

InTime Timing Closure Methodology for Vivado

Kirvy Teo / Ng Harnhua

Contents

Introduction.....	3
Understanding the InTime Optimization Phases	4
Phase 1: Learning Lifecycle.....	4
Phase 2: Last-Mile Optimization	5
Combining Phase 1 & Phase 2.....	6
InTime Optimization Process	7
InTime Recipes	7
Recipe Selection.....	7
Parameter Selection	9
Parameter effects at Different Build Stages.....	9
Build parameter inter-dependencies	10
The InTime Default Recipe	10
Multiple Rounds and Runs.....	10
Longer Runtimes and Aggressive Build Parameter Selections	11
Goal-Based Build Parameter Selection.....	11
The Deep Dive Recipe	12
Moving to the Next Phase.....	12
Last-Mile Recipes	14
Selecting an Appropriate Parent Result	14
Available Last-Mile Recipes.....	15
Faster Convergence.....	17

Likelihood of Meeting Timing..... 17

Re-run Previous Strategies..... 17

Minimize Run Time with Timing Estimates 18

Use Setting Filters for Specific Design Issues 19

Conclusion20

Appendix A: InTime Recipes21

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 which have 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, adversely leading to longer turnaround times and reduced productivity.

To deal with these challenges, the InTime Timing Closure Methodology is a set of best practices and guidelines to determine the best build parameters under 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 or 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 Timing Closure Methodology, the 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 sets of 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 compilations 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.

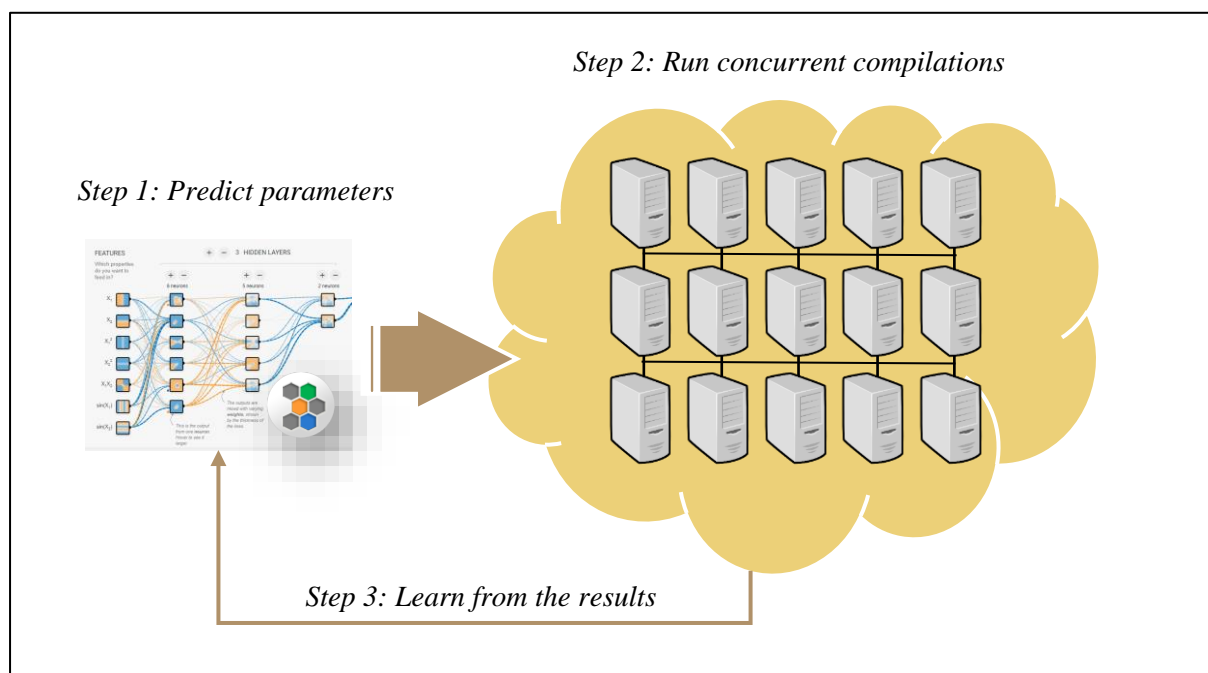
Understanding the InTime Optimization Phases

InTime is a software plugin that runs FPGA tools such as Xilinx Vivado (“Vivado”) 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 machine learning techniques and domain-specific heuristics to predict and narrow down the best parameters. 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 Methodology is to progressively optimize a design over several rounds of synthesis and place-and-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

Best Result: TNS of [-45.703](#) in job 72. Best in selection is [-127.478](#) under job 22 and its child jobs.

History	Change	▼ 2: TNS	▲ 3: Worst Slack	Worst Setup	Worst Hold	Worst Pulse Width	Area	Power	Fmax	Ru
▼ impl_1										
▼ explorer_3	-10437	-0.677	-0.677	0.012	0.000		54.94	24.647	399.04	08:
▼ calibrate_1	-7606.8	-0.544	0.277	-0.544	0.000		41.84	34.241	297.00	04:
▼ calibrate_2	-10752.8	-0.544	0.277	-0.544	0.000		42.29	33.975	297.00	04:
● calibrate_24	-45.703	-0.287	-0.287	0.016	0.000		59.66	27.893	399.68	08:
● calibrate_5	-104.19	-0.308	-0.308	0.016	0.000		59.4	30.716	394.63	07:
● calibrate_9	-911.218	-0.534	-0.534	0.016	0.000		60.54	27.505	384.02	09:
● cal_speed_tns_...	-1012.1	-0.463	-0.463	0.016	0.000		58.36	31.331	394.32	09:
● calibrate_19	-1171.43	-0.486	-0.486	0.016	0.000		56.84	27.545	396.83	09:

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. There are two types of optimizations in this phase:

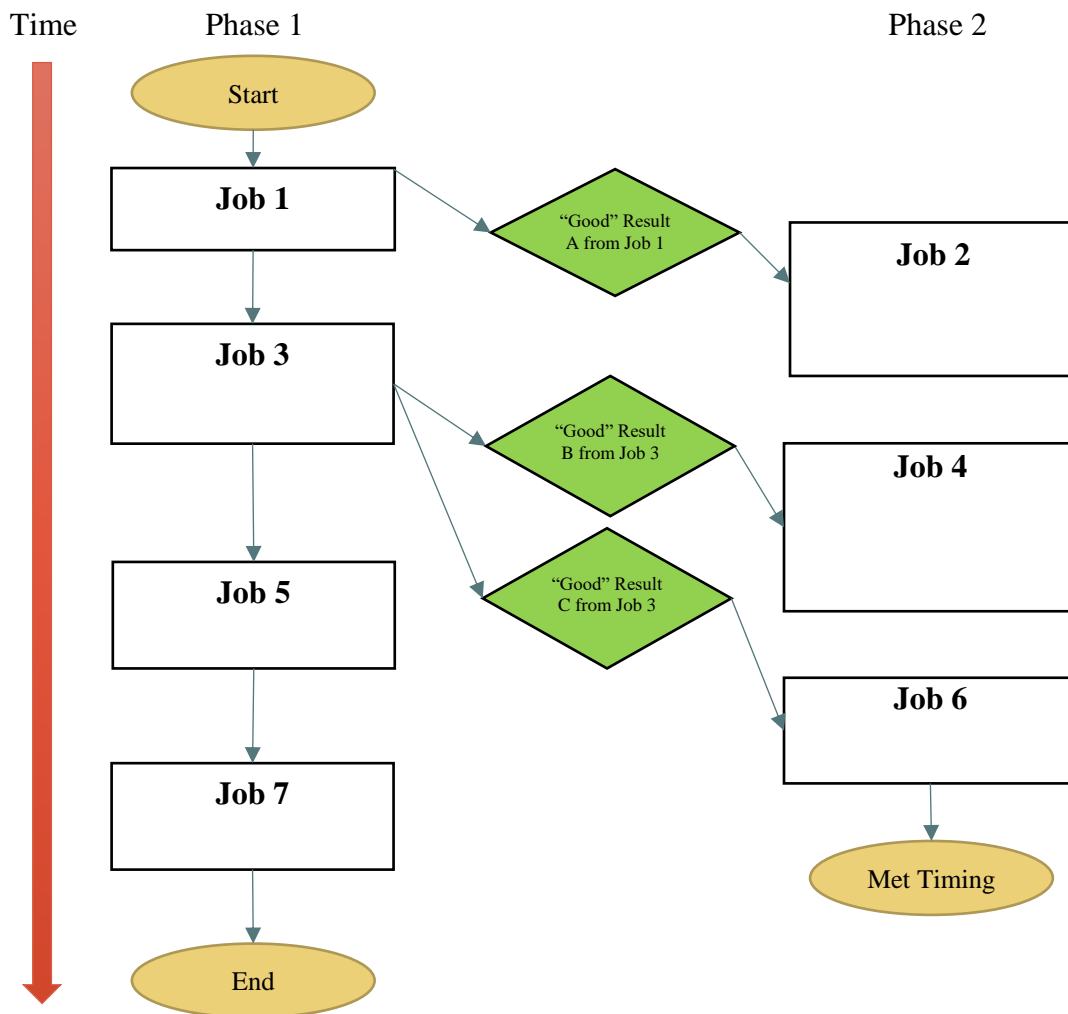
1. **Random**
Running placement exploration, effort levels and exploiting clock uncertainties.
2. **Incremental**
Using a particular build to run post-route physical optimization iteratively.

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

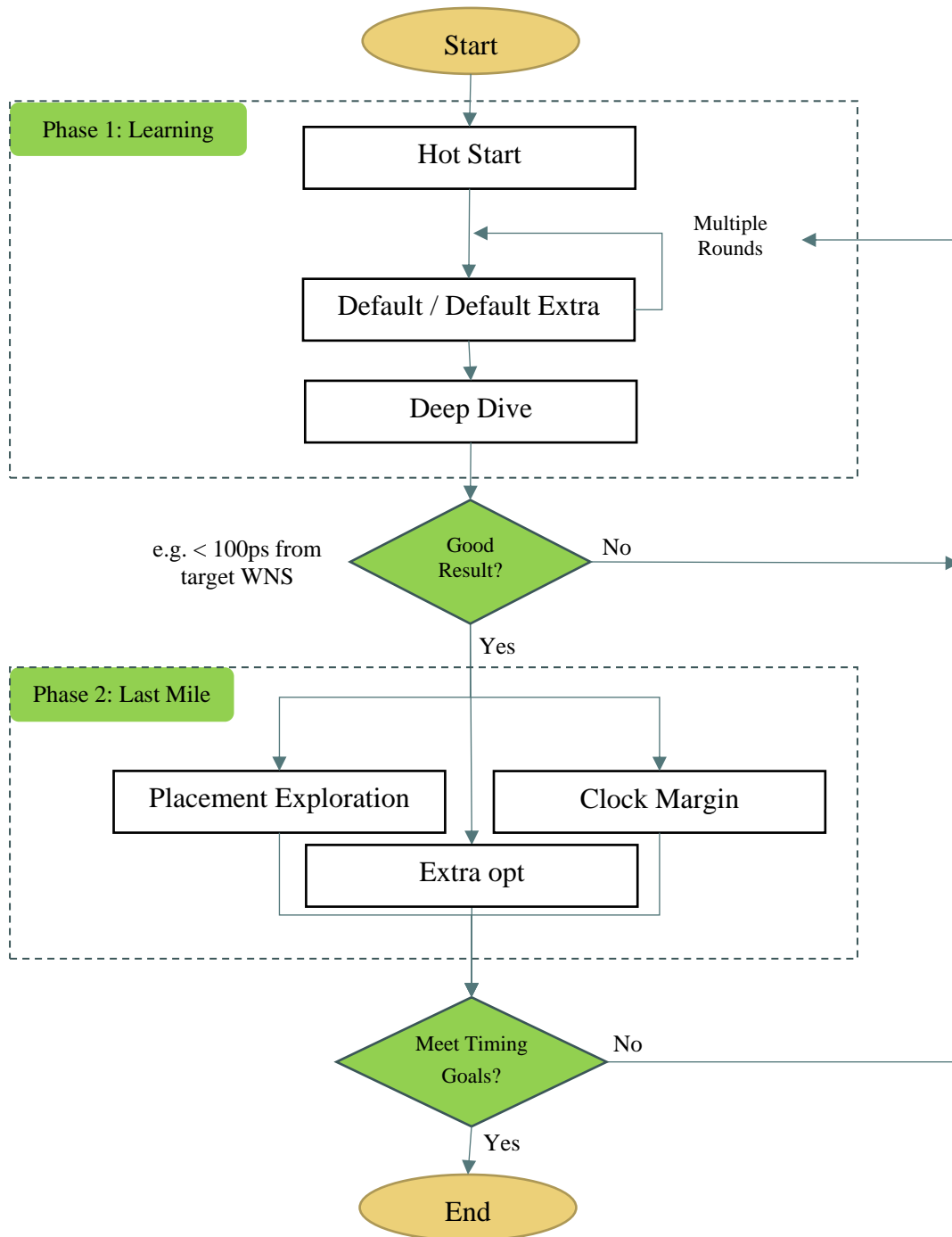
Firstly, use Learning recipes during Phase 1 to automatically select build parameters in light of all the build result data accumulated in previous rounds. Employ Last-Mile recipes when your design is close to meeting your timing targets. General recipes include basic compilation of the design as is, 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, the recommended Methodology is to assume that there are zero data points and start with the “Hot Start” (or “Explorer” for older versions of InTime) recipe, a Learning recipe. The figure below shows the typical optimization process of InTime for Vivado.

Figure 4: InTime Optimization Flow for Vivado Designs



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

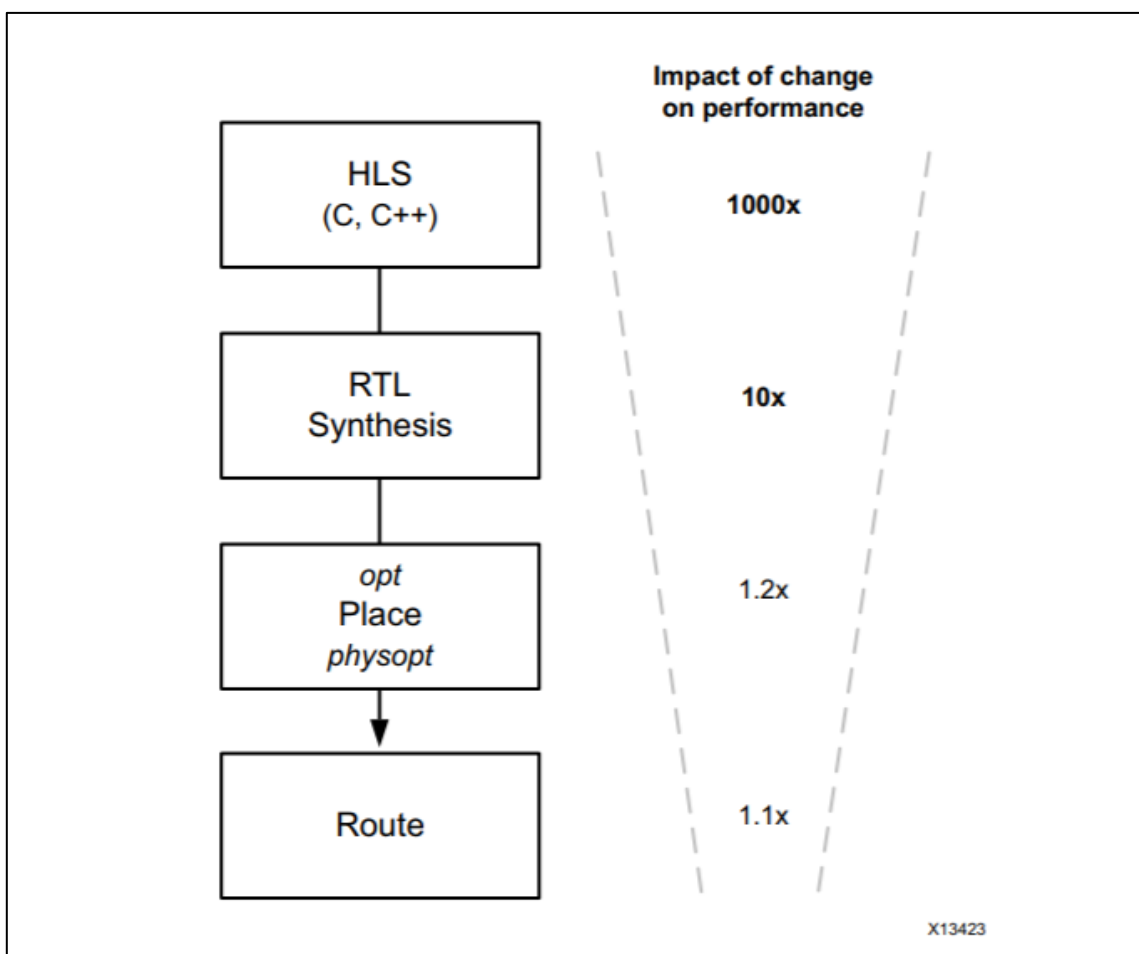
Hot Start can also be used as a litmus test to gauge whether or not timing closure is possible. Refer to the section “[Possibility of Meeting Timing](#)”

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 from the [Vivado UltraFast Design Methodology Guide](#).

Figure 5: Xilinx UltraFast Design Methodology



“As shown in the following figure, early stages in the design flow (C, C++, and RTL synthesis) have a much higher impact on design performance, density, and power than the later implementation stages. Therefore, if the design does not meet timing goals, Xilinx recommends that you revisit the synthesis stage, including HDL and constraints, rather than iterating for a solution in the implementation stages only.”

You can see that using the appropriate parameters for synthesis 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 tools' 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 algorithms, 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 a highly flexible recipe

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 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

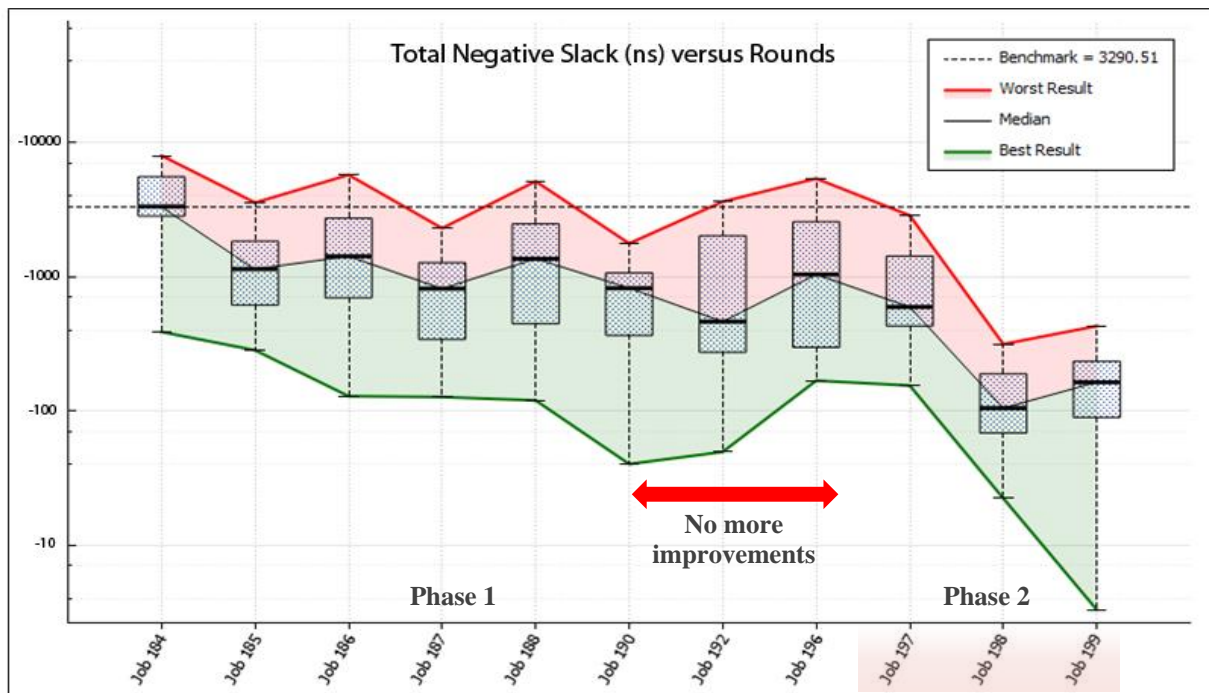


Figure 6 also illustrates how the “best result” in a round can fluctuate up and down over a number of rounds, as opposed to a consistent downwards trend. This is an expected artifact of design space exploration.

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 the user guide [“Set Flow Properties”](#) for more information

Goal-Based Build Parameter Selection

Default selects parameters based on your target goal, which can “Speed-TNS”, “Area” or “Power” in InTime’s Flow Properties section. Note that InTime does not optimize based on WNS, because the critical path responsible for WNS is usually different in various strategies and is generally not an apples-to-apples comparison between two 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 round in Phase 1.

For example, Figure 7 shows that only eight (can be up to 15) build parameters are different between a Deep Dive strategy and a Default strategy in a Vivado design, whereas the number of dissimilar parameters between two Default strategies can be 30 or more.

Figure 7: Differences in parameters for Deep Dive strategy

AUTO	SYNTH_DESIGN	FORCE
AUTO	CASCADE_DSP	8
AREAMAPLARGESHIFTREGTOBRAM	CONTROL_SET_OPT_THRESHOLD	ALTERNATEROUTABILITY
10000	DIRECTIVE	30000
NONE	FANOUT_LIMIT	NONE
AUTO	FLATTEN_HIERARCHY	ONE_HOT
OFF	FSM_EXTRACTION	OFF
1	GATED_CLOCK_CONVERSION	1
-1	KEEP_EQUIVALENT_REGISTERS	-1
-1	MAX_BRAM	-1
-1	MAX_BRAM_CASCADE_HEIGHT	-1
-1	MAX_DSP	2200
1	MAX_URAM	-1
0	MAX_URAM_CASCADE_HEIGHT	-1
FALSE	NO_LC	1
AUTO	NO_SRLEXTRACT	0
1	NO_TIMING_DRIVEN	FALSE
3	RESOURCE_SHARING	AUTO
FALSE	RETIMING	1
FALSE	SHREG_MIN_SIZE	3
FALSE	OPT_DESIGN	FALSE
FALSE	BLOCK_RAM_POWER_OPTIMIZATION	FALSE
FALSE	BLIFG_OPT	FALSE
FALSE	CARRY_REMAP	FALSE
FALSE	CONTROL_SET_MERGE	FALSE
DEFAULT	DIRECTIVE	DEFAULT
FALSE	HIER_FANOUT_LIMIT	FALSE
DEFAULT	ROUTE_DESIGN	HIGHERDELAYCOST
NONE	DIRECTIVE	NONE
TRUE	EXTRA_OPT	TRUE
FALSE	TNS_CLEANUP	FALSE
DEFAULT	POST_ROUTE_PHYS_OPT_DESIGN	FALSE
1	CLOCK_OPT	DEFAULT
FALSE	DIRECTIVE	1
FALSE	IS_ENABLED	FALSE
FALSE	ROUTING_BASED_OPTIMIZATION	TRUE
FALSE	SLR_CROSSING_OPT	

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, in the situations described below, go directly to Phase 2: Last-Mile Optimization.

1. Improvements in the Worst Slack or in the TNS have plateaued after three or more rounds. This behavior happens usually 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 targets 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 percentage range of improvement. Other recipe adopts an iterative flow on the checkpoint file (DCP) to achieve incremental improvements.

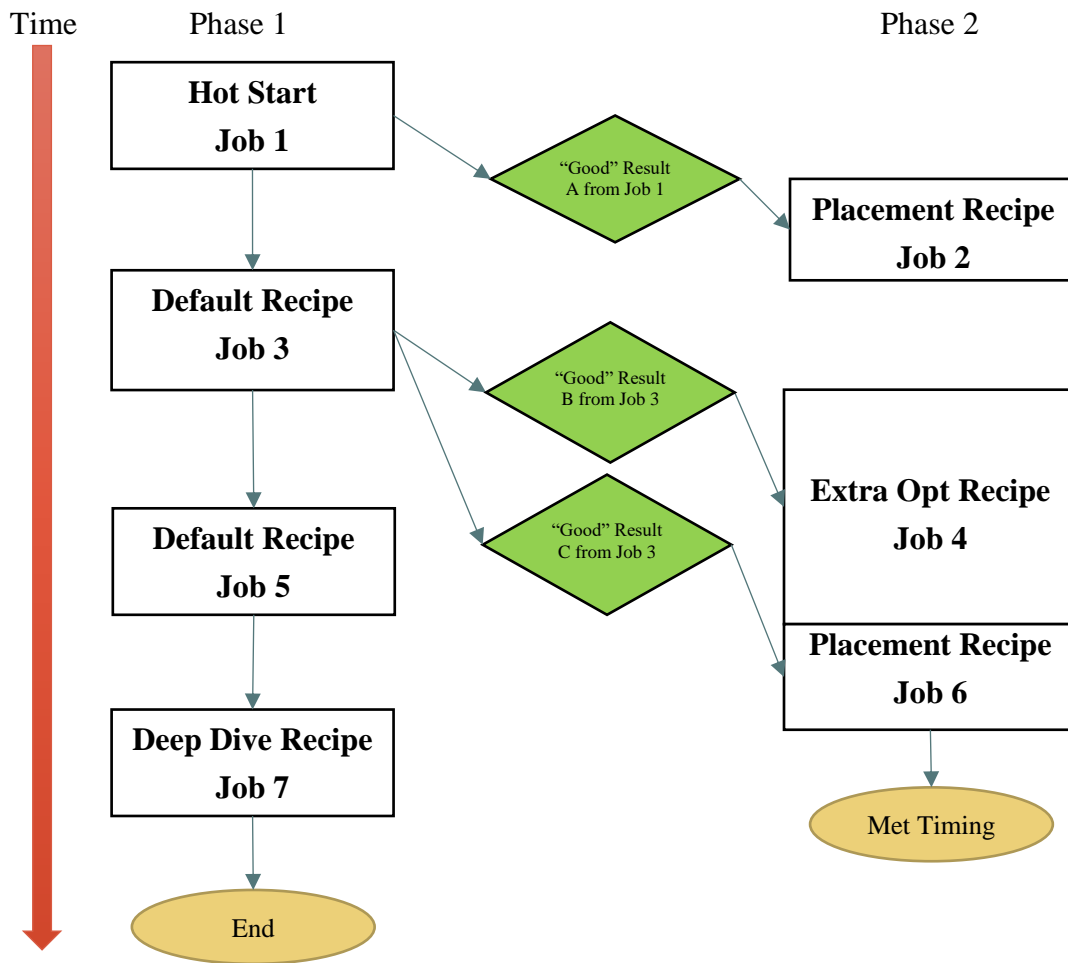
As mentioned earlier, use Last-Mile when you are close to meeting the timing target.

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 that you select. 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 in a previous section, 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

Recipes are constructed based on the build parameters and flows provided by the Vivado tools. Differences in those tools mean that some recipes are only available for certain tools.

	Tools	What it does
Extra Opt Exploration	Xilinx Vivado	This is an iterative flow where running placement, physical optimization and/or routing in a loop on the same Design Checkpoint (DCP), can yield better results. Note that the magnitude of optimization is dependent on the build itself, even for the same design. We recommend this strategy when the critical paths are consistently the same or related.

There are nine different strategies in this recipe. Each strategy can run up to 48 hours in nine iterations. If there are no improvements for three consecutive iterations, the recipe stops and the best result will be displayed.

Placement Exploration

Xilinx Vivado

As the placement seed sweep feature (also known as Cost Table sweep in ISE) is no longer available in Vivado, this recipe uses other means to trigger similar behavior. There is no upper limit on the number of strategies you can run, but we recommend no more than a hundred (100) and one round.

You can initiate this recipe either on an XPR or DCP project. The results will vary, with the XPR project giving a much larger magnitude of values.

For more information, please refer to this [article](#).

Clock Margin Exploration

Xilinx Vivado

This recipe uses the “clock uncertainty” property (as explained in Vivado [UG906](#)) to focus the implementation tool’s efforts on the appropriate clocks.

We recommend using this recipe on the best results of the Default, Default Extra and Extra Opt Exploration recipes. There is only one round, but there are two InTime flow properties to specify.

- *Clock Margin Initial Value* (ns)
- *Clock Margin Incrementing Step Value* (ns)

For more information, please refer to this [article](#).

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 providers, 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 limited build machines; first by reducing the required run time per round and then the number of rounds required to converge onto optimized results.

Likelihood of Meeting Timing

For Vivado 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) or 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 unlikely that timing closure can be achieved – design changes are probably necessary at that point.

Re-run Previous Strategies

In the spirit of design reuse, if (or when) a design is modified (for example, minor feature changes or bug fixes), we recommend re-using previously known-good strategies. The "Re-run Strategies" recipe in the General category allows you to select strategies to run again (see Figure 9 below).

Figure 9: Re-run best strategies

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

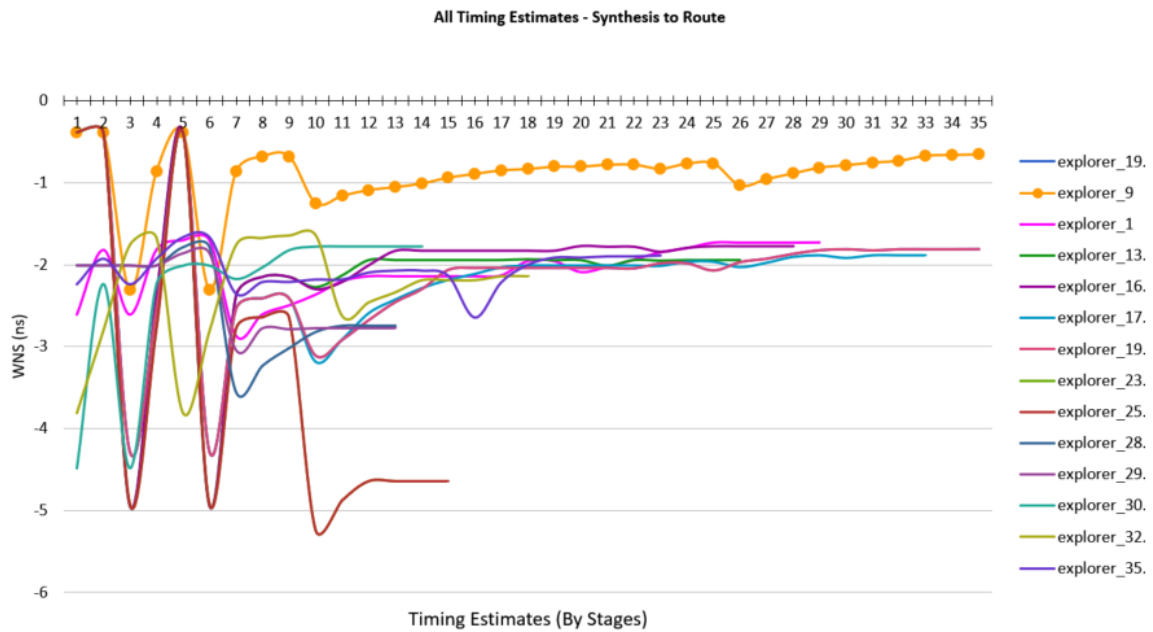
This approach works very well when you move to a newer version of your FPGA tool. With a newer version, design performance results may change even with default build parameters. Instead of running InTime Optimization Phases 1 and 2 rounds from scratch, this recipe takes advantage of possible correlations 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

Vivado timing estimates obtained after intermediate build steps provide a preview of the final results, with the caveat that the accuracy of the estimates increases as the build process approaches the routing stage. 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 compared to post-placement when the build is only halfway done. The chart below tracks Worst Slack estimates across different build stages for a design with about 70% logic utilization on a multi-die device. 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

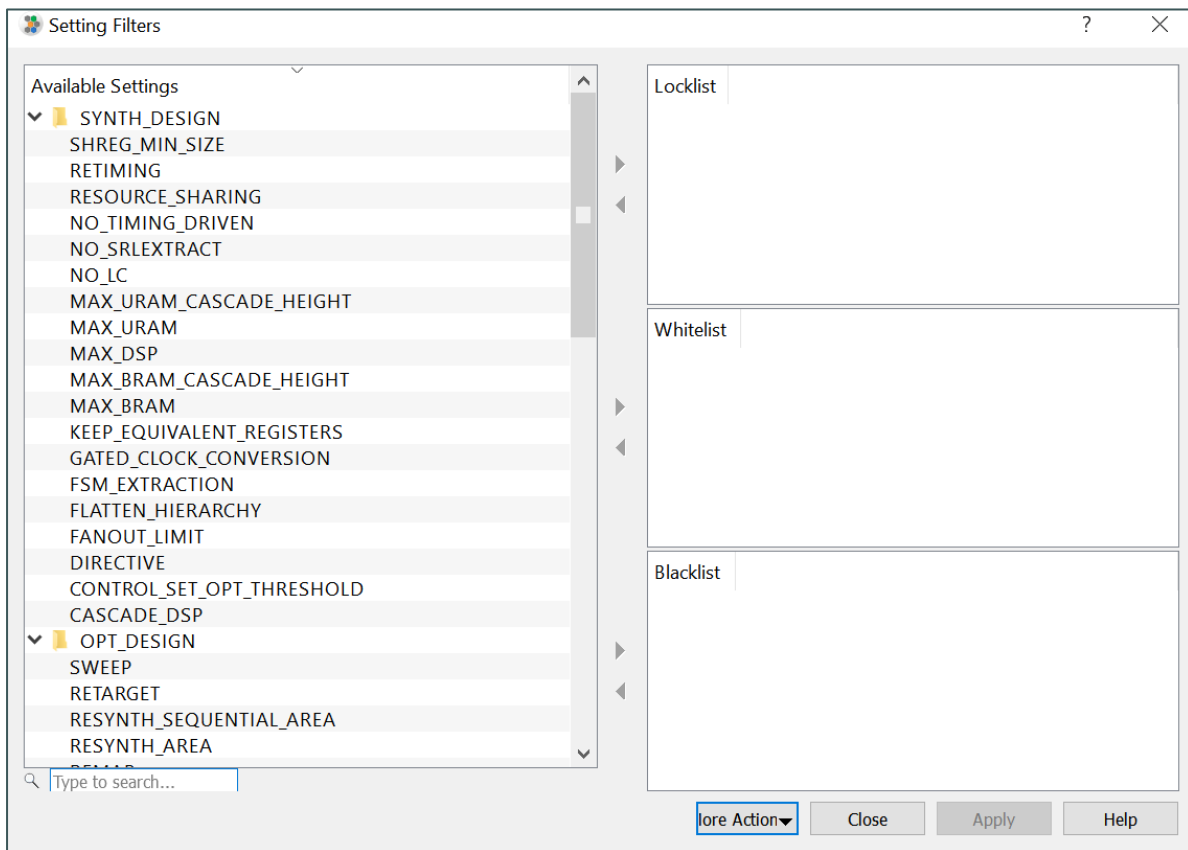


InTime has post-placement TNS and Worst Slack thresholds that you can specify to terminate poor builds and reallocate build resources to other strategies. More details about the relationship between post-placement and final timing results can be found in this [article](#).

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 what parameters should be included or excluded.

Figure 11: Settings Filters



For example, in a routing congestion scenario, the biggest impact on timing is often resource utilization; in other words, obtaining the best possible synthesis result is essential. In such a situation, running the Default recipe with known-good synthesis parameters in the “Locklist” portion of the Settings Filters ensures that the same synthesis parameters are used in all the 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. With new technologies like cloud computing, certain timing issues can be resolved without overhauling the entire RTL or sacrificing critical functionality.

Appendix A: InTime Recipes

	Recipes	Vivado	ISE	Descriptions
Learning	InTime Default	Y	Y	Performs first time calibration, exploration and optimization of your design
	InTime Default Extra	Y		Performs actions in Intime Default recipe, and run additional optimizations
	Hot Start	Y		Generates initial strategies for your design by correlating it with other designs in the InTime database
	Deep Dive	Y	Y	Performs deeper analysis of result of your design
	Explorer	Y		Explores different optimization in your design
Last-Mile	Extra Opt Exploration	Y		Explores optimization that focus on placement, physical optimization and routing
	Placement Exploration	Y		Explores the effect of placement adjustment on the design
	Clock Margin Exploration	Y		Explores the effect of clock uncertainty on the design
	Map Seed Exploration		Y	Explores the effect of map seeds on the design goal
General	Just Compile My Design	Y	Y	Compiles the active revision in your project
	Rerun Strategies	Y	Y	Rerun all marked strategies
Advanced	Custom Flow	Y	Y	Uses strategies specified by the user to compile

Document Revision History

	Date	Changes Made
1	08 May 2018	Initial Version