This quickly becomes impractical, because we usually are limited by availability of CPU cycles and synthesis licenses.

3.2 A Bold Assumption

However, much of the effort of this simple-minded approach is wasted. We expect that for very small changes in constraint, there will often be no change in the output. So, many of the runs above will be producing the exact same netlist.

Imagine that we have already completed some number of syntheses, and are considering doing more to improve the resolution. If we divide the constraint space into intervals at the points already synthesized, then for each interval, the netlists at the two endpoints of the interval are either identical or different. If they're different, doing another synthesis in that interval will always provide new information. At worst, if the result is the same as one of the interval's endpoints, it will narrow down the location of the breakpoint between the two netlists. (Reducing the position uncertainty by half is exactly one bit of information.) And if we're lucky, it might be a completely new netlist. But if the two endpoints are identical, then it's very likely that a new synthesis will only give the same netlist, and we will have learned almost nothing.

So, to maximize the information gained, we can make the following bold (and slightly risky) assumption: If two adjacent constraints produce *exactly* the same netlist (character for character), then probably all intermediate constraints will too. If we believe this, then we don't have to try any more constraints inside that interval. Instead, we can focus our synthesis runs on the intervals where the endpoints differ.

3.3 The Algorithm

Given that assumption, the algorithm is fairly straightforward:

- Determine the range of interest.
- Synthesize twice for the endpoints of the range. This gives us our first interval. Put it in a queue.
- Do a binary splitting search on the delay constraint. That is, take an interval from the queue, check to see if the endpoints differ, and if so, synthesize with constraint equal to the average of its endpoint constraints and put both resulting subintervals in the queue. Repeat until done.

Because this search adaptively targets only those areas of the graph that have actual netlist changes, it requires fewer synthesis runs to achieve any given resolution. In theory, the number of runs to resolve a single breakpoint to $1/N$ th of a nS is proportional to $\log_2(N)$. So the total number

of runs to resolve an entire graph should be about $K \cdot \log_2(N)$, where K is the number of breakpoints. Assuming each circuit has only a finite number of breakpoints, then this approach will be exponentially faster than the naive one in the limit of high resolution. (This assumption might not be true! Some numerical algorithms are known to have fractal basins of attraction, so there's no reason why a synthesis tool couldn't too.)

In practice, I've been able to get resolution as fine as 0.0000001 nS in a few thousand runs instead of the millions of runs that would be required by the naive approach. So, the speedup can be as much as a factor of 1,000 when you're trying for very high resolution.

3.4 Implementation Details

I implemented this system for Design Compiler using a Perl front end that talks to `dc_shell` through a pipe. This allowed more effective programming techniques, such as subroutines, which are not possible in the `dc_shell` command language. One tricky point to watch out for is buffering; if you use the default buffering method (which is block buffered for efficiency), the system may deadlock. The way to prevent this is:

```
open(DC,"|dc_shell") || die "Can't open a pipe to dc_shell" ;
select(DC) ;
$| = 1 ; # Flush after every write.
```

The above code only connects to `dc_shell`'s input, and doesn't let you read the output of `dc_shell` back into Perl. That can be done too, but is slightly more complicated (it involves opening 2 pipes, forking a child process, and having the child exec `dc_shell`). Instead, the script waits for certain output files to exist before continuing. The core of the search algorithm is only a few lines:

```
# breadth-first splitting search
sub split_interval
{
    local($l,$r) = @_ ;
    if (&netlist_diff($l,$r))
    {
        $m = ($l + $r) / 2.0 ;
        &compile($m) ;
        &schedule($l,$m) ;
        &schedule($m,$r) ;
    }
    return 1 ;
}

sub do_todo
{
    while (@todo)
    {
        $next = shift(@todo) ;
        print STDERR "Splitting $next\n" ;
        &split_interval(split(' ', $next)) ;
    }
}
```

The `do_todo` routine just loops while there is anything in the `@todo` queue, popping off the first item and calling `split_interval` on it. Each item on the queue is a pair of constraint values (smaller first). Each call to `split_interval` checks that the netlists produced by the endpoint constraints aren't identical, and then does a compile on the midpoint of the interval and schedules the two new smaller intervals to be split later. The above code does not make use of Perl 5's list-of-lists feature, so it can be run under Perl 4 too.

I saved all the netlists and area and timing reports. Disk space is cheap. The method to tell whether two netlists were identical was just to check their size (if they're different size they're obviously different), and then compare them with the Unix command `cmp` if they were the same size.

All the graphs were generated using `gnuplot`, an excellent piece of free software. It is capable of plotting to your screen, to a color or monochrome PostScript printer, or (in the case of this paper) to an EPS file which can be imported into document editors.

4.0 In Conclusion

We've seen that it's possible to get high-resolution pictures of the behavior of logic synthesis tools without doing millions of synthesis runs. Such pictures answer some questions, and raise others. The understanding gained from them can lead to wiser tool use and more realistic expectations about tool behavior.

We've seen that Design Compiler behaves somewhat chaotically with respect to delay constraint when asked to make a circuit faster than it knows how. This chaos explains the "luck" factor in timing optimization. Getting the best result is therefore unlikely in only one synthesis run. The prudent designer will plan accordingly.

An earlier version of this paper was presented to Synopsys engineering personnel. Don Chan of Synopsys implemented his own version of this system, and is using it to help evaluate new releases. I hope that, over time, this will help Design Compiler become more predictable.

5.0 Acknowledgments

This work was largely inspired by Edward Tufte's three magnificent books on the use of graphics to communicate information:

- *The Visual Display of Quantitative Information*
- *Envisioning Information*
- *Visual Explanations*

These books are essential reading for anyone who analyzes data or makes or uses graphs. Tufte believes in letting the data speak for itself, and I've tried to do that.