**This document is downloaded from DR-NTU (https://dr.ntu.edu.sg)
Nanyang Technological University, Singapore.**

Driving Timing Convergence of FPGA Designs through Machine Learning and Cloud Computing

Kapre, Nachiket; Chandrashekaran, Bibin; Ng, Harnhua; Teo, Kirvy

2015

Kapre, N., Chandrashekaran, B., Ng, H., & Teo, K. (2015). Driving Timing Convergence of FPGA Designs through Machine Learning and Cloud Computing. 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines, 119-126.

https://hdl.handle.net/10356/81244

https://doi.org/10.1109/FCCM.2015.36

© 2015 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. The published version is available at: [http://dx.doi.org/10.1109/FCCM.2015.36].

Downloaded on 29 Mar 2024 19:11:55 SGT

Driving Timing Convergence of FPGA Designs through Machine Learning and Cloud Computing

Nachiket Kapre, Bibin Chandrashekaran
School of Computer Engineering
Nanyang Technological University
Singapore, 639798
nachiket@ieee.org

Harnhua Ng, Kirvy Teo
Plunify Inc.
67 Ayer Rajah Crescent
Singapore, 139950
harnhua@plunify.com

***Abstract*—**
Machine learning and cloud computing techniques can help accelerate timing closure for FPGA designs without any modification to original RTL code. RTL is generally frozen closer to system delivery target to avoid injecting new unforeseen bugs or significantly affecting design characteristics. In these circumstances, developers trying to close timing are either at the mercy of random trials through placement seed exploration or through vendor-provided design space exploration tools that run a few compilation trials with changes to the CAD tool options (or parameters). Instead, we propose evaluating multiple CAD runs in parallel on the cloud, supported by a Bayesian learning and classification framework for generating multiple CAD parameter combinations most likely to help attain timing closure. We maintain a database of FPGA CAD tool parameters (input) along with associated variations in timing slack (output) to enable the learning process. A key engineering resource we use here is cheap and abundant parallelism made possible through cloud computing frameworks such as the Google Compute Engine. Across a range of open-source benchmarks, we show that learning helps improve total negative slack (TNS) scores by 10.5× (geomean) when compared to a single baseline run of Quartus 14.1 and by 7× (geomean) when compared to Altera Quartus 14.1 Design Space Explorer (DSE).

I. Introduction

In [1], Richard Feynman narrates his interest in picking locks during his time at Los Alamos. For his initial exploits, Feynman relies on being able to try multiple combinations as quickly as he can (and on sheer luck). However, as the locks get more complex, he devises increasingly clever tricks to prune the search space based on mental models of how the locks operated. Modern FPGA backend CAD tools are similar to Feynman's locks. While they serve an important purpose: compiling circuits to FPGA bitstreams, they are hard to get right (or crack). So far, designers have relied on ad-hoc tuning techniques and intuition to ensure that their designs meet user specifications. Borrowing from Feynman, we hope to uncover the fundamental principles behind the selection of CAD parameters and use that knowledge in an automated manner to guide timing convergence of user designs.

Modern FPGA design flows typically start with a design space exploration phase of the system-level aspects that inform low-level RTL design. During the RTL design phase, a large amount of engineering effort and costs are allocated to design verification. While pursing functional correctness, designers will occasionally run FPGA CAD tools to get a gut-feel for the resource utilization and timing scores of the final compiled design. However, majority of backend tool runs happen towards the latter stages of the design process. This is an unfortunate consequence of the long runtimes associated with modern CAD tools and the complexity of configuring the CAD tool parameters (*e.g.* command-line switches, project settings) for optimum results. This also means that if the design fails to meet timing by a small margin closer to the delivery window, any RTL change would impose significant engineering cost on the team. While some RTL modifications may be inevitable, it is often possible to tweak the CAD tool parameters to ensure designs meet their timing targets. Existing strategies for managing this selection rely on designer intuition accumulated from years of experience with the tools. A popular approach is to run multiple instances of the CAD tools with different placement seeds (cost tables). Both these approaches require relying on in-house expertise which is expensive to build or on pure luck to help achieve timing closure. Modern CAD tools such as Quartus often export hundreds of user-selectable parameters that are impossible to manage together using human intuition alone. More worryingly, the number of parameters has only increased with each generation of the CAD tool.

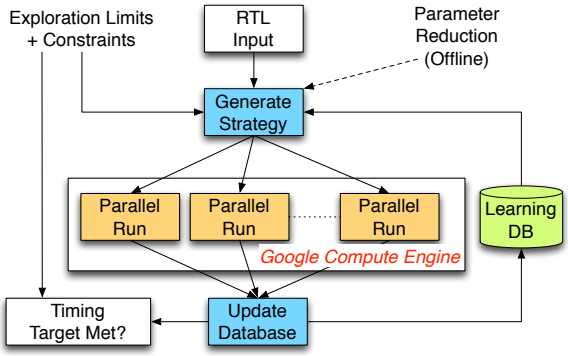


Fig. 1: High-Level Organization of InTime (showing use of cloud computing resource for parallel runs, and machine learning database for guiding convergence)

In this paper, we hope to replicate the benefits of designer

intuition through automated machine learning techniques, and *parallelism* through cloud computing resources using the **InTime** [2] tool. We show a high-level diagram of InTime in Figure 1. Unlike placement seed exploration, we expose a larger set of CAD tool parameters (60–70) to automated selection while relying on Bayesian techniques to approximate human intuition in sorting out influential CAD tool parameters and setting them correctly. Furthermore, we provide initial hints in the form of a starting set of parameters to drive the search in favorable directions based on device/tool characteristics and some high-level knowledge of the designs. We exploit parallelism to help collect enough samples to drive our learning algorithms. We then formulate a new set of parameters to attempt in the subsequent trials. By choosing the extent of parallelism, and number of learning steps, we can achieve timing closure faster than other techniques and naïve parallel exploration.

The key contributions of our paper include:

- Design of a Bayesian learning and classification and framework supported by Principal Component Analysis pruning of parameter sets for delivering timing closure for stubborn RTL designs.
- The development of cloud computing backend to parallelize the FPGA CAD runs to improve the quality of learning through sufficient sample generation.
- Quantification of the benefits of learning, impact of parallelism, evaluation of computing costs across a range of open-source benchmarks gathered from *opencores* [3] repositories.

II. Background

A. Multi-stage FPGA CAD Flow

The FPGA compilation flow starts with Verilog/VHDL as input and ends with an executable FPGA bitstream as its output. The internal stages are often organized as a series of sub-problems such as (1) synthesis, (2) mapping and packing, (3) placement, and (4) routing. Each of these sub-problems solve an NP-complete problem using heuristics and they often require multiples hours to days of runtime for the largest FPGAs available today. The Altera Quartus CAD tool supports up to 80 different CAD tool parameters that can be optimized or tweaked creating a design space that is impractical to explore exhaustively. In this case, without modifying RTL, we can adjust a select few synthesis parameters, packing constraints as well as placement-and-routing options i.e. at each step of the flow. However, the key challenge is to decide which parameters matter and how to change them. The correct operation and tuning of these heuristics must then rely on designer intuition and knowledge to help the design meet the system-level throughput goals such as timing targets. When the design does not meet timing, the designer attempts a trial-and-error approach by setting up multiple trials by modifying a few parameters to help guide convergence. As mentioned earlier, a common trick most developers will try is placement seed exploration which exploit randomness to drive the placement process. In Figure 2, we show the impact of such placement seed exploration on overall slack for the `vga` benchmark. Here slack represents the extent by which the design did not meet timing. Larger negative values of slack mean that the design is far away from meeting timing, while smaller values are closer and better. As we increase the number of seed exploration runs, the best timing slack observed continues to improve up to an extent. This clearly shows (1) the scope of improvements that are possible through a parameter exploration, (2) the need for multiple trials to exploit noise properties of FPGA CAD tools, and (3) the limitation of luck in directing convergence.

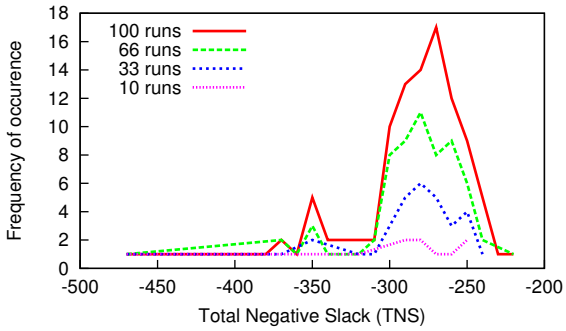


Fig. 2: Impact of Placement Seeds on Timing slack for the `vga` benchmark running on Quartus 14.1 Web Edition for Cyclone IV FPGA (More trials yield better results)

B. Related Work

The underlying concepts of smart design space exploration and machine learning techniques have been explored in allied domains of high-performance computing, computer microarchitecture optimization and CPU compiler flows. The complexity of modern CPU architectures and the compilation frameworks necessitates a rigorous exploration of the different interacting components in optimizing the final implementation.

In supercomputing systems, extracting every bit of performance benefits out of a single compute node have significant consequences at the macroscale. ATLAS [4] package allows HPC developers to automate the laborious process of code-generation and optimization of the numerical routines for each specific computer architecture and input problem size through selection of parameters such as cache blocking factor, loop unroll count, and others. In this case, the approach is applied to commonly-used libraries such as BLAS, and LAPACK and a brute-force exploration of the possible combinations is adequate. The optimization is applied to the shared libraries once during installation and then automatically picked up by any applications using these libraries. In our approach, each FPGA design is unique and a brute-force exploration is not feasible due the sheer size of the parameter space.

Modern CPU microarchitecture design is an increasingly challenging and complex process with several configurable choices at the designers disposal such as core counts/types, cache sizes, issue widths, ALU/FPU counts, and various pipelining options. In [5], the authors show a way to build regression models for designing CPU micro-architecture through

a strategic search of the design space (potentially reducing the search space of ≈22 billion points to less than 4000 samples) while achieving median error rates of ≈4%. In the scope of the parameter space size, this problem shares the challenge with the FPGA design process. However, each FPGA CAD invocation runs for hours instead of the few minutes of simulation time required for the CPU microarchitecture studies making it particularly important to drive convergence faster.

In [6], a smart iterative search strategy is used during C/C++ compilation phase to optimize the generated binary without modifying original source code. The tool explores the design space through embedded heuristic knowledge that allows pruning of the search space to manageable sizes. While modern C/C++ compilers are complex tools, the compile times are still much lower than the FPGA design flow. Coupled with the larger pool of C/C++ programs, this represents a larger training set and much faster training process compared to FPGA design.

There has been work on quantifying the impact of CAD tool noise for open-source academic CAD tools in [7]. Here, the authors identify the impact of changes to solution quality when using VPR 5.0.2 under different timing targets and input net ordering. They report a critical path delay gap in the range of 17-110% when compared to nominal behavior. Academic FPGA tools are much simpler than commercial tools, but it is useful to see the scope of improvements possible in even such simple tool flows.

In [8], the authors develop a strategy inspired by Design of Experiments (DoE) to customize the parameters of the soft processor design space. They do this by carefully selecting a subset of the parameters and their associated ranges for experimentation. In contrast with our approach of targeting the backend flow while keeping RTL fixed, the DoE-inspired approach modifies the RTL generators to improve results. Additionally, we use machine learning techniques to drive selection of parameters rather than rely on DoE techniques.

In [9], the authors consider the impact of ordering of LLVM passes on the quality of hardware solution for high-level synthesis. They observe a variation in excess of 10% by composing various compiler *passes* in different ways. This study is similar to the approach taken in [6], in that it focuses on compiler-level optimizations instead of modifying original C code. However, even here the changes in HLS compiler options translate into RTL-level changes which require re-verification before shipping the design. Additionally, the predictive models used in the HLS tools are still not sufficiently accurate to predict the post place-and-route timing results due to the complex interactions with the backend flows. Our approach freezes the RTL and relies only on the final placed-and-routed timing scores to drive learning.

III. Machine Learning for FPGA CAD

A. InTime Organization

At a high level, InTime is a software plugin for FPGA CAD tools. Given a timing target specification, RTL code and user constraints, InTime will formulate a strategy to help guide the CAD flow towards timing convergence through multiple rounds of intelligent trials. It derives its intelligence by learning from previous trials to formulate the next plan of action. We have previously shown a representation of the InTime flow in Figure 1. We now show an abstract execution profile in Figure 3.

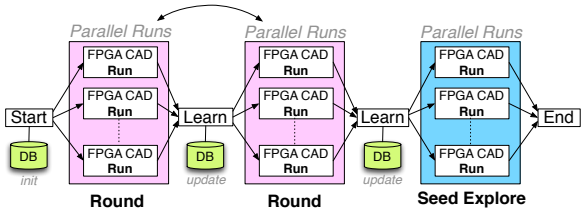


Fig. 3: Execution Steps of InTime (showing 3-round scenario, but multiple rounds possible)

Two key aspects of InTime stand out: (1) the application of machine learning with a metrics database, and (2) the use of cloud computing resources. InTime organizes the execution in a series of sequential *rounds*; each round contains a series of parallel *runs*. Within each run, InTime will generate an assignment of values to CAD tool parameters. While the parameters are internally represented as boolean values, we also support continuous variables as well (see next section). Runs within a round do not interfere with or depend on each other in any way and can be fully parallelized subject to available compute resources. Each round is a synchronization point to facilitate the learning algorithm to adjust its predictions based on the results of the runs in the previous rounds. While there are many ways to accumulate and share learned knowledge, we presently start with a clean database for each design. However, when formulating CAD parameters for a run for the first time, we use statically formulated strategies that are known to work best for that particular device or tool version of design property according to CAD tool documentation and engineering experience. In this particular respect InTime operates in a manner similar to Altera DSE. As InTime learns the peculiarities of the design, the assignments to the CAD tools parameters start to converge to their ideal final values. The learning algorithm observes what the timing results are good and what are bad, and keeps track of the CAD parameters for each result. If necessary, at the end, closer to convergence, InTime performs a cleanup phase that runs a short placement seed exploration round to *mop up* any available slack. The use and availability of cheap cloud computing resources are also crucial to help make InTime feasible. While InTime can work with any compute cluster, the elastic ability to scale compute resources as desired in cloud-based shared environments is a cost effective approach.

B. Bayesian Classification and Learning

Machine learning routines are at the heart of the InTime flow. Without data analysis and learning, we would simply be relying on untracked, undigested statistical behavior and

noise in CAD tools to throw up a better result. As discussed earlier, each stage of the FPGA CAD tool can influence the final solution. Our learning approach first identifies and exports the list of tunable CAD tool options that are considered *safe* from the perspective of generating netlists (*e.g.* disabling or modifying timing target or overriding user constraints is not permitted). While the list of possible parameters available is large ≈80, we restrict our attention to 60 or so parameters depending on the target device/tool version.

We formulate the learning problem by first associating a boolean variable x_i for each CAD tool parameter i ($1 \leq i \leq N$ where N is the total number of CAD parameters). To keep the parameter analysis simple, we assume that each CAD tool parameter is generally independent of the others (see Section III-C on how we orthogonalize the parameters). Every parameter has a default value that is used when it is not explicitly set by the user (or by InTime). Most parameters turn out to have only two possible values (boolean yes/no options). For parameters with a greater number of discrete choices, we convert those into boolean variables encoding each independent choice. Similarly, for continuous variables, we classify the range into sub-ranges (bins) and assign variables for a sub-range (bin). Since the relationship between the input X and output y is uncertain due to the noise effects in FPGA CAD tools, we model this uncertainty through probabilities. Hence, we associate a probability p_i with each variable x_i. This probability indicates the likelihood of that variable (CAD tool parameter) affecting timing convergence. When p_i is close to 1, we enable the corresponding CAD tool parameter. The result of the CAD tool execution is captured in a solution variable y_j for each invocation of the CAD flow j ($1 \leq j \leq M$, where M is the total number of CAD runs). Since InTime is aimed at delivering timing convergence, y is the total negative slack in the design (but, this could be area, power or some other figure-of-merit). Thus, we have X_j representing the inputs to the CAD tool and $y[j]$ capturing the results for run j. The history of all observed y values are captured in Y. Not every combination of CAD parameters can result in a successful compilation. Sometimes the CAD tools themselves may crash due to software bugs or server issues, such as having insufficient computational memory. Overly aggressive CAD parameters can also cause a design to not fit into the target FPGA, resulting in a compilation failure. InTime tracks all failures as well so that it does not repeat them in future runs, for *he who does not learn from history is doomed to repeat it*.

Every FPGA design has an initial value of y produced from a single invocation of the CAD tool. If y is negative, then the design has not met its timing target. If y is zero, then it has met its timing target. Our overall goal is to obtain one or more X for which y is better than the initial value (ideally 0). Learning proceeds in a series of stages which can be represented as a function f that generates new proposals for X based on the previous history of X_j-$y[j]$. To ensure learning has enough samples for making intelligent proposals, we need results from multiple CAD runs before invoking the learning step.

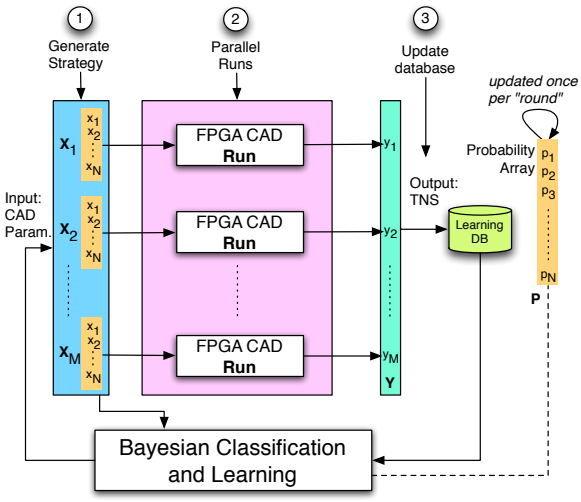


Fig. 4: High-Level Flow Diagram of the InTime Learning Process (X_i is a vector of input CAD parameters for run i, Y is the vector of all TNS scores across M runs. P is a probability vector associated with each CAD parameter.

Our overall approach, shown in Figure 4, consists of three steps that are run in a loop as requested by the user:

1. **Generate Strategy**: In this stage, we generate candidate proposals for estimating values of X that are likely to contribute to better values of y. We run Bayesian learning to update our parameter probabilities to generate these candidates for X. This is the *learning* stage of our flow.
2. **Parallel Runs**: Once we have a set of proposals for X, we launch a CAD run for each proposal. Each run within this proposal set (called a *round* earlier) is independent of another. This is where we exploit *cloud computing*.
3. **Update Database**: Once we collect the results y for a given set of X proposals, we store the results in our database for a subsequent use.

We now explain our Bayes classifier in more detail. Assuming all FPGA tool options are conditionally independent of one another, given Y, we can describe the Naïve Bayes Classifier to obtain the probability distribution over possible values of Y in Equation 1.

$$P(y = y_k | X_1 ... X_n) = \frac{P(y = y_k) \prod_i (P(X_i | y = y_k))}{\sum_j (P(y = y_j) \prod_i (X_i | y = y_j)} \quad (1)$$

Given a new vector X of CAD parameters used in actual compilation runs, Equation 1 shows how to calculate the probability that the corresponding solution TNS, y, will take on a particular value. Here $P(Y)$ is the array of probabilities that captures the likelihood that the particular input parameter set $X_1 \cdot X_N$ yields a lower timing slack score. The term $P(X|Y)$ is the conditional probability of each CAD tool parameter influencing the resulting slack. To teach the classifier what

is preferred, we tag each value of y as "Good" if greater than the initial value of y (less negative slack) and "Bad" if equal or less than the initial value of y (more negative slack). Each value of X and Y in the training set is then fed to the Naïve Bayes classifier. After training, the classifier produces the probability that Y will be "Good" or "Bad" given a new X. In other words, each combination of FPGA tool options will be tested against this model to predict if it will produce a better or worse timing result. For each design, the predicted "Good" X will be used to compute the timing results. For learning to be effective, we need to collect a sufficient number of samples of X and the corresponding Y to ensure that the classifier has a reasonable population sample. It is worth noting that the classifier stages run in a few seconds on input vector sets of 60–70 (#parameters) $\times$100–200 (#runs). If the predictions prove effective, the time saved on not having to perform unproductive exploratory compilations will more than compensate for the time spent on gathering samples and training the classifier. At the end of each round of compilations, if the predictions of X were accurate, in other words, predicted "Good" X resulted in better timing performance and X flagged as "Bad" really led to poorer TNS, InTime will subsequently update the probability p_i associated with each variable x_i so that future generated strategies become more and more informed.

C. Parameter Reduction

The online component of InTime manages the execution of the parallel CAD run, prediction via classification and directs the learning rounds. There is also an offline component in this flow that is run to help improve the effectiveness of classification and identify independent variables. This step need not be explicitly carried out by the end user, but is part of the data analysis methods present in the InTime software. It is typically run for each new device family, speed grade and CAD tool version. We already know that the space of possible CAD tool parameters that we can tweak is large. Of the $\approx$80 available CAD tool parameters, not all affect timing performance in significant ways. Some may influence area rather than timing. Others may be design-dependent; for example, some parameters may not be of much use in DSP-heavy applications and will potentially confuse the classification phase. Even different FPGA device families and Quartus versions exhibit varying cause-and-effect relationships between CAD tool parameters and compilation results. One potential drawback of having too many parameters or dimensions of analysis is known as over-fitting, whereby the classifier's effectiveness decreases beyond a certain number of parameters. Intuitively, the classifier generalizes better if it does not learn about exceptions or random effects caused by some CAD parameters which are not really meaningful influences at all.

Parameter reduction, as the name suggests, is a technique to help prune the parameter exploration space to a manageable size by eliminating those parameters that matter less for the device/tool combination. We use the popular Principle Component Analysis (PCA) to determine the most significant CAD parameters that affect the FPGA design under optimization. PCA aims to discover the variables which account for the most variability in a dataset. Unfortunately, a simplistic application of PCA can be susceptible to outliers (*e.g.* a few FPGA CAD runs gone bad) so outliers are removed using the boxplot method. Assume that $Q1$ and $Q3$ refer to the first and third quantiles of Y and IQR (the Inter-Quantile Range) is the difference between $Q3$ and $Q1$. We filter out all results where $TNS > Q3 + IQR \times 1.5$ and $TNS < Q1 - IQR \times 1.5$ as suspect outliers. It might appear counter-intuitive to discard overly good TNS values as well as exceedingly bad ones but because PCA uses variances to determine parameter significance, "too good" values as well as "too bad" ones might skew the analysis outcome towards certain parameters. We want to avoid getting caught in local minima traps.

PCA in general also requires a large amount of training data to be effective. The first significant variable, or Principle Component (PC), is the one that causes the most variance in Y. The subsequent PC is orthogonal to the previous one and must cause the maximum variability among the remaining variables. In computational terms, we find the PCs by calculating the eigenvectors and eigenvalues of the covariance matrix, a representation of how each data point is related to another.

By sorting the eigenvectors in order of decreasing eigenvalues, InTime picks the m largest eigenvalues to represent the Components in order of significance. Those Components in turn correspond to m CAD tool parameters that produce the most variation in timing results for that design. Here, m is determined by a process of experimentation where the m CAD parameters corresponding to the largest m eigenvalues were used to compile the FPGA design. The m for which the TNS most closely matches that obtained by the full set of CAD parameters is deemed the optimal m. We generally find that selecting m from 5 to 20 delivered the best results across our benchmark set.

IV. Methodology

A. Compute Setup

We conduct our experiments on the Google Compute Engine across a series of 6 compute nodes configured for the task. We set up an instance group of 5 high-memory machines (machine-type `n1-highmem-4`) with the configuration of 4 vCPUs (virtual cores) and 26 GB of RAM per machine. In addition, we configured a sixth master node (machine-type `n1-standard-2`) with 2 vCPUs and 7.5 GB of RAM for hosting our CAD tools installation disk (500 GB standard persistent disk exported over NFS to other machines). All the instances run Ubuntu 14.10 64-bit Linux hosted on a 10 GB persistent disk that holds the operating system. We run InTime in server-client mode where the server runs on the master node while clients run on each of the instance group machines.

Multiple rounds of compilation runs, or "jobs", are submitted to the server which then farms them out to the clients for execution. When compilation is done, the server collects

Benchmark	Lines of Code	LUTs	FFs	Estim. Freq. (MHz)
aes	676	4439	3968	153.84
ecg	1422	14663	7443	153.84
switch	52454	9712	7020	111.11
vga	2647	1525	822	181.82
viterbi	1266	3626	1250	153.84
xge-mac	3173	2969	1776	153.84

TABLE I: RTL and Post-Synthesis Characteristics of Benchmarks showing varying complexities, characteristics and logic requirements

the results into a single database which drives the learning process. We use Altera Quartus 14.0 and 14.1 Web Edition for our experiments and select the Cyclone IV EP4CE115F29C7 as the target FPGA. While the target FPGA is a small device, our benchmark designs occupy significant portions of the chip sizes stressing the CAD tool execution. We supply specific timing targets to our benchmarks and disable insertion of IO buffers to enable our benchmark PIs to map successfully in cases where the number of input and output pins in a design exceed that available on the target FPGA.

Furthermore, we specified maximum runtimes for each job so that a run is considered a failure without any result if its compilation runtime exceeds two times the duration of its initial compilation runtime. In general, the CAD tools tend not to produce better results if they cannot converge within this timeframe guideline.

For experiments and tuning of our benchmark set, with our machine configurations mentioned earlier, the overall cloud compute costs were roughly $300–400 (USD). This is extremely competitive when compared to engineering, verification and RTL modification costs that may otherwise be necessary without our tools.

B. Benchmarks

We use a range of open-source benchmarks from **opencores** repositories to support our experiments. We tabulate the post-synthesis characteristics of these benchmarks in Table I. These benchmarks cover a range of application domains and varying problem sizes that can stress our CAD flow. We intentionally provide hard-to-achieve timing targets during our experiments to force InTime and Quartus to work tirelessly to achieve positive outcomes. Hence, even though the TNS values do not reach 0, we can lower them to much greater extent than Altera DSE as shown in the following Section V. For the Altera DSE flow, we configure the optimization to use the "Exhaustive Search of Exploration Space" search method when exploring the solution space.

C. Learning Tools

We perform bulk of our learning computations using R [10] and Weka [11] machine learning libraries. For offline PCA analysis, we also exploit a cloud-based offering called IBM Watson Analytics that helps identify the most important parameters for our dataset with minimal setup. We do not, however, use Watson during online execution as the product is still in beta, and lacks a reliable API. We write our own Bayesian learning routines in R that run in each round.

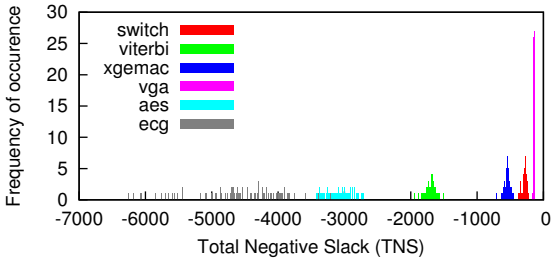


Fig. 5: Range of timing scores (TNS) for various `opencores` benchmarks with placement seed exploration

V. Results

We now describe the results of using InTime on our benchmark set. We first highlight the impact of placement seed exploration on TNS distribution, then describe the results of running InTime in various configurations on TNS scores, compare InTime against Altera DSE and closely inspect a few cases to showcase the benefits of InTime.

A. Timing Slack Distribution for Placement Seed Exploration

In Figure 5, we show the result of performing placement seed exploration runs across our benchmark set (100 runs per benchmark). Designs with large negative values of timing slack are furthest away from meeting their respective timing targets. The slack distributions show an interesting effect where those designs that are further away from achieving timing closure tend to show larger variance in slack values than those that are closer. InTime is capable of either (1) improving slack values; (2) delivering lowest slack values possible with less runtime; or (3) minimizing the area taken up by the compiled design in the target FPGA.

B. Effect of InTime on Achieved TNS

In Figure 6, we investigate the effect of InTime learning on the `viterbi` and `vga` benchmarks respectively while keeping the total number of runs fixed at 100. We compare our timing scores with the default Quartus run result and the red bars indicate worse scores while the green bars indicate better scores. We consider multiple InTime configurations and summarize our experimental observations below:

- **1 round 100 runs**: In this scenario, there is no machine learning and simple use of parallelism with InTime's canned strategies. As expected, bulk of our results show no improvement with a few outliers improving their results. In such a scenario, we are simply exploiting parallelism made available with cloud computing resources. Instead, the use of placement seed exploration is likely to yield better results.
- **3 rounds 33 runs**: Here, each round corresponds to a *learning event*, and in this case, we have 3 opportunities to enforce learning. As observed, we note a significant improvement in timing scores after the first round which continues until termination.

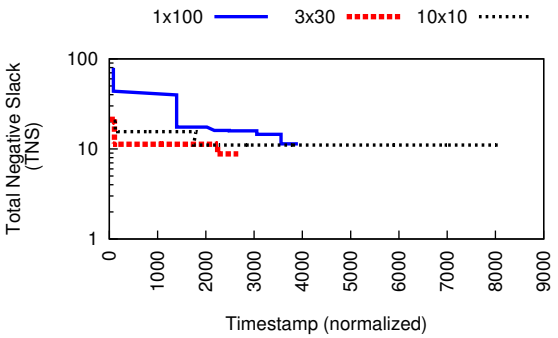


Fig. 7: Tracking Timing Scores over Time for `vga` benchmark

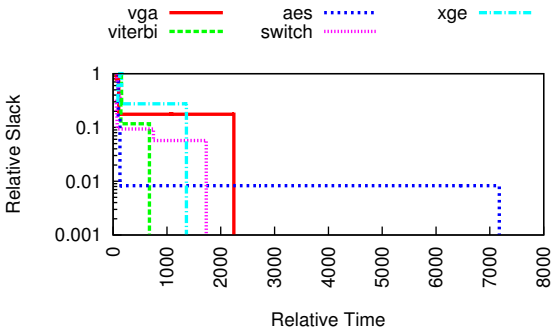


Fig. 8: Scaling TNS values across benchmarks

- **10 rounds 10 runs**: In this scenario, we split the 100 total runs into 10 rounds with 10 distinct learning events. One may expect, *more* learning to be useful, but that is clearly not the case. In this situation, the limited number of runs per round constrain the extent of exploration that is needed to make intelligent judgments over the next steps to consider. A middle ground approach where we provide the tools enough freedom to explore before committing to learning is ideal. *Nicely resonates with what happens in real life where taking time away to discover yourself is often quite valuable*.

C. Comparing DSE and InTime

In Figure 7, we show the impact of using various InTime configurations on TNS convergence as a function of cloud compute time. We can see how the 3×33 setting converges the fastest to the lowest TNS value than the other configurations. The 10×10 setting performs particularly poorly by taking more time and converging to a poorer result.

In Figure 8, we see the impact of normalized slack values as a function of time (also normalized) taken to deliver the solution across all benchmarks. On one hand, designs such as `viterbi`, `xge` and `vga` converge to their minimum slack values relatively fast as they have low TNS scores to start with. On the other hand, `aes` shows substantial improvement in quality but does take longer due to longer compiles.

Benchmark	**Total Negative Slack (best)**				
	Quartus (1 run)	DSE	Δ	InTime	Δ
aes	2905.4	2964.8	1×	148.9	20×
ecg	4767.4	4963.9	1×	2437.2	1.9×
vga	32.4	18.6	1.7×	8.8	3.6×
switch	3921.5	529.4	7.4×	98.2	40×
viterbi	2348.4	2289.9	1×	21.4	111×
xge-mac	542	511.8	1.1×	235	2.3×
Geom. Mean			1.5×		10.5×

TABLE II: Comparing InTime with Quartus (1-run) and Altera Design Space Explorer

D. Comparing DSE and InTime configurations

In Table II, we compare the results of InTime exploration with default Quartus runs as well as Altera's DSE tool. Across all our benchmarks, we are able to outperform DSE with lower timing scores. In some cases, where these values are close, the use of parallelism through Google Compute Engine helps to accelerate convergence. The stubborn `aes` benchmark in particular shows a promising reduction in slack. The `vga`, `viterbi` and `aes` benchmarks have the lowest slack scores despite high initial starting slack costs. The `ecg` is a particularly tough benchmark for all tools, and even though the reduction in slack is almost 2× the overall score still stays high.

VI. Conclusions

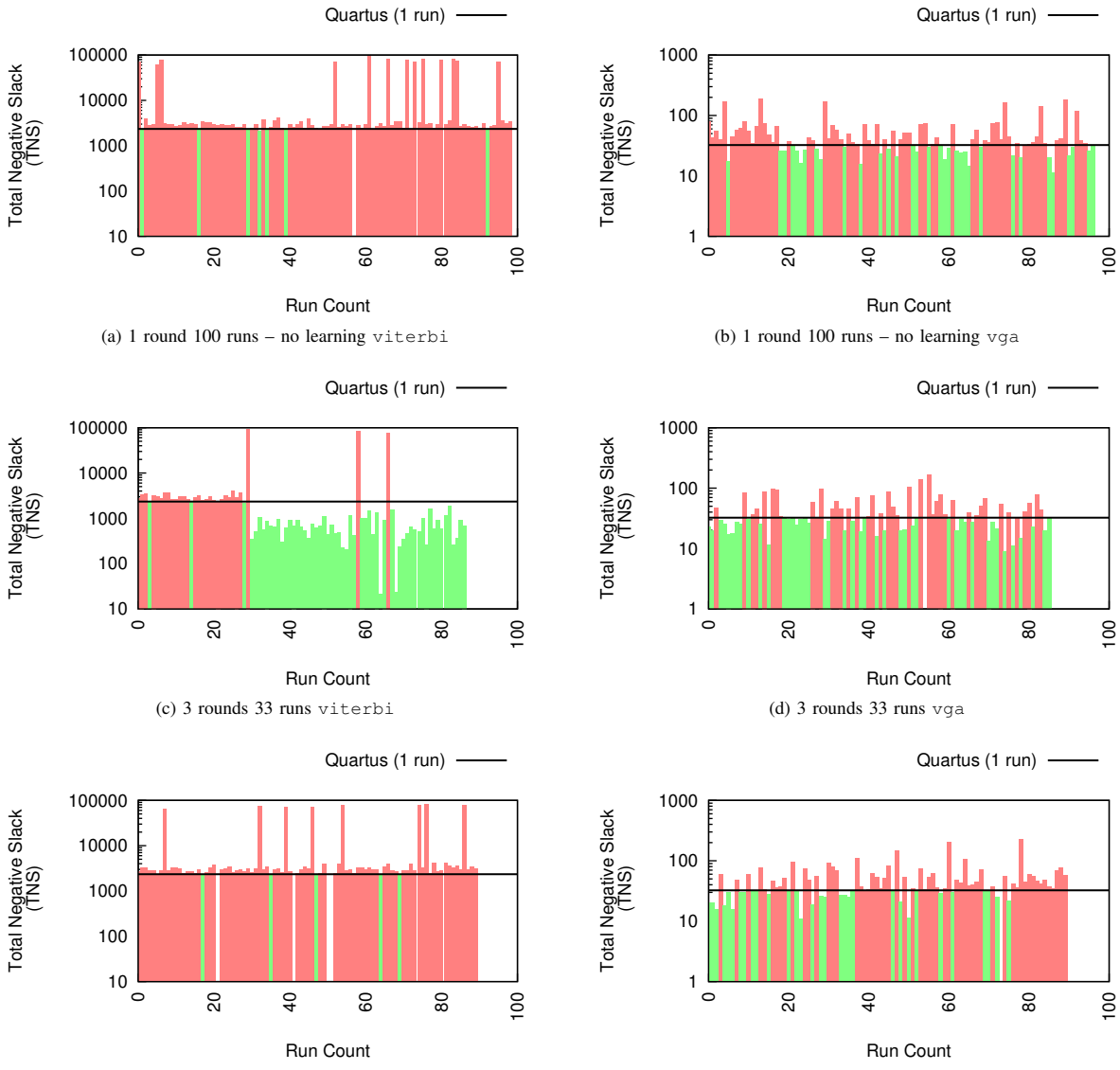
Charles Darwin [12] shows us how nature manages the process of improving life through evolution. Evolution is a mechanism of searching and discovering better solutions through multiple organic random trials and the survival of the fittest. Through a combination of parallel processing with cloud computing (multiple trials) and Bayesian classification and learning rounds (survival of the fittest), we can automate the process of choosing the ideal combination of FPGA CAD tool parameters for improving timing slack. For learning to be effective, we need to generate sufficient samples to drive convergence. We find the configuration with 3 learning rounds and 30 parallel CAD runs per round to generally work well for our benchmarks. We demonstrate substantially superior timing slack results compared to Altera's Design Space Explorer by 7× (geomean) across a range of `opencores` benchmarks. We hope to continue exploring newer machine learning algorithms, predictive models based on RTL characteristics and adaptive parallelism to further improve InTime while lowering cost.

References

[1] R. P. Feynman, R. Leighton, and E. Hutchings, *"Surely You're Joking, Mr. Feynman!": Adventures of a Curious Character*. W W Norton and Company Incorporated, 1985.

[2] N. Kapre, H. Ng, K. Teo, and J. Naude, "Intime: A machine learning approach for efficient selection of fpga cad tool parameters," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15. New York, NY, USA: ACM, 2015, pp. 23–26. [Online]. Available: http://doi.acm.org/10.1145/2684746.2689081

[3] Community, "OpenCores: Free open source IP Cores and Chip Design," 2004.



(a) 1 round 100 runs – no learning `viterbi`

(b) 1 round 100 runs – no learning `vga`

(c) 3 rounds 33 runs `viterbi`

(d) 3 rounds 33 runs `vga`

(e) 10 rounds 10 runs `viterbi`

(f) 10 rounds 10 runs `vga`

Fig. 6: Evolution of timing scores (TNS) for InTime-driven optimization for `viterbi` (left column) and `vga` (right column) benchmarks. Here, green bars indicate better TNS scores, and red bars indicate worse TNS scores than a single Quartus run

[4] R. C. Whaley and J. J. Dongarra, *Automatically tuned linear algebra software*. IEEE Computer Society, 1998.

[5] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *ASPLOS XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. ACM Request Permissions, Nov. 2006.

[6] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August, "Compiler optimization-space exploration," *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pp. 204–215, 2003.

[7] R. Y. Rubin and A. M. DeHon, "Timing-driven pathfinder pathology and remediation: quantifying and reducing delay noise in VPR-pathfinder," in *FPGA '11: Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM Request Permissions, Feb. 2011.

[8] D. Sheldon, F. Vahid, and S. Lonardi, "Soft-core Processor Customization using the Design of Experiments Paradigm," in *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE '07*, 2007, pp. 1–6.

[9] Q. Huang, R. Lian, A. Canis, J. Choi, R. Xi, S. Brown, and J. Anderson, "The Effect of Compiler Optimizations on High-Level Synthesis for FPGAs," *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, pp. 89–96, 2013.

[10] T. R. C. Team, *R: A Language and Environment for Statistical Computing*, Aug. 2013.

[11] M. Hall, E. Frank, G. Holmes, and B. Pfahringer, "The WEKA data mining software: An Update," *SIGKDD Explorations*, vol. 11, no. 2, 2009.

[12] C. Darwin, *The Origin of Species by Means of Natural Selection, or the Preservation of Favored Races in the Struggle for Life*. HarperCollins Publishers, 1859.