FPGA Design Performance Optimization for Complex Designs

| Whitepaper

InTime Timing Closure Methodology for Quartus

Kirvy Teo / Harnhua Ng

PLUNIFY

FPGA Design Performance Optimization for Complex Designs

InTime Methodology for Quartus

Kirvy Teo / Harnhua Ng

Contents

Introduction

With advancements in FPGA architectures and technology, FPGA designs are getting more and more complex. The growing prevalence of high-speed interfaces, mixed-signal blocks and usage of 3rd-party Intellectual Property (IP) blocks are some of the factors that have exponentially increased the difficulties of FPGA timing closure. In response to these new challenges, the latest FPGA tools have evolved accordingly, possessing more advanced methodologies and better synthesis and place-and-route algorithms to handle modern designs. In particular, these improvements have led to new build parameters with noticeable impact on design performance. The other inevitable consequence is an increase in overall build time and in compute resource demands. With newer and larger device families, build times and tool memory requirements may skyrocket, leading to longer turnaround times and reduced productivity.

To deal with these challenges, the InTime Design Optimizer tool is based on best practices and guidelines to determine the best build parameters, with the condition that the design is currently immutable, i.e. you cannot change your RTL or constraints. InTime uses machine learning principles to achieve timing closure and optimization, treating the FPGA synthesis and place-and-route tools as black boxes and analyzing design performance across a whole range of build parameter variations.

Under the InTime Optimization Methodology, an effective build process is no longer a one-designer-to-one-machine operation. Instead it is a systematic series of calculated steps done by one or many designers on multiple build machines. From the resulting analysis, InTime deduces and recommends good build parameters aimed at improving design performance. The guidelines in this document will help you achieve your performance goals in the minimum number of builds and fastest turnaround time possible.

Commonly-used terms

- **Total Negative Slack (TNS)**: Sum of the negative slack in your design. If 0, then the design meets timing.
- **Worst Negative Slack (WS or WNS):** The most severe amount by which timing fails in your design. If positive, then there are no timing failures.

Understanding the InTime Optimization Phases

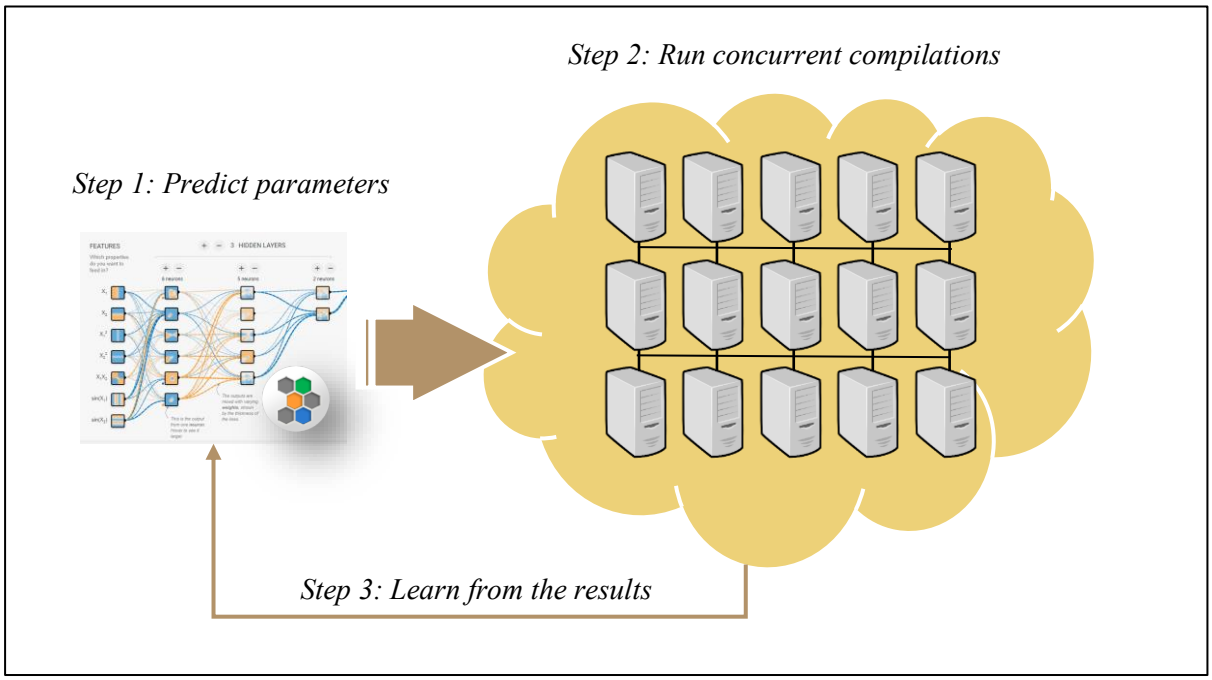
InTime is a software plugin that runs FPGA tools such as Intel Quartus Prime ("Quartus") in the background. The FPGA tools offers many build parameters that affect an FPGA design on a global as well as local block level. InTime's goal is to determine the optimum build parameters for the design. To deal with an enormous design space consisting of countless combinations of parameters, InTime uses a machine learning approach combined with domain-specific heuristics to predict and narrow down the best parameters. Machine learning is a means to converge more quickly on the optimal parameters compared to doing random variations.

To maximize timing closure effectiveness, it is necessary to generate sufficient data points from build results and learn from past results. There are two phases to the optimization process: Phase 1 is the "Learning Lifecycle" and Phase 2, "Last-Mile Optimization".

Phase 1: Learning Lifecycle

In this phase, the recommended InTime Optimization Methodology is to progressively optimize a design over several rounds of synthesis and place-&-route builds in an iterative "build-and-learn" lifecycle.

Figure 1: Learning Lifecycle

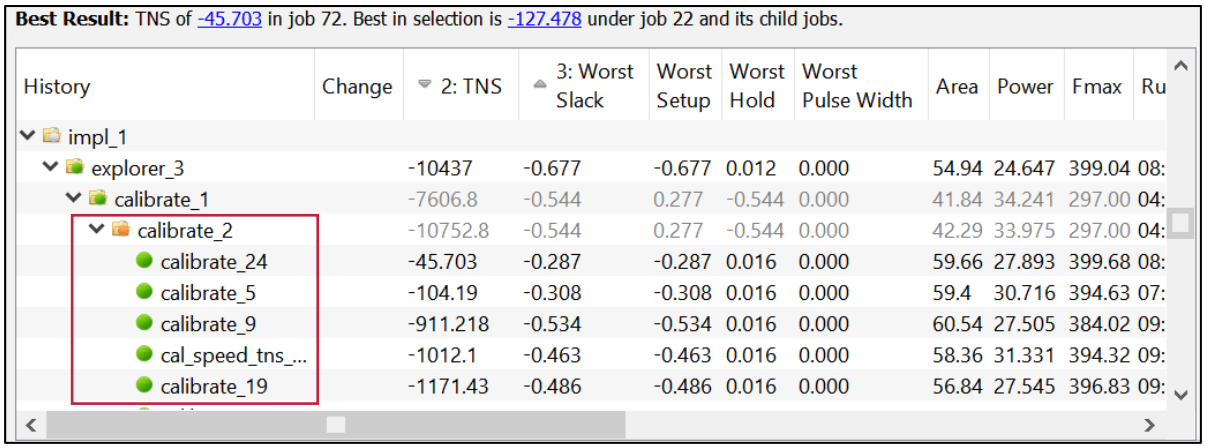


1. Use the current InTime database and train a machine learning model to predict combinations of build parameters.

2. Leverage on compute power to run multiple builds concurrently.
3. If one or more of the builds meet timing, optimization stops and your goal has been achieved.
4. Otherwise, InTime applies machine learning to learn from the build results and uses the updated data model for subsequent predictions. Repeat steps 1 to 4 if necessary.
5. If there is at least one "good" result ("good" is defined in the "Moving to the Next Phase section"), proceed to Phase 2.

The build optimization rounds are represented and displayed in InTime's history window (see Figure 2). The entire history of compilations and their sequences is stored and saved in a hierarchical tree structure.

Figure 2: History Window of InTime



Each round is called a "job" and each combination of parameters is called a "strategy". In the figure above, "calibrate_24" or "calibrate_19" are names of strategies, and a job, denoted by the red rectangle, can consist of one or more strategies. Each job has a "parent" result on which the strategies are based, and each new job adds a hierarchical level that branches out from its parent.

Phase 2: Last-Mile Optimization

The second phase begins when at least one of the results is close to meeting the performance target or if results have stopped improving in Phase 1. In the former case, the optimization relies on specific techniques that stimulate minor (compared to those in Phase 1) variations in the results. This phase consists of:

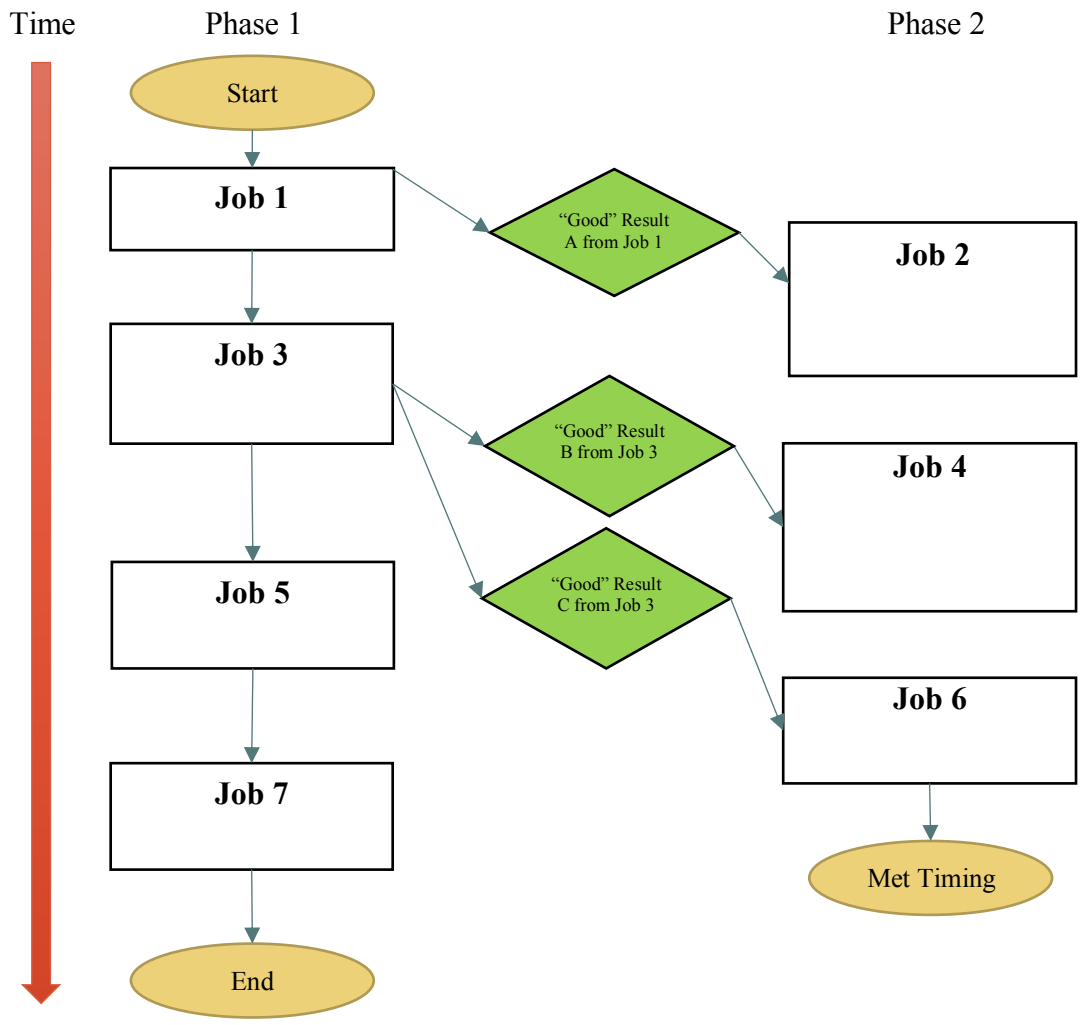
1. **Random**
   Running placement seed exploration, effort levels and placement adjustments.

The quality of results in Last-Mile Optimization is a function of compute power. Having more compute power ensures that the builds finish faster. However, the key to obtaining good Phase 2 results is that you must first attain a sufficiently good result in Phase 1.

Combining Phase 1 & Phase 2

Since each round of the Learning Lifecycle (Phase 1) can produce multiple results which are worthy of Last-Mile Optimization (Phase 2), you can actually run both Phases together in parallel. It is not necessary to wait for either Phase to complete before starting the other (See Figure 3).

Figure 3: Using Phase 1 in parallel with Phase 2



InTime Optimization Process

InTime Recipes

Within each phase, there are multiple “recipes”. A “recipe” represents an algorithm to select build parameters, either through machine learning, randomly or via other methods. Each recipe generates one or more strategies. Different recipes are used under different conditions. InTime comes with four recipe categories:

1. Learning
2. Last-Mile
3. General
4. Advanced

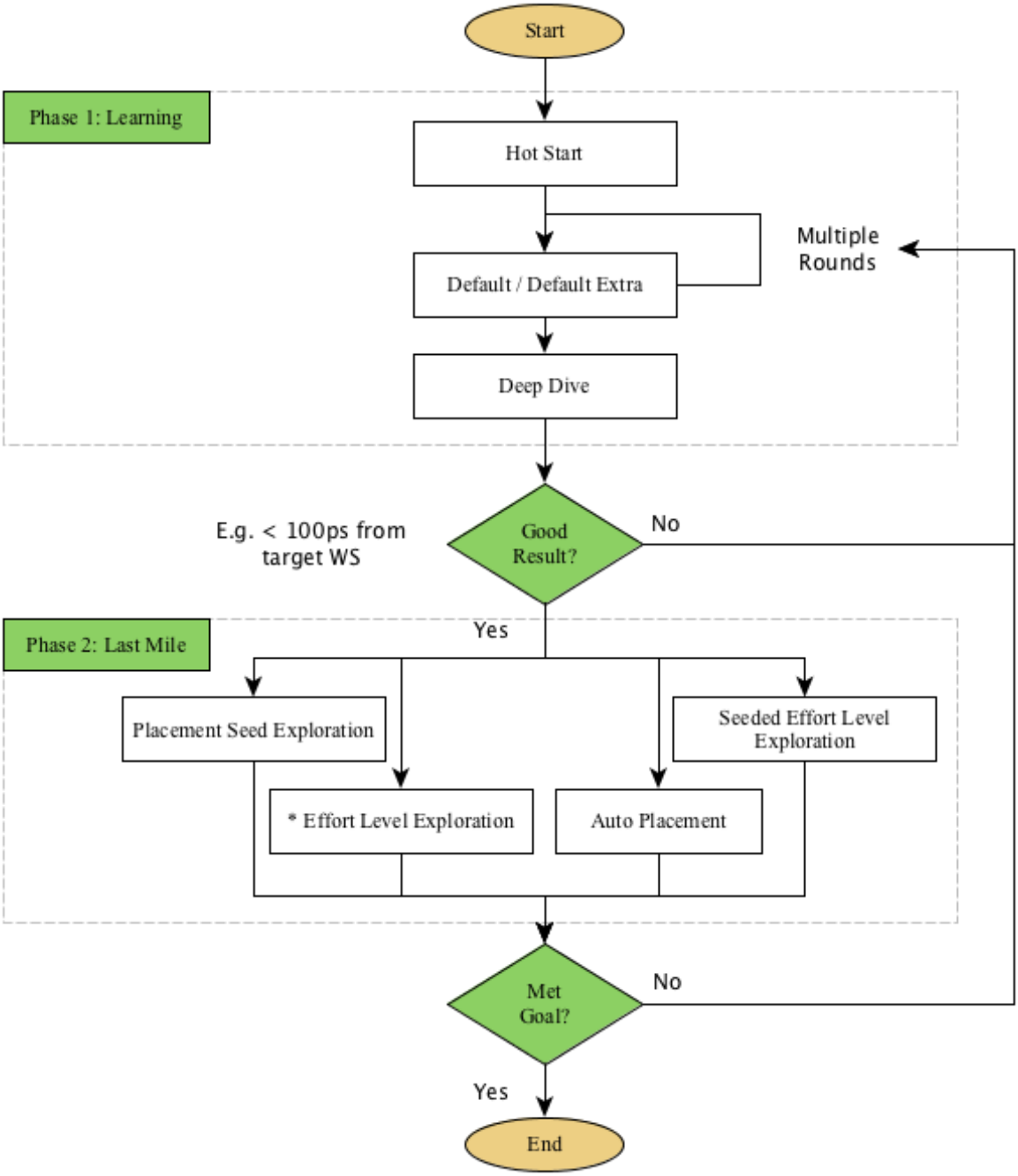
Firstly, use Learning recipes during Phase 1 to automatically select build parameters in light of all the build result data accumulated in previous rounds. Employ Last-Mile recipes when your design is close to meeting your timing target. General recipes include a basic script to build your design as is, plus a way to re-build existing strategies. Finally, Advanced recipes give you the flexibility to run customized strategies that can be generated outside of InTime.

A complete list of all the recipes can be found in Appendix A.

Recipe Selection

For a design new to InTime, our recommended approach is to assume the absence of any data points and start with the “Hot Start” recipe, a Learning Lifecycle recipe. The figure below shows the typical optimization process of InTime for Quartus.

Figure 4: InTime Optimization Flow for Quartus Designs



Hot Start serves multiple purposes and consists of known-good strategies as well as strategies distilled from your InTime database. Its first objective is to explore different parameters that worked well for known issues, for example congestion and high utilization in different FPGAs and designs. The other purpose is to generate initial data for subsequent machine learning rounds.

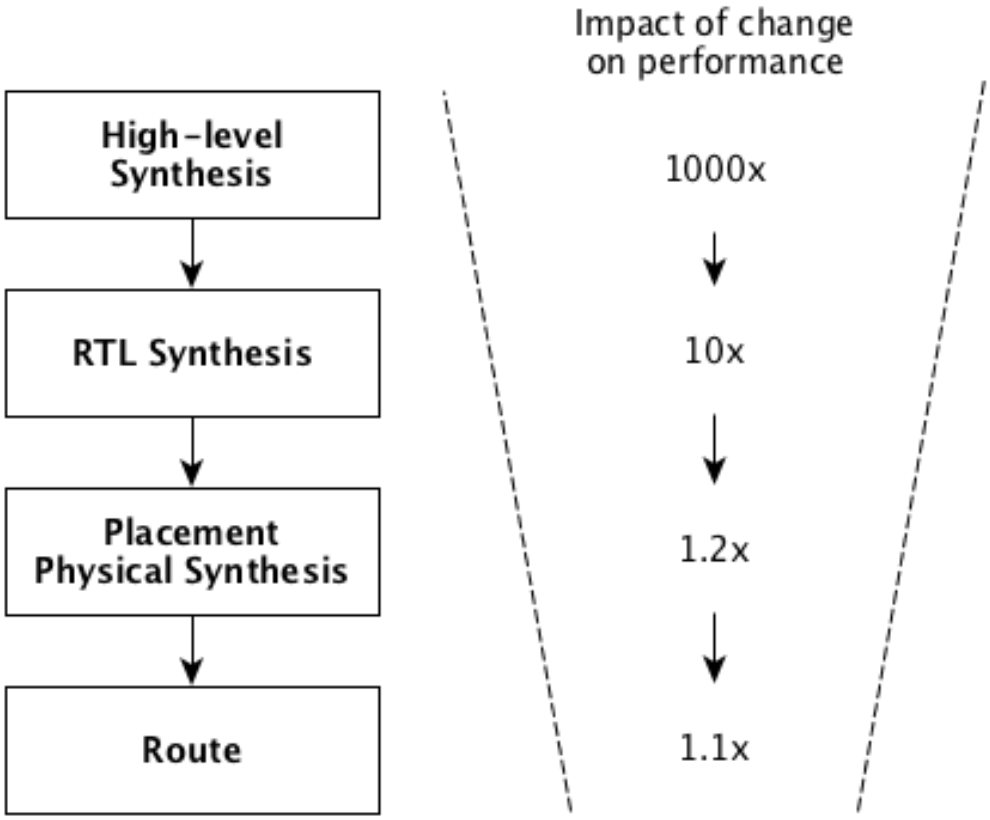
Hot Start can also be used as a test to gauge whether or not timing closure is possible. Refer to the section "Possibility of Meeting Timing"

Parameter Selection

Parameter effects at Different Build Stages

The impact to results at different build stages can vary by orders of magnitude. Refer to the following figure.

Figure 5: Optimization Funnel



Using good synthesis parameters will generally have a larger impact on the overall design performance than varying routing parameters. This is especially important if your design has a high Worst Slack.

Build Parameter Inter-dependencies

The FPGA tool user guides describe in varying levels of detail what each build parameter does and what aspect of performance it targets. By itself, a build parameter is usually easy to understand in terms of why, when and how to use it. However, it is vital to understand that

build parameters do not operate in isolation and have complex, inter-dependent relationships with other build parameters.

For example, synthesis parameter A may affect placement parameter B, or placement parameter B may override the effects of routing parameter C. With between 30 to 80 different build parameters depending on the FPGA tool, it is difficult to fully comprehend whether a single parameter in a particular group of parameters is good or bad. Using machine learning, InTime establishes a consistent and disciplined approach to deciding what is good or bad.

The InTime Default Recipe

The InTime Default ("Default") recipe is the follow-up to Hot Start. In the absence of Hot Start, you can also use Default as your first recipe. This recipe is highly flexible.

Multiple Rounds and Runs

The Default recipe requires multiple rounds and will be compute-intensive. Ideally, you should use enough machines to complete them as quickly as possible. The default number of rounds is three (3) and the number of runs (strategies) per round is thirty (30). In many cases, more rounds are necessary due to the need for more data points. The recommended total number of builds is at least 100, regardless of rounds and runs. The figure below shows eight rounds of Default. Each round corresponds to a "job". The green line represents the best TNS result for a particular round/job, and the red line shows the worst TNS for that round/job.

Figure 6: TNS improvements based on Default Recipe

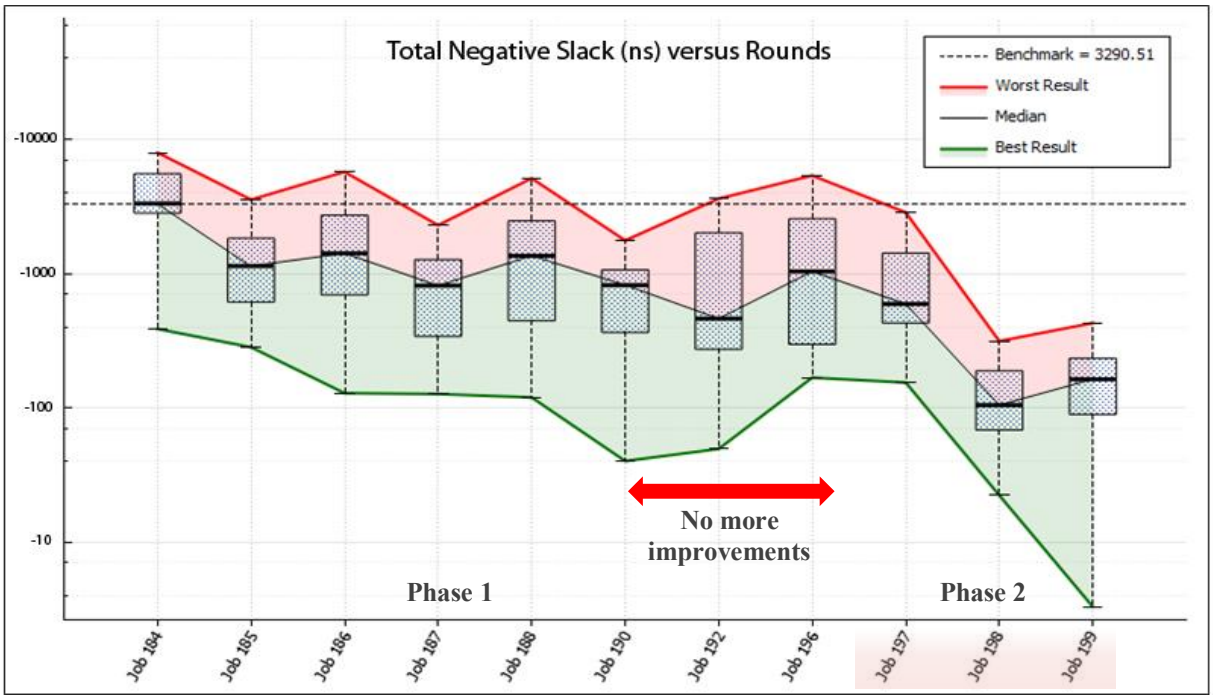


Figure 6 shows how the best result of a round can fluctuate over multiple rounds, as opposed to having a consistent downwards trend. This is expected in design space exploration.

Longer Runtimes and Aggressive Build Parameter Selections

Compared to Hot Start, Default has a wider range of parameter selections. This is necessary to trigger a broader variation of results to learn from. If the design's utilization is very high, strategies are more likely to over-fit due to Default's usage of more aggressive parameters. The other consequence of employing aggressive parameters is an increase in build runtime. Because overly-long runtimes tend not to result in improved timing, InTime recommends a maximum runtime of 2x the original build time. Maximum runtime is an InTime tool property that the user can specify to have builds automatically terminated beyond a certain elapsed time. Refer to our user guide "Set Flow Properties" for more information.

Goal-Based Build Parameter Selection

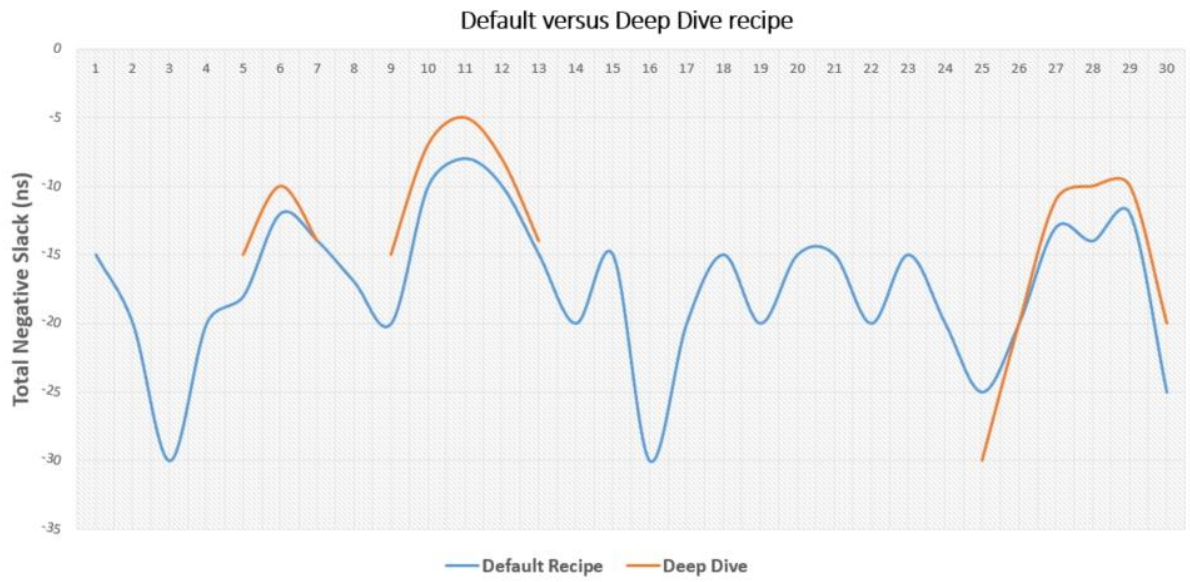
Default selects parameters based on your target goal, which can be "Speed-TNS", "Area" or "Power" in InTime's Flow Properties section. Note that InTime does not optimize by targeting Worst Slack/WNS, because the critical path responsible for WNS is usually different in various strategies and is generally not an apples-to-apples comparison between results. Better WNS is achieved as an effect of optimizing for better TNS.

The Deep Dive Recipe

Compared to Default, the Deep Dive recipe ("Deep Dive") has a much less aggressive build parameter selection and is typically employed after running Default. It focuses on using existing good results and varying about 10-20% of those parameters. Deep Dive is also helpful when you are still not meeting timing after Phase 2 and need to run more rounds in Phase 1.

For example, Figure 7 shows shows that by using a cut-off of -50ns Total Negative Slack as the threshold for "good" results, the percentage of good results for "deep dive" is higher - around 8x better. Deep Dive looks only at the better results and attempts to find other results that are within range of the local maxima.

Figure 7: Differences in effects for Default vs. Deep Dive



Use Deep Dive when the number of good results is less than 5% compared to the total number of results. Another situation is when there are good outliers, i.e. one or two results are extremely good, this recipe will focus on those outliers and explore the optimization space close to them.

Moving to the Next Phase

The objective of the Default and Deep Dive recipes is to meet timing or get as close to timing closure as possible in Phase 1. Repeat this Phase until one of the following conditions is met:

1. The design has met timing, in which case the optimization stops.
2. The design does not show any improvement in the first 50 compilations.
   In that event, usually the design or optimization requires very specific build

parameters. Another option is to review location constraints, for example, IP block constraints or specific placement constraints. Releasing such constraints may give the tools more freedom to explore and optimize the final timing performance.

In both the above two conditions, optimization ends.

However, if the situations below apply to you, go directly to Phase 2: Last-Mile Optimization.

1. Improvements in Worst Slack or in TNS have plateaued after three or more rounds. This behavior happens usually starting from the 4th round onwards or after 100 compilations.

2. There are "good" results compared to the original timing results. Good results can be defined as
    a. a result within 300ps of meeting your timing target or timing closure
    b. a Worst Slack or TNS that is 80% better than the original value.

Last-Mile Recipes

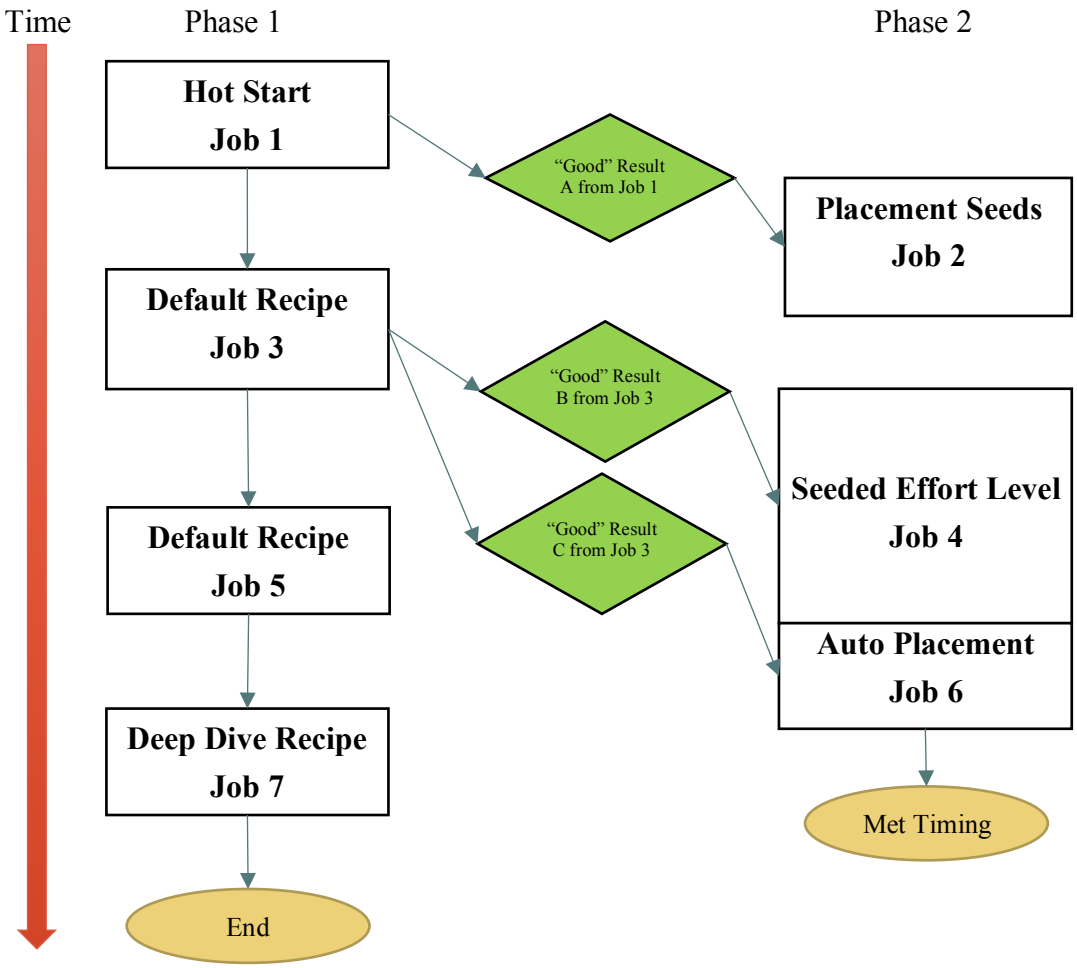
Some Last-Mile recipes, employed in Phase 2, stimulate and create randomness in FPGA placements. Although random in nature, such an approach has proved to produce a consistent, albeit limited range of improvements. Use Last-Mile recipes when you are close to meeting the timing target.

Selecting an Appropriate Parent Result

In Phase 2, first select an existing result from Phase 1 as the "parent". Optimizations in this phase are based on the parent's parameters. This is usually one of the better results, for instance, the one with the best Worst Slack or the best TNS, or the lowest utilization.

As mentioned previously, any round of Phase 1 can produce one or more results that can be run in Phase 2, so you can actually execute both Phases concurrently. It is not required that you wait for either Phase to complete before starting the other (see Figure 8). The only limitations are probably the availability of compute resources and/or FPGA tool licenses.

Figure 8: Concurrent Phases and Jobs



Available Last-Mile Recipes

Last-Mile recipes are constructed based on the build parameters and flows provided by the Quartus tools.

	Tool(s)	What it does
Auto Placement	Quartus II, Quartus Prime	A single-iteration flow that analyzes the worst critical paths in your design, and adjusts the locations of certain path elements in order to improve timing. The extent of improvement is dependent on how much the Fitter thinks that the locations have changed. For more information, please refer to this article.
Effort Level Exploration	Quartus II, Quartus Prime	There are tool options to make the Fitter work harder during placement to find better solutions. One of these options is called, "Effort Level". This recipe exercises different placement effort levels to find improved results for your design. Runtime and degree of improvements vary according to the effort level used. Higher effort levels do not necessarily yield better results.
Placement Seed Exploration	Quartus II. Quartus Prime	This recipe provides the means to perform a Seed Sweep, an operation that changes the initial placement of a build, thereby affecting the results by around +/- 5%.
Router Effort Exploration	Quartus II, Quartus Prime	Similar to the Placement Effort Exploration Recipe above, but applies to routing operations instead. This recipe supports certain device families only.
Seeded Effort Level Exploration	Quartus II, Quartus Prime	This recipe combines the best aspects of the Placement Effort Level and Placement Seed Exploration recipes in that it first executes a round of Effort Levels, followed by Seed

	Sweeps on the most improved results. Compared to running Effort Level or Placement Seeds Exploration separately, this recipe automates the transition between the two.

Faster Convergence

The InTime Timing Closure Methodology is more efficient when more compute resources are made available. Although compute resources have been commoditized by cloud computing, many organizations are still restricted to using internal servers due to security concerns. The following techniques help your results to converge faster even if you have a limited number of build machines; first by reducing the required run time per round and subsequently, the number of rounds required to converge onto optimized results.

Likelihood of Meeting Timing

For Quartus designs, the Hot Start recipe can also be used as a litmus test of whether the timing closure or optimization goal can be achieved. Our guideline is that if the best Worst Slack achieved by Hot Start is around -0.5ns, then it is likely that InTime can achieve half of that (-0.25ns) and better with subsequent recipes. Of course, the underlying assumption of this document is that the design to be optimized is properly constrained and created in accordance with the FPGA vendor's recommended design methodology. Extreme circumstances like logic utilization of more than 95% will also make timing closure more challenging.

Hot Start also serves as an assessment of a design's "health". For example, if the post-route Worst Slack results are all worse than -0.8ns, it is less likely that timing closure can be achieved – design changes are probably necessary at that point.

Re-run Previous Strategies

Design reuse plays a significant role in the InTime flow. When a design is modified (for example, feature additions or bug fixes), we recommend re-using known-good strategies from previous iterations as they are likely to yield good results. The "Re-run Strategies" recipe in the General category allows you to select strategies to run again (see Figure 9).

Figure 9: Re-run best strategies

History	Change	2: TNS	Worst Slack	Worst Setup	Worst Hold	Worst Pulse Width	Area	Power
impl_1								
impl_1								
calibrate_17								
calibrate_4		-1371.39	-0.415	-0.415	0.016	0.000	52.7	27.125
calibrate_1		-9312.81	-0.544	0.277	-0.544	0.000	38.61	29.914
calibrate_8		-4173.63	-0.955	-0.955	0.016	0.000	49.13	26.612
calibrate_15		-7353.45	-0.839	-0.839	0.016	0.000	56.9	28.66
cal_speed_tns_low_1		-398.411	-0.502	-0.502	0.016	0.000	57.46	26.857
calibrate_18		-214.834	-0.431	-0.431	0.016	0.000	54.42	26.681
calibrate_21		-929.695	-0.379	-0.379	0.016	0.000	53.59	26.578

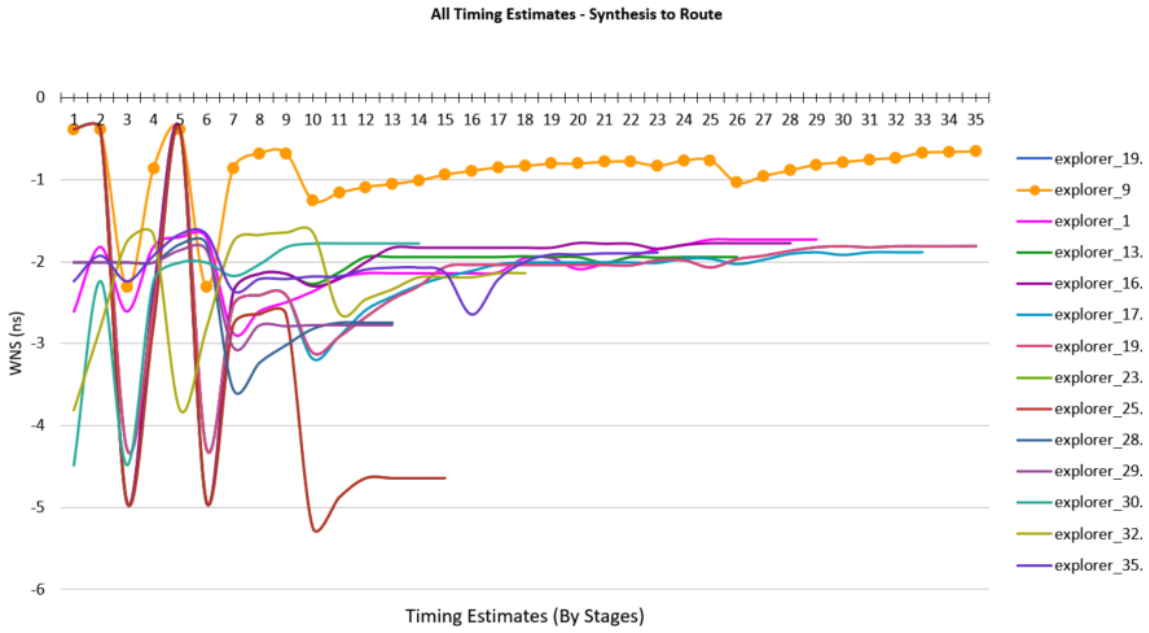
This approach also works very well when you move to a newer version of your FPGA tool. With a newer version, design performance results may change even with default build parameters. Instead of running InTime Optimization Phases 1 and 2 rounds from scratch, this recipe takes advantage of possible correlations with the previous version of the synthesis and place-and-route tool to reduce the number of builds needed to get to timing closure.

Minimize Run Time with Timing Estimates

Quartus timing estimates obtained at each intermediate build step provide a preview of the final results, with the caveat that the accuracy of these estimates improves as the build process approaches the routing step. This implies that you would have to wait longer to get a clearer picture of your final timing performance.

The InTime Timing Closure Methodology recommends using post-placement timing as a key indicator. Right after placement, the tradeoff between runtime and timing estimate accuracy seem to be the best. Routing is often the most time-consuming stage, whereas when placement completes, the build is only about halfway done. The chart below tracks Worst Slack estimates across different build stages for a design with about 70% logic utilization. Although most of these intermediate timing estimates improve as the FPGA tool optimizes the design, if the post-placement timing estimate is poor, the post-route result tends to be bad as well. Instead of hoping for a black swan-type post-route result, it is generally better to make a decision to continue or give up at the post-placement stage.

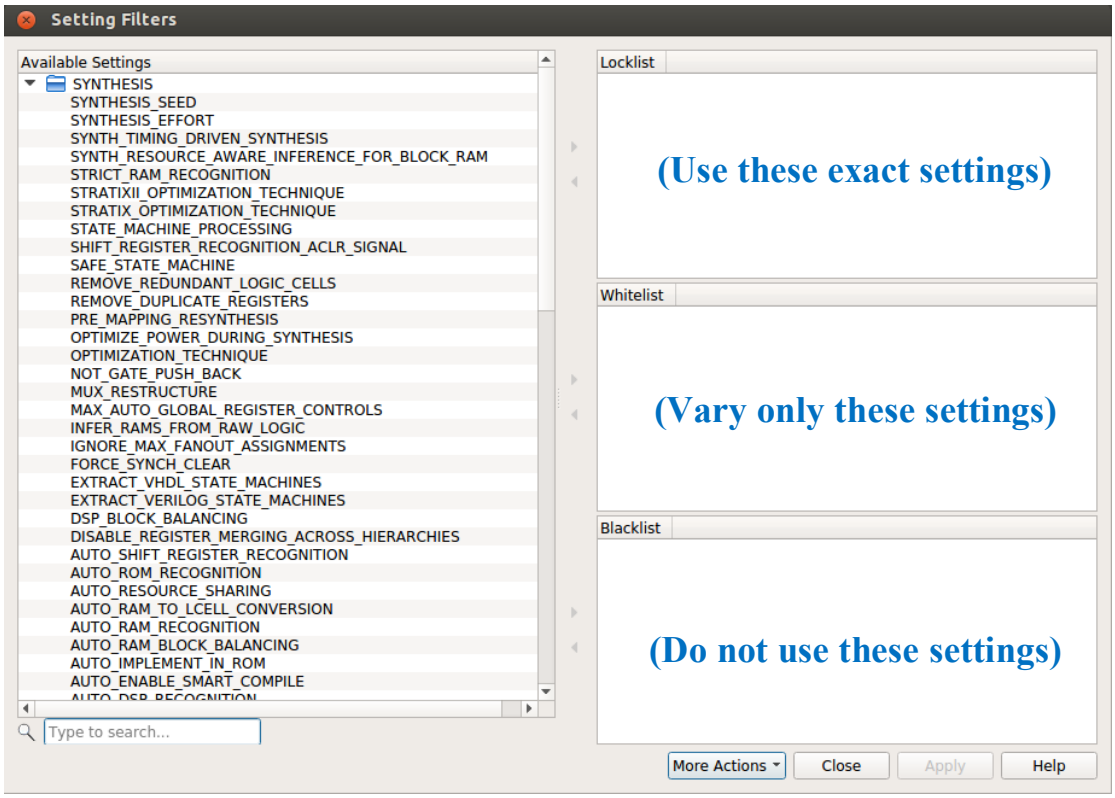
Figure 10: Intermediate Timing Estimates



Use Setting Filters for Specific Design Issues

Settings Filters allows the user to specify the degree of exploration for one or more build parameters. As the designer knows the design best, they may want to specify certain build parameters that have been known to perform better (or worse) for the design. Therefore, it may be more productive to prime (or limit) the optimizations by specifying upfront parameters that should be included or avoided.

Figure 11: Settings Filters



For example, in a congested design, finding the best synthesis parameters before exploring implementation ones is often more efficient. In such a situation, running the Default recipe with synthesis parameters in the "Whitelist" section of the Settings Filters ensures that synthesis is explored first.

Subsequently, if you take the best synthesis parameters and place them in the Locklist (making sure to clear the Whitelist as well), running the Default recipe again with those good synthesis parameters locked-down ensures that they are used in all builds.

More details about this feature can be found in the InTime documentation - "How to use project setting filters".

Conclusion

Timing Closure is a critical step in your design flow. Planning for timing closure is essential for pre-empting potential issues. Design complexity has increased tremendously and it is necessary to look beyond traditional approaches to solve these problems. The InTime Timing Closure Methodology outlines a systematic way of achieving consistent performance improvements. Assisted by technologies like cloud computing, most timing issues can be resolved without overhauling the entire RTL or sacrificing critical functionality.

Appendix A: InTime Recipes

	Recipe	Quartus-II/ Prime Std	Quartus Prime Pro	Description
Learning	Hot Start	Y	Y	Generates initial strategies for your design to evaluate how much it can be optimized.
	InTime Default	Y	Y	Performs first time calibration, exploration and optimization of your design.
	InTime Default Extra	Y	Y	Runs InTime Default and picks the best result to perform additional optimizations.
	Deep Dive	Y	Y	Performs deeper analysis on existing results, especially on differences between good and bad ones.
Last-Mile	Auto Placement	Y	Y	Adjusts the physical locations of certain critical path elements in order to improve timing.
	Effort Level Exploration	Y	Y	Tries different placement effort settings, trading against runtime, to improve performance.
	Placement Seed Exploration	Y	Y	Performs different placement initilizations on the design.
	Router Effort Exploration	Y	Y	Similar to Effort Level Exploration but uses Routing Effort settings instead.
	Seeded Effort Level Exploration	Y	Y	Combines Effort Level and Placement Seed Exploration by first running the former and then pick the best results to run the latter on.
General	Just Compile My Design	Y	Y	Compiles the active revision in your project.
	Rerun Strategies	Y	Y	Rerun strategies selected by the user.
Advanced	Custom Flow	Y	Y	Uses strategies specified by the user.

Document Revision History

	Date	Changes Made
1	08 June 2018	Initial Version