Fitting Algorithms, Seeds, and Variation

The Spice of Life

Version 1.0 - November 1st, 2011 - Quartus II 11.0SP1 - by Ryan Scoville

Introduction:

For most designers, the fitter is a black-box. They hit play, wait for what seems to be too long of a time, and then analyze the results, only to find their small change to unrelated logic caused problems elsewhere in the design. This document's purpose is to give a simple overview of the solution space the fitter must deal with, and the algorithms it uses to optimize a design within that space. With that understanding, we will see how variance occurs from compile to compile, and how one can use seed sweeping to understand and possibly take advantage of that variance.

Table of Contents

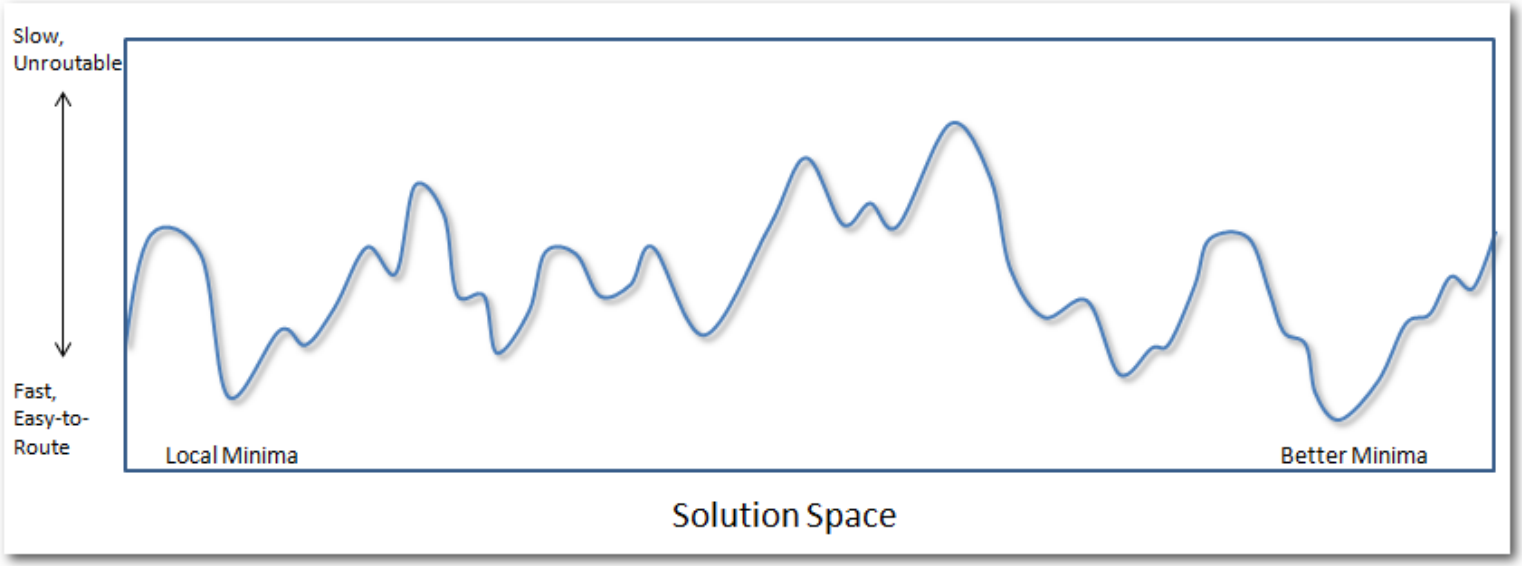
The Placement Solution Space

Before dealing with how the fitter works, it's important to first understand what the solution space looks like. Let's start with a mid-density Stratix IV device, the EP4S230, which has 182,400 registers in the fabric(not counting I/O registers, DSP registers or Embedded Memory registers). If we were to fit a design containing only 1 register, there are 182,400 different solutions to that problem. A simple design of only 2 registers would increase the number of solutions to 182,400 X 182,399 = 33,269,577,600. A design with three registers has 6,068,304,415,084,800 different solutions(that's six quadrillion). A quick google search says there are ~9.4 x 10^79 atoms in the observable universe. As soon as our design has grown to 16 registers, the number of solutions for fitting those 16 registers has grown larger than the number of atoms in the observable universe. As can be seen, the solution space grows incredibly large, incredibly fast, and so a normal design for that device, which might have 75,000 registers, has an incredibly large number of solutions.

Hopefully that gives a little "sympathy for the fitter", in that it must quickly work through an enormous solution space to find a good result. This also helps explain why a small change in one portion of the design causes different results on other parts. The reason is that the solution space is so large, that as soon as the fitter makes one move differently, it will suddenly be in a completely different area of the solution space, and hence comes up with a different solution.

Of course, just because there are lots of solutions, the fitter needs criteria to determine the quality of fit. The two primary critera are performance(slack) and routability. Most of the time these two go hand-in-hand, whereby putting connected elements closer together will reduce the amount of routing required and increase the slack on all paths going through that connection, but there are times when these goals will conflict. Also note that there are other criteria, such as reducing power, but slack and routability are by far the major drivers.

Let's look at a sample view of a solution space with these criteria:



This two dimensional graph represents a tiny, over-simplified, portion of a solution space. The lower the point, the faster the design will run and the easier it will be to route. In reality, the solution space has many, many dimensions and would be exponentially larger. As can be seen, there are peaks and valleys, and it's the fitters goal to try and find the lowest point in the solution space.

Now that we have criteria, design of similar size, which would have a similar number of solutions, may have different looking solution spaces. One design might have a small section of code that is timing critical, while the rest of the design easily meets timing. This design might have very small variation, as the fitter always optimizes this one section and the rest of the design, no matter how differently it gets placed from compile to compile, has no affect on the final slack.

Another design of similar size may have many blocks throughout that are all timing critical, all competing for better placement and faster routing resources. Compile to compile, the fitter will find different solutions whose critical paths jump around between various hierarchies in the design. High-speed designs, where tens of thousands of paths all have few levels of logic and all need very good placement and routing, are especially prone to this, since a single long route on any of these thousands of paths can make that path the critical path.

Annealing

With this enormous solution space, how does the fitter converge on a solution? Quartus II uses an annealing algorithm, which was originally used to simulate the cooling of metals, which I'll briefly describe. Metals are initially heated into a liquid so they can be shaped to the desired product, and are then cooled back into a solid. If cooled to quickly, the atoms will arrange in non-ideal structure that will become a stress point of the metal, i.e. a weak point at which it could break. The goal then is to cool the metal as quickly as possible and yet achieve a required strength. A description on annealing can be found here:

http://en.wikipedia.org/wiki/Annealing_(metallurgy)

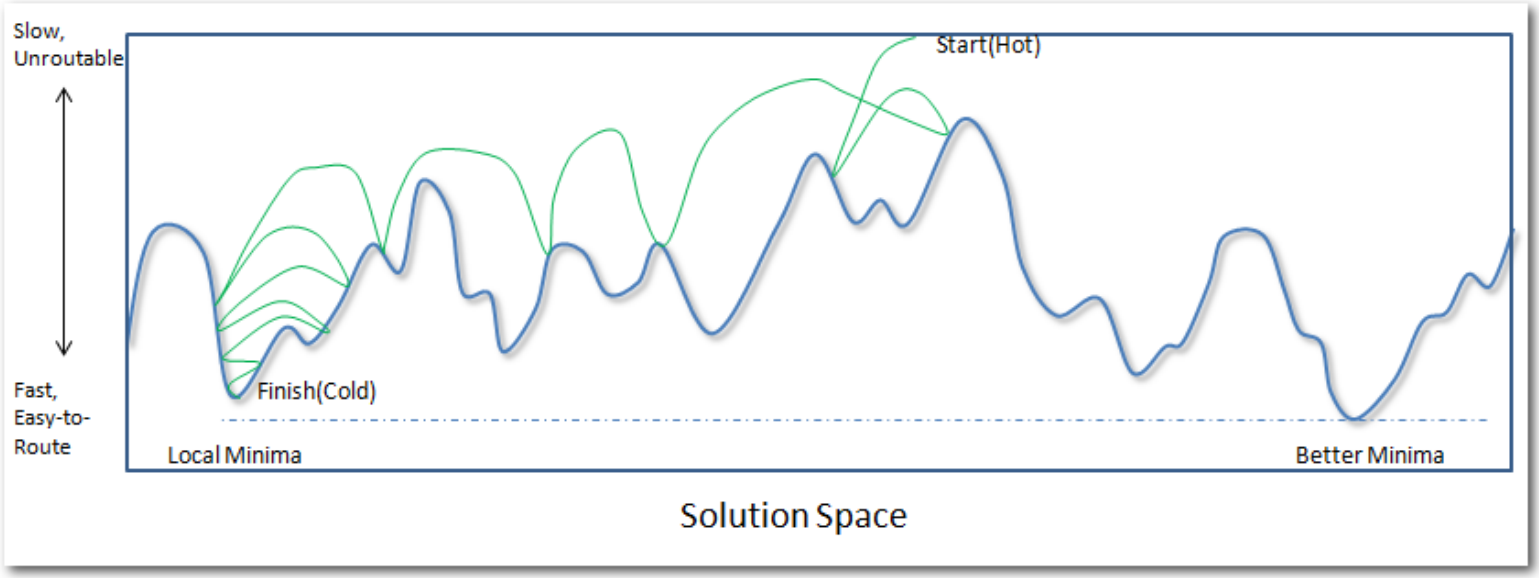
A description of Simulated Annealing algorithms can be found here:

http://en.wikipedia.org/wiki/Simulated_annealing

The fitter's placer uses simulated annealing. The fitter initially throws everything into the device in a legal placement. Legal means no space will contain an illegal element, like a FF in a LUT location, and no space will contain more than one element. This placement is legal, but it is a complete mess that is completely unroutable and would have terrible timing. At this point the fit solution is extremely "hot" and the fitter will move elements around, always maintaining a legal placement, as it cools toward a better fit.

This explanation of how the placer works is significantly over-simplified. With every release of Quartus II, some of Altera's best engineers work on optimizing the fitter for better performance and

faster compile times while always preparing for new architectures. But keeping things simple, lets look at an illustration of the fitter working in that solution space:



The green line exhibits the fitter's moves within this solution space. The fitter is constantly searching for a lower(better) point in the solution space. In this example it finished at a good local minima, but it turns out a better one was nearby. Note that I don't say Best Minima, because in general the solution space is so large that it is impossible to ever find the "best solution" and instead we are just concerned with finding better minima.

Annealing Points of Interest

Although the previous drawing is quite primitive, there are quite a few interesting things we can derive from it.

Temperature – Note that the fitter's line moves up, which means it has worse results, before going back down. This is an interesting attribute of annealing algorithms. The fitter accepts bad moves, i.e. moves that might make the timing or routability worse. The picture easily shows why, since if the fitter did not accept bad moves that pull it out of a local minima, it would have been stuck in the initial valley below its Start point and never found a better solution. From a placement perspective, think of a core in the design that may work best by being placed in the bottom left corner, but most of its nodes are initially placed in the top-right. By moving a single element to the bottom left, the overall results may be worse, since that node is still connected to elements in the top-right. But as more and more nodes are moved to the bottom left, the fitter is able to move out of its local minima, which may not have given very good results, to a better area in the solution space.

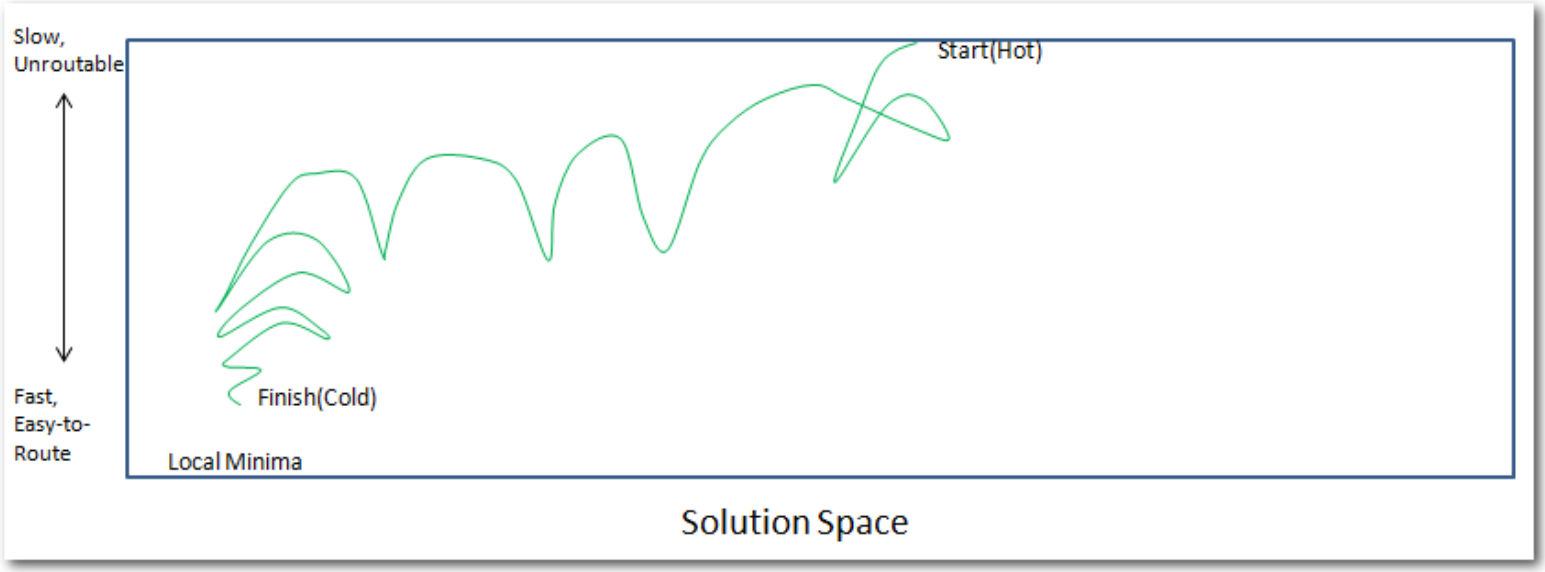
The temperature determines how often the fitter accepts bad moves. Early on when the device is hot, the fitter is more willing to accept bad moves. As the device cools, it accepts fewer and fewer bad moves. The annealing table determines how slow or fast the fitter moves from hot to cold.

Fitter Settings – In the menu Assignments -> Settings -> Fitter Settings -> More Settings -> Placement Effort Multiplier, the user can directly control how quickly the fitter cools. The default is 1, and as the user increases the number, the fitter will proceed from hot to cold more gradually. When I modify this setting I usually choose a setting of 2 or 4. The only major trade-off is compile time, and note that the returns quickly diminish. Anything above 8 will have the fitter bouncing around the solution space for a much longer time but barely increases the likelihood of finding a better solution. Increasing the Placement Effort Multiplier does not guarantee any specific compile will be better than a compile with a lower Placement Effort, it just increases the odds that this will happen.

Also, since the placer is also working on routability, increasing the placement effort can make a design more routable. It may be counter-intuitive at first, but if a design has routing issues, one of the main controls to help is to increase the placement effort.

Another point of interest is how the fitter effort levels, "Standard", "Auto" and "Fast", found under Assignments -> Settings -> Fitter Settings, deal with annealing. The above diagram represents the Standard level, in which the fitter will run all its algorithms and finish at a local minima. The Auto Fit works similar, but once the fitter determines the design is routable and meets timing with margin, it stops. Because of this, it won't necessarily finish at the the local minima, but it saves compile time while still meeting the user's requirements. Designs that don't meet timing will never stop early, and hence should have similar results between Auto Fit and Standard Fit. Fast Fit works like Standard Fit except it moves from hot to cold much more quickly(it's not just a low Placement Effort though, as there are a number of optimizations done under the hood to make this run quickly). Fast Fit will give worse results, although usually not significantly worse, while achieving considerable savings in fitter time.

Blind Man – An important point is that the fitter has no idea what the solution space looks like. In our example, it looks obvious that the fitter should have gone to the right and found the Better Minima, but in reality it has no idea a Better Minima exists without fully exploring that section of the Solution Space. In fact, as far as the fitter is concerned, this is what the solution space looks like:



The only thing the fitter knows about is where it's at, and so the fitter works much like a blind man wandering through an enormous house, only knowing where a wall is once he's bumped into it. This analogy makes the fitter seem inefficient, but…

Annealing Efficiency – Annealing algorithms are excellent at find a good solution very quickly. They are good at dealing with different types of problems too, which is important since each design is essentially a different problem, and any hand-tuning for one type of design will often cause problems or inefficiencies for other designs. That being said, if there were a single "best case" solution, an annealing algorithm is unlikely to find it, although there aren't any algorithms that will find the best solution unless it's a trivial solution.

Hierarchy – Just thought I would point it out, but the fitter works in a very different manner than how people approach the problem. We tend to think of a design in terms of hierarchy, such as "the fitter should put the DDR3 interface on the bottom, then have it connected to the ingress path above it, which then connects to the control block…". To the fitter, the design is basically flat, with a bunch of nodes that have connections and timing constraints, and approaches placement from this low level view. (Note: There are some decisions based on hierarchy, but not a lot). Now, the two approaches often converge on similar solutions, but the fitter is often able to find solutions that are more subtle. A hierarchy might be split up if it has a natural division point that easily meets timing. The fitter may achieve better results by interlacing hierarchy placement, rather than cordoning them off into separate areas.

That being said, if a user's hierarchical view is really the best solution, than some up-front floorplanning with LogicLock can often help guide the fitter into a better solution early on, allowing it to fine-tune from there. I will discuss floorplanning in more detail in a follow-up document, but it's nice to see that most users' top-down view of fitting differs from the fitter's bottom-up approach.

Cold Stop – Note that the green line finishes at the very bottom of a local minima. This means the fitter does not see any moves that could improve timing, and the only possibility of getting better timing would be to re-heat the device to a much worse fit and try to bounce into another local minima. This is a very compute intensive task, and the fitter is just as likely to find itself in a worse solution instead of a better one. Remember that the fitter has no idea what the solution space looks like besides where it has already been. The question comes up where a user may fail timing by a small amount, say -20ps slack, and ask why the fitter couldn't try just a little bit harder to improve timing. The answer is that the fitter already has tried "a little bit harder" and can't see any moves that would improve timing. If it ripped up a little logic, it would just work its way back to the same local minima it is already in.

The only solution is to rip large amounts of logic. If you go back to the original solution space, which does have a better minima on the right side, the only time the fitter is near that is at the very beginning, so it's not just a small move that would get better results, but almost a complete re-fit. That is where seed sweeping comes into play.

Other Sources of Variation

Although the enormous size of the solution space is a major source of variance, there are others. For example, the router runs after placement, and so two designs with identical placement might have different routing and hence different results. I am not going into the router since the user has almost no control over routing, at least on a large scale.

Timing estimation is another source of variance. Final timing sign off is a compute intensive process, where Quartus II actually runs its own calculations similar to a HSPICE simulation. Since annealing algorithms require timing analysis to be run over and over, it doesn't have time to run a full TimeQuest run, and instead relies on estimates. An interesting example is that route times can be affected by what is routed near them, so a partition that has its placement and routing locked down can still see some slight variation in its timing. This variation is generally less than tens of nanoseconds, but still present.

A straightforward way to think of it are three components run in series, each attributing variation along the way.

Placement(most variation) -> Router(some variation) -> TimeQuest(least variation)

One thing that has been asked is if a fitter could be made with less variation. I believe the answer is yes, but it would be at the expense of performance. In other words, it could get consistent performance, but it would be consistently worse than the current fitter, which obviously isn't worth the trade-off. Another way to look at the issue though is if your design can consistently meet timing, so that the worst results from variation are still good enough, then you have a design that is essentially immune from the effects of fitter variation. I know this is easier said than done, as designs are constantly having new logic added, timing requirements keep going up, and as soon as you feel like your design is in good shape you end up being told that you must go with a slower speed grade to save costs. It's a constant battle.

Seeds

What is a seed?

So what does a user do that causes variation in a design? Simply put, pretty much everything that goes into a compile is a potential source of variation:

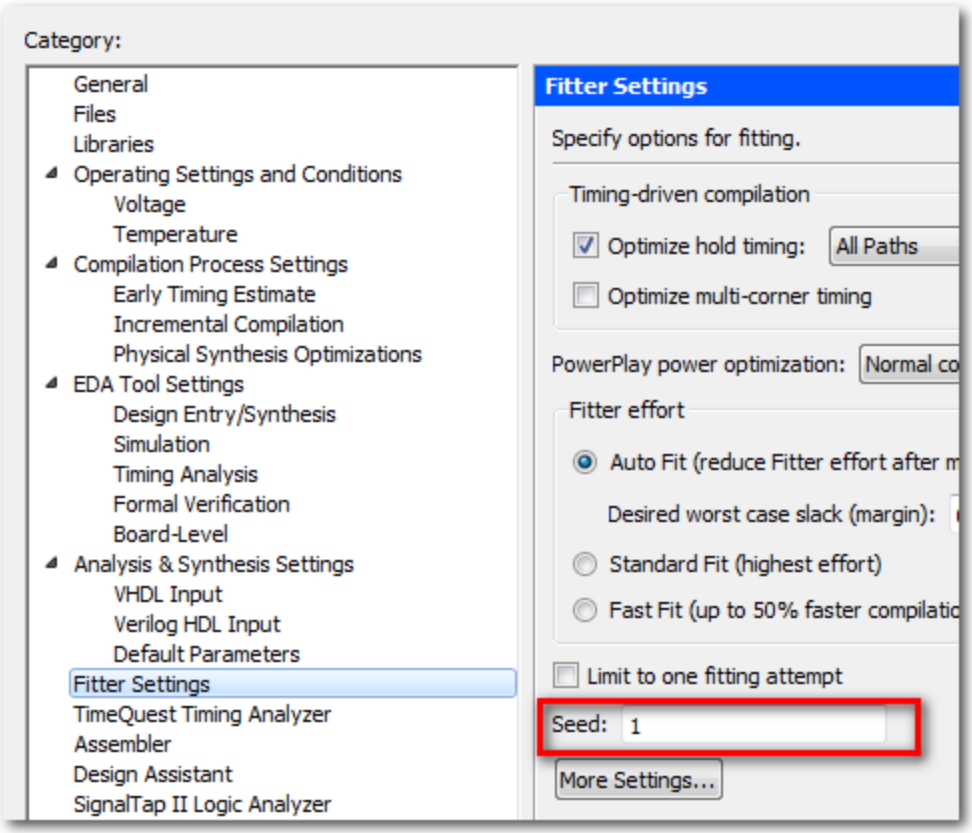
- All design files, e.g. VHDL, Verilog, Schematics, EDIFs, etc.
- All timing constraints in .sdc files
- All assignments in the .qsf
- Version of Quartus II
- OS(Linux and Windows give different results due to a minor difference in floating point results)

As discussed, the smallest change up front will cause the fitter to make a slightly different move, sending it into a different region of the massive solution space and ultimately ending up with a different result. These small changes can come from anything, moving a top-level I/O location, changing a timing constraint or simply changing the name of a net in the design.

Now, all of these either have a purpose in the design(design files, timing constraints, Quartus II constraints) or are something the user is stuck with(Quartus II version, OS). So if the user wants to see if a recompile will give them different results, they could modify about any one of these, but it would be better to have something that has no other purpose than to cause variation. That is why the Seed setting was added, so the user could nudge their design into a different area of the solution space and hence get different results, without modifying something in their design that had a purpose.

Side note: In the days before there was a seed setting, I remember randomly changing net names in HDL files, modifying timing constraints by 1ps, and other changes that I didn't want to make but were the only way to perturb the design into another area of the solution space.

The seed assignment can be found under Quartus II Assignments -> Settings -> Fitter Settings:



It is also in the .qsf like so:

set_global_assignment –name seed 1

If the user has never modified the seed from its default 1, it will not show up in the .qsf.

The Magic Seed

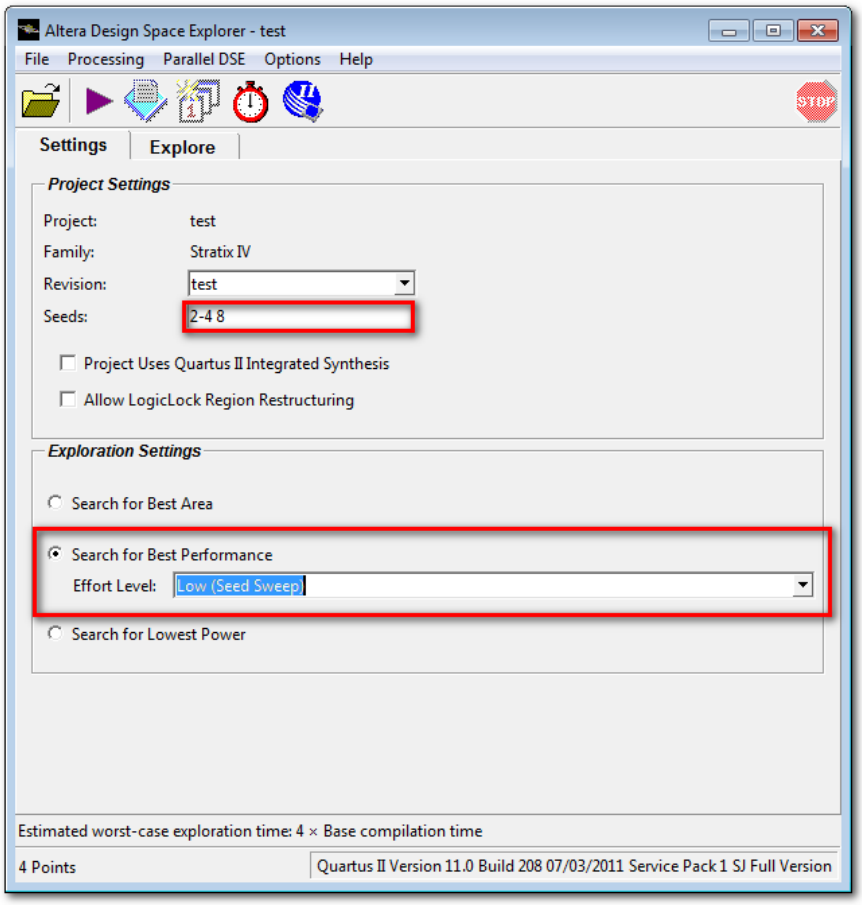
I've been asked innumerable times what the best seed is, and have often been told by users that they had found the magic seed for their design. The problem, I believe, is that there is too much concentration on just the seed value. Remember that the results of a compile are based on all inputs:

Results = HDL + Schematics + .sdc + .qsf + Quartus II Version + OS

That's the bulk of it, although I'm surely missing something(RAM initialization files come to mind). Anyway, the actual seed setting is only one line in the .qsf, where a change in any of those inputs is essentially a new seed. When thought of that way, there is no magic seed because you're never going to run the same seed twice, as something is always changing in the design. The only case I can think of is if the results from a particular compile were lost and it was re-compiled, in which case it should return the exact same results. (Yes, there is NOTHING random in Quartus II, so the same input will give the same output every time. Incremental Compilation and Rapid Recompile can cause deviations from this, but for deterministic reasons.)

Design Space Explorer for Seed Sweeping

Under Quartus II Tools, one can access the Design Space Explorer(DSE), which will close Quartus II and is a tool for running multiple compiles. To run a seed sweep, the user only needs to change the effort level to Low and select the seeds they want to run:



The above settings will run seeds 2,3,4 and 8. (There is no reason to skip seeds like this, since they are all random. I did it purely to show the syntax of entering a range as well as specific numbers. I also could have put "2 3 4 8")

To run as a script, the user can do the following:

quartus_sh --dse -nogui -seeds 1-4 -skip-base -project top -revision top_rev
cd dse
quartus_sh --restore -output best_restored best.qar

In this example, the project file is top.qsf and the settings file is top_rev.qsf. Often the .qsf name matches the project name and hence the –revision option is unnecessary. I also added the option to –skip-base, where it won't re-run the existing seed if it was already compiled. I also added two lines

to unqar the file best.qar that gets created and contains the best results. To get more information on the options, run the following:

quartus_sh --help=dse

Besides creating a best.qar, DSE will create a <project_name>.dse.rpt that shows the steps it took, and a results.csv file that contains the slack of all four analyses (setup, hold, recovery and removal) for every clock domain in the design, across all three timing models(it's quite a bit of information). More information about the Design Space Explorer can be found in Quartus II help. (Another nice feature is that other machines can be put into DSE slave mode, whereby the seed sweep can be run across multiple machines rather than serial compiles).

When to Sweep Your Design

Design Profile

So why do a seed sweep? Probably the first reason is to get a sense of the design's variance. For example, two designs both might have a 5ns requirement on the main clock, and both might meet timing when compiled, but after running 5 seeds, one might find the following slacks for the designs:

Design 1 Slack: 0.217 0.344 -0.043 0.275 -0.090

Design 2 Slack: 0.322 0.375 0.281 0.403 0.224

Design 1 has an average of slack of 0.140ns and a standard deviation of 0.195ns. Design 2 has an average slack of 0.321ns and a standard deviation of 0.071ns. Working on Design 1, the user should expect that compiles will fail timing at a relatively common rate, and if it fails timing that does not mean anything changed for the worst. Working on Design 2, if the design fails timing, the user should probably make note of the changes they made and see if that compile was just an extreme outlier or if they did something that adversely affected timing. What to do with that information depends on the situation. Knowing your designs average result and variation from there is valuable information.

That being said, most users have a pretty good idea of their design variation just by looking at the timing results after each compile. Designer 1 probably already knows that their design occasionally fails, while Designer 2 knows they've always made timing. Just the normal process of making changes and recompiling the design is very similar to running a seed sweep, and as long as the user monitors their results, their often isn't any need to explicitly run a seed sweep.

Monitoring the Affect of a Change

Any design change can make a design faster, slower or stay the same. Likewise, a design change will essentially be like running another seed, and hence there will be variation that causes that particular compile to be faser, slower, or the same. Running a seed sweep allows the user to tease out the variation, since it is essentially random, and determine what impact their design change has on the results.

For example, if I'm putting a max_fanout assignment on some registers to try and get better performance (see http://www.alterawiki.com/wiki/Register_Duplication_for_Timing_Closure), then I would be curious if my changes really helped, since it could be masked by design variation. Let's say I do this on Design 1 above, which has a standard deviation of 71ps. So maybe the compile before the assignment had 275ps slack (much better than average), but after making the assignment, the result is 180ps slack. The design got worse. But that may just be due to variation, and running multiple seeds may show that the average is 190ps, which is 50ps better than before.

Duplicating registers is something that has almost no downside. It runs quickly, only adds a few registers, and should not ever hurt performance. In general I might add a max_fanout assignment and as long as the results didn't get considerably worse(as they shouldn't), would not worry about running a seed sweep. But a design change that does have a potentially negative impact might make more sense. For example, turning on Physical Synthesis will increase compile times. Running a seed sweep might show that it gains you ~60ps of slack, but increases the compile time by ~1 hour. That information is useful in that the designer can decide if and when they want to turn that option on. Another example might be a design change that should improve performance, but also makes the design larger. If it's a noticeable improvement, there is no need to run a seed sweep, but if the improvement seems marginal or non-existent, running an extra seed or two might provide more insight as to whether the design change is worth the area trade-off.

If a user runs a seed sweep after every modification, they will surely be wasting time. Instead they should use seed sweeps as a tool, to be pulled out only when the information is worth the time it takes to run the compiles and analyze the results.

Performance Gain

This is the number one reason I see users running seed sweeps, in that their design does not meet timing without running multiple seeds. There are times when this is absolutely necessary, such as the final run of a design before release, but quite often I see designers rely on seed sweeping in order to close timing early in the design cycle. If this is the only way it closes timing, then of course it makes sense, but I strongly recommend the user concentrate on optimizing their design for better performance in parallel.

Designs always grow, and slow down a little bit as more logic gets packed in. Even if it's less than a 1% drop in performance, that might be enough to make a design that met timing every other compile to suddenly make it only one out of ten. More logic is then added, new critical paths are created, and suddenly the designer is running thirty seeds hoping one of them will pass. I have had several users complain about compile times, only to find it has nothing to do with their individual compile times, but the number of seeds their running. This is usually an untenable situation.

Users often assume that if a design meets timing once, the fitter should be able to do it every time, and if it can't, that's a problem with the fitter. Hopefully this document helps explain that variation in results is generally unavoidable. My opinion is that seed sweeping should be one of the last tools for improving performance, after the user has exhausted all other methods. That does not mean

they shouldn't be doing seed sweeps to meet timing along the way, it just means that if the only way their design meets timing is with a seed sweep, they should be sure to put effort into other methods for timing closure.

Final Note: This document is mainly informational about how the fitter works, why there is variation in results, and how seed sweeping can be done to examine that variation. I plan on writing a follow-on document that discusses Incremental Compilation and LogicLock(floorplanning), as well as other tactics for improving performance. I will be sure to update this wiki site with a link to that document when it is complete.