A Parallel Bandit-Based Approach for Autotuning FPGA Compilation

Chang Xu[1,*], Gai Liu[2], Ritchie Zhao[2], Stephen Yang[3], Guojie Luo[1], Zhiru Zhang[2,†]

[1] Center for Energy-Efficient Computing and Applications, Peking University, Beijing, China
[2] School of Electrical and Computer Engineering, Cornell University, Ithaca, USA
[3] Xilinx, Inc., San Jose, USA

*changxu@pku.edu.cn, †zhiruz@cornell.edu

Abstract

Mainstream FPGA CAD tools provide an extensive collection of optimization options that have a significant impact on the quality of the final design. These options together create an enormous and complex design space that cannot effectively be explored by human effort alone. Instead, we propose to search this parameter space using autotuning, which is a popular approach in the compiler optimization domain. Specifically, we study the effectiveness of applying the multi-armed bandit (MAB) technique to automatically tune the options for a complete FPGA compilation flow from RTL to bitstream, including RTL/logic synthesis, technology mapping, placement, and routing. To mitigate the high runtime cost incurred by the complex FPGA implementation process, we devise an efficient parallelization scheme that enables multiple MAB-based autotuners to explore the design space simultaneously. In particular, we propose a dynamic solution space partitioning and resource allocation technique that intelligently allocates computing resources to promising search regions based on the runtime information of search quality from previous iterations. Experiments on academic and commercial FPGA CAD tools demonstrate promising improvements in quality and convergence rate across a variety of real-life designs.

1. Introduction

Over the last three decades, FPGAs have evolved from a small chip with a few thousand logic blocks to billion-transistor system-on-chips containing hardened DSP blocks, embedded memories, multicore processors, alongside millions of programmable logic elements. Concurrently, FPGA development tools have also grown into sophisticated design environments. Compiling an RTL design into bitstream typically involves heuristically solving a sequence of complex combinatorial optimization problems such as logic synthesis, technology mapping, placement, and routing [7].

To meet the stringent yet diverse design requirements from different domains and use cases, modern FPGA CAD tools commonly provide users with a large collection of optimization options (or parameters) that have a significant impact on the quality of the final design. For instance, the placement step alone in the Xilinx Vivado Design Suite offers up to 20 different parameters, translating to a search space of more than 10^6 design points [3]. In addition, multiple options may interact in subtle ways resulting in unpredictable effects on solution quality. Traditionally, navigating through such an enormous design space requires designers to rely on either prior design experience or vendor-supplied guidelines. Such ad hoc design practices incur costly manual effort to achieve the desired quality of results (QoR). Worse, each new design may require a drastically different set of options to achieve the best QoR [24].

One solution to improve design productivity is employing meta-heuristic search techniques to explore the parameter space automatically. Figure 1 shows the improvement of the worst negative slack (WNS) of three designs generated by Vivado, each tuned using three different search techniques: active learning, Bayes classification, and greedy mutation. From our experiments, it is evident that the most effective search technique (in terms of the number of Vivado runs needed to close timing) varies across different designs. Intuitively, distinct designs often present vastly different structures of the search space. Besides, different phases of the design space exploration benefit from different search techniques. For example, stochastic methods such as genetic algorithm may be more useful during the initial phase of the search, while first-order optimizations like gradient descent are very efficient in finding local minima when the promising search space is narrowed.

The above observations clearly motivate the use of an ensemble of search heuristics rather than one particular technique to effectively explore the design space of FPGA compilation. Similar insights were also gained in the OpenTuner project, which aimed to provide an extensible open-source framework for software program autotuning [4]. OpenTuner currently incorporates a collection of search techniques to

FPGA'17, February 22-24, 2017, Monterey, CA, USA
 ISBN 978-1-4503-4354-1/17/02 . . . $15.00
DOI: http://dx.doi.org/10.1145/3020078.3021747

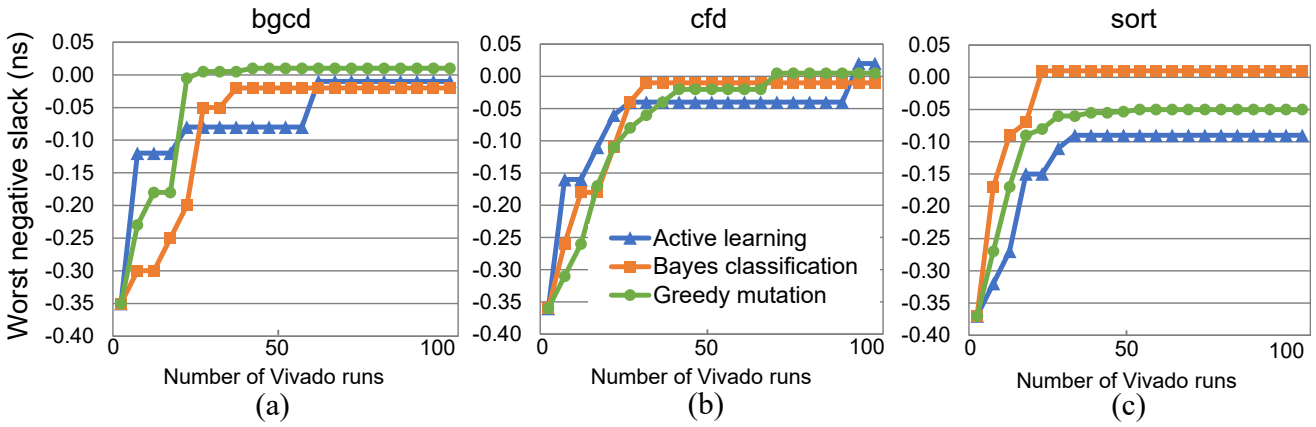

Figure 1: The search traces of three designs using three different search algorithms with the goal of improving worst negative slack — We use meta-heuristic algorithms to analyze results from Vivado, and guide the selection of Vivado configuration parameters. The x-axis denotes the number of Vivado runs. (a) `greedy mutation`, a simple genetic algorithm, is to first to close timing for binary GCD; (b) `active leaning`, a semi-supervised machine learning technique, is the first to close timing for computational fluid dynamics; (c) `Bayes classification`, the naïve Bayes classifiers, is the first to close timing for the bubble sort design.

provide robustness against different search spaces and uses the multi-armed bandit (MAB) algorithm [11] to determine the allocation of trials between the available techniques dynamically. In addition to applications in program autotuning [4], MAB has already been applied to many important optimization problems in various fields, such as artificial intelligence [19, 22] and operations research [9, 15].

Since FPGA CAD tools usually require long execution times (minutes to hours for real-world designs), it is crucial to significantly speed up the MAB-guided search without sacrificing the final QoR. An intuitive approach is launching multiple machines simultaneously, each conducting a MAB-guided search within the solution space independently. Alternatively, one can use a more efficient scheme that dynamically partitions the solution space into multiple partitions, and allocates additional computing resources to regions that are more likely to generate high-quality solutions.

In this paper, we propose *DATuner* — a parallel bandit-based framework for autotuning FPGA compilation. DATuner is built on OpenTuner but instead focuses on improving the productivity and quality of FPGA-targeted hardware designs. We also propose scalable and effective parallelization techniques based on dynamical solution space partitioning to speed up the convergence of DATuner. Our main contributions are as follows:

1. We adapt OpenTuner to tune the CAD tool parameters for FPGA compilation and demonstrate the effectiveness of the bandit-based approach in improving the design QoR.
2. We propose a scalable parallelization scheme which accomplishes the following: (1) efficiently partitions the global solution space into promising subspaces; (2) allocates compute resource among subspaces to balance the *exploration* of unknown subspaces and the *exploitation* of subspaces with known high-quality solutions.
3. Experiments with DATuner on academic and commercial FPGA CAD tools demonstrate very encouraging improvements in design quality across a variety of real-life benchmarks. We believe that our framework is also applicable to many other EDA problems.

The rest of the paper is organized as follows: Section 2 introduces the preliminaries that serve as the basics of this work; Section 3 discusses our proposed techniques; Section 4 presents the experimental results; Section 5 summarizes the related work, followed by conclusions in Section 6.

2. Preliminaries

In this section, we provide an overview of the MAB problem formulation, its usage in OpenTuner, as well as the basics of the FPGA compilation process.

2.1 Multi-Armed Bandit Approach

The MAB problem and its solutions are extensively studied in statistics and machine learning [11]. The classic problem is formulated as a game, where the player has a fixed number of actions to choose from (i.e., *arms*), each of which having a reward given by an unknown probability distribution with an unknown expected value. The game progresses in rounds, and in each round, the player chooses one action (i.e., *pull an arm*) and obtain a reward sampled from the corresponding distribution. The reward loosely captures the effectiveness of an arm, and crucially, its probability distribution is learned during the process of the game. The objective is to maximize the total payoff after all the rounds. An effective MAB algorithm must find a right balance between *exploitation* (choosing the known best arm to obtain the highest ex-

pected reward), and *exploration* (selecting an infrequently used arm to gain more information about its reward distribution). Choosing an infrequently used arm sacrifices short-term gain for the possibility of discovering greater payoff in the long run. Existing methods are usually randomized algorithms, which pick an action in each round based on the history of chosen actions and observed rewards so far [5, 6]. The quality metric of an MAB algorithm is *regret*, which is the ratio between optimal payoff (obtained by pulling the optimal arm every round) and that generated from the MAB algorithm. Several known MAB algorithms can achieve a regret of $O(log\, N)$ for an `N`-round MAB, which has been shown to be asymptotically optimal [8].

Recently, an open-source autotuning framework called OpenTuner has adopted MAB to improve the runtime of software benchmarks [4]. Specifically, OpenTuner incorporates an ensemble of meta-heuristics for effectively searching the space of software compiler options. Examples of these heuristic methods include differential evolution, genetic algorithm, particle swarm optimization, and simulated annealing. The MAB algorithm used in OpenTuner treats each search method as an arm, and measures its reward using the area under curve mechanism — If an arm has yielded a new global best, an upward line is drawn, otherwise, a flat line is drawn. The area under this curve (scaled to a maximum value of 1) is the total payoff attributed to the corresponding arm. To balance exploitation and exploration, OpenTuner ranks each arm with a weighted sum of the area under curve metric and the frequency of its previous uses. Notably, OpenTuner reported up to 2.8x speedup at no programming cost by automatically tuning GCC flags.

2.2 FPGA Compilation Flow

The mainstream FPGA compilation flow takes an RTL design as input and generates a device-specific bitstream. This process involves several distinct and modular steps: logic synthesis, technology mapping, placement, and routing. Synthesis lowers the RTL design into a technology-independent logic or gate-level network. Technology mapping then maps this network into a netlist of look-up tables (LUTs). Placement determines the physical location of each LUT in the netlist, and routing connects all signal paths using the available programmable interconnects.

Many of these steps involve NP-hard problems. To tackle the difficulty of solving these problems, experts propose some approximated solutions with heuristic-based methods. FPGA CAD tools often provide designers with a set of configuration parameters that select between heuristics or influence the behavior of a heuristic. Examples of parameters include enabling remapping or retiming in logic synthesis, deciding how much to spread logic for congestion or how much to weight wire delay. Table 1 shows the tunable parameters available in the open-source Verilog-to-Routing (VTR) toolflow [16], covering logic synthesis, packing, placement and routing. With commercial FPGA CAD tools from Altera/Intel and Xilinx, a much larger collection of switches are available (roughly 60 to 80 options are exposed to designers).

Table 1: List of tunable VTR configuration parameters.

Parameter	Value	Stage
resyn	{on,off}	logic synthesis
resyn2	{on,off}	logic synthesis
resyn3	{on,off}	logic synthesis
alpha_clustering	[0,1]	packing
beta_clustering	[0,1]	packing
allow_unrelated_cluster	{on,off}	packing
connection	{on,off}	packing
alpha_t	[0.5-0.9]	placement
seed	{1,2,3,4,5}	placement
inner_num	{1,10,100}	placement
timing_tradeoff	[0.3-0.7]	placement
inner_loop_recompute	{0,1,5}	placement
td_place_exp_first	{0,1,3}	placement
td_place_exp_last	{5,8,10}	placement
max_router_iterations	{20,50,80}	routing
initial_pres_fac	[0.3-100]	routing
pres_fac_mult	[1.2-2]	routing
acc_fac	[1-2]	routing
bb_factor	{1,3,5}	routing
astar_fac	[1-2]	routing
max_criticality	[0.8-1]	routing
criticality_exp	[0.8-1]	routing
base_cost_type	{'demand_only', 'delay_normalized'}	routing

2.3 Autotuning FPGA Compilation Parameters

Obviously, these tool options together create an enormous design space which cannot be effectively explored by human effort alone. Besides, there is no single set of compilation parameters that works for all designs [24]. Thus, we propose to automatically configure the FPGA tool options to achieve a faster design closure and better QoRs. The QoR metrics can be timing slack, resource usage, or power consumption. Due to the slow runtime of FPGA compilation, effective parallelization is paramount to ensure the viability of autotuning.

In this paper, we propose a parallel search methodology named DATuner. DATuner adopts OpenTuner as its core search engine to leverage the advantage of an ensemble over a single technique.

3. DATuner Techniques

In this section, we first propose a dynamic solution space partitioning method for parallelization. We also provide a MAB-based method for uneven computing resource allocation. Then we illustrate our paralellization framework.

3.1 Motivation

We formulate the EDA autotuning problem as a search problem in an N-dimensional solution space called S_0, where

each dimension can be either continuous or discrete. Obviously, S_0 grows exponentially with the dimensionality of the solution space. When N is large, it is impractical to exhaustively traverse the search space to find the optimal solution. A simple approach is to partition S_0 into subspaces, launch multiple parallel searches, and assign each search instance to explore within one subspace. However, since some subspaces are more promising than others in terms of yielding good solutions, it is vital to properly partition the solution space in a way that the majority of search instances are assigned to the most promising subspaces. Of course, just as there is no single set of tool parameters that works well for all designs, no static partitioning of the solution space is optimal across designs.

Finding a suitable partitioning of the search space is key to improving the quality of the search. But doing so manually is usually hard, and requires adequate domain knowledge. In addition, just as there is no single set of parameters that works well for all designs, no static partitioning of the parameter space is optimal across designs. Thus, we propose a novel parallelization method, where we gradually identify promising subspaces via dynamic partitioning based on QoR samples obtained at runtime. We further propose an MAB-based compute resource allocation method for uneven sampling — more compute resources are assigned to promising subspaces to increase sampling quality.

3.2 Dynamic Solution Space Partition

We propose to partition the solution space dynamically, where the partitioning does not rely on any prior knowledge; instead it is decided by posterior knowledge learned during runtime. More specifically, our partitioning method iteratively constructs a *space partitioning tree* (SP tree), where the root node of the tree represents the initial solution space, and each intermediate node represents a subspace. The leaf nodes of the SP tree collectively form the active partitioning that is currently used by the parallel search instances. In each iteration, we select a subspace and divide it into multiple smaller partitions. As a result, a leaf in the SP tree will become an intermediate node that branches out to multiple new children, with which each "newborn" representing a newly created subspace.

Given an N dimensional solution space, we propose to dynamically partition the search space into subspaces and allocate more computing resources for searching within promising subspaces. This dynamic partitioning process is illustrated in Figure 2. S_0 represents the initial solution space and $S_1, S_2, ..., S_n$ are the subspaces iteratively created during space partitioning process. Figure 2(a) shows the known samples that are explored, where each grey dot indicates a sample with a good QoR and red crosses are those with poor QoRs. At each step of the partitioning process, a key decision is to select the most profitable dimension out of the N dimensions to partition, such that we can gain as much information about the solution space as possible. Here we propose to examine the entropy and information gain [20] when partitioning a specific dimension. Specifically, for each dimension i in the N-dimensional search space, we compute conditional entropy assuming we partition along dimension i and derive the corresponding information gain. We select the dimension with the highest information gain to partition.

Formally, for a subspace S_i, we define the set of known samples within S_i as D_i. We further label each sample as good or bad based on its associated QoR, and use Dg_i and Db_i to denote subset of good and bad samples within D_i, respectively (note that $Dg_i \cup Db_i = D_i$). Then we define the entropy of D_i as

$$H(D_i) = -(\frac{|Dg_i|}{|D_i|}\log(\frac{|Dg_i|}{|D_i|}) + \frac{|Db_i|}{|D_i|}\log(\frac{|Db_i|}{|D_i|}))$$

where $|D_i|$, $|Dg_i|$, $|Db_i|$ are the cardinalities of sets D_i, Dg_i, and Db_i, respectively. We next define the conditional entropy of D_i conditioned on a specific dimension of S_i. Suppose d represents the parameter chosen for the dth dimension of S_i, and we assume d has k possible discrete values. If we further define $H(D_i|d = j)$ as the entropy of D_i conditioned on d taking the value of j, we will have

$$H(D_i|d) = \sum_{j=1}^{k} \frac{|D_{i,d=j}|}{|D_i|} H(D_i|d = j)$$

Here $|D_{i,d=j}|$ is the cardinality of the set of samples in S_i with d set to j. With the above notations, we can formally define the information gain along dimension d as $G(D_i, d) = H(D_i) - H(D_i|d)$. Our dynamic space partitioning algorithm creates a new subspace by partitioning along the dimension with the highest information gain. If the chosen dimension has k possible parameter values (continuous value will be discretized), it will get partitioned and become an intermediate node in SP tree with $(k-1)$ new children.

For the example in Figure 2(a), we have the following calculations when deciding whether to partition along x or y dimension. Here we assume have sampled 14 design points and eight of them have good QoRs. Therefore, the entropy of the initial solution space S_0 is $H(D_0) = -(\frac{8}{14}log(\frac{8}{14}) + \frac{6}{14}log(\frac{6}{14})) = 0.986$. We then compute the conditional entropy of D_0 conditioned on dimension x and y as $H(D_0|x) = \frac{8}{14} * 0.81 + \frac{6}{14} * 0.91 = 0.853$, and $H(D_0|y) = \frac{6}{14} * 1.00 + \frac{8}{14} * 0.96 = 0.977$. Finally, we compute the information gain for partitioning along x or y dimension as $G(D_0, x) = 0.133$ and $G(D_0, y) = 0.009$. Since the former information gain is higher, we decide to partition S_0 along the x dimension, which results in two new subspaces S_1 and S_2, as shown in Figure 2(b,c). In the next iteration of this process, we follow the same method and choose to further partition along the y dimension in S_2 as shown in Figure 2(d).

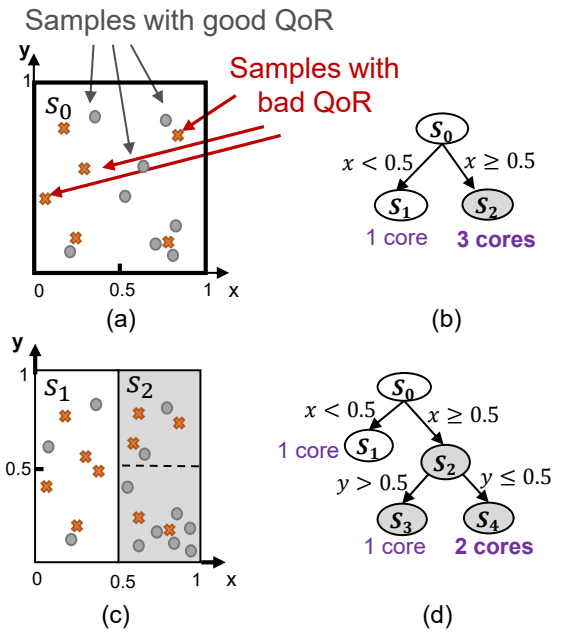

Figure 2: Conceptual illustration of the solution space partitioning method — The solution space is 2D with each dimension constrained to [0,1]. grey circles represent samples with good QoR and red crosses are poor samples. (a) and (c) show the partitions and the known samples at each iteration. (b) and (d) show the corresponding SP trees. Given four cores, we assign a different number of cores to each subspace based on the quality of solutions in that subspace. At each step, we choose the dimension with the highest information gain to partition.

3.3 MAB-directed Computing Resource Allocation

Once we obtain an SP tree, we need to properly allocate the available compute resources to the subspaces to maximize the overall sampling rewards. The key here is to balance exploitation (i.e., allocating more search instances to the most promising subspace) with exploration (i.e., sampling less promising subspaces to obtain a better estimate of enclosed solutions). Just as we can leverage MAB to choose the search technique when exploring within a subspace, we also propose to use it to solve the resource allocation problem. More concretely, we treat the subspaces as arms. To calculate the reward, we employ the UCBI algorithm [5], which defines the reward of a subspace S_i at round t as $\text{payoff}(S_i, t) = \overline{x_i} + \sqrt{\frac{2 \ln t}{n_i}}$. In this formulation, $\overline{x_i}$ is the average QoR of samples in S_i and n_i is the number of times (i.e., frequency) S_i has been chosen so far by the MAB. Essentially, the QoR and frequency terms are used to balance exploitation and exploration in the compute resource allocation.

3.4 Parallelization Framework

Figure 3 shows the overall flow of the proposed parallelization scheme, which follows a master-slave model. The master is responsible for distributing the parallel slave processes (search instances) to different subspaces. It starts with the original global space, and after collecting a sufficient number of samples, it performs dynamic space partitioning and allocates search instances to appropriate subspaces. A slave process invokes its search techniques to explore a specific subspace, and reports its result backs to the master. The master then further partition the search space and reallocates the slaves. This process iterates until attaining the target QoR or reaching timeout.

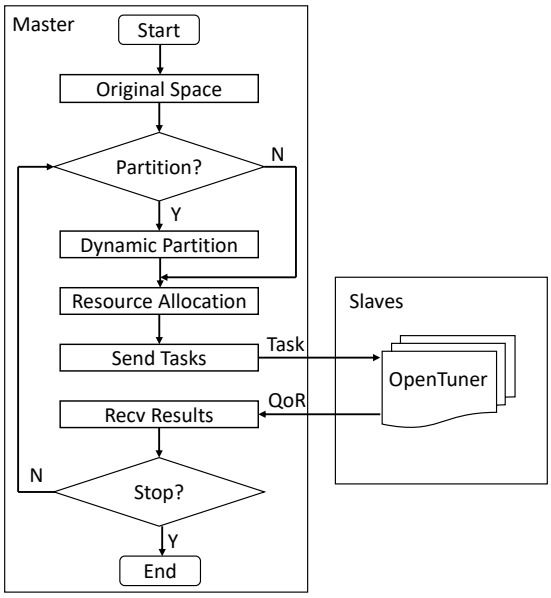

Figure 3: DATuner parallelization framework — We apply Master/Slave parallelization model and use MPI for communication. Master sends `Tasks` to slaves. `Task` specifies which subspace that slave instance should explore and may contain a historical best found QoR sample (if exists) in this subspace for slave instance to use as a search seed.

4. Experimental Results

In this section we evaluate the QoR improvement and convergence time of DATuner on two widely-used FPGA CAD tools: (1) the academic Verilog-to-Routing (VTR) framework [16] (version 7.0) and (2) Xilinx Vivado Design Suite [3] (version 2016.1). We run DATuner with eight machines (each machine with one search instance). Each machine has a quad-core Xeon processor running at 2.8GHz.

We make use of a set of real-life benchmarks from several different domains including image processing, general-purpose computing, communication, security, and computer vision. For the experiments targeting VTR, we choose 10 benchmarks covering different domains from the VTR-7.0

benchmark suite [16]. For the experiments using Vivado, we use five large industry designs. The characteristics of the VTR benchmarks and industry designs are summarized in Table 2 and Table 3, respectively.

Table 2: Profiles of the benchmarks used for VTR tuning.

Circuit	LUT	BRAM	Description
MkPktMerge	239	7344	Packet processing
Diffeq1	362	0	Scientific computing
Ch_intrinsics	425	256	Memory system
Raygentop	1884	5376	Ray tracing
MkSMAdapter4B	1960	4456	Packet processing
Sha	2001	0	Cryptography
Boundtop	3053	32768	Ray tracing
Or1200	3075	2048	RISC-V Processor
Blob_merge	8067	0	Image processing
Stereovision0	9567	0	Computer vision

Table 3: Profiles of the industry benchmarks used for Vivado tuning.

Circuit	FF	LUT	Constraint (ns)	Device
Design1	14545	14122	2.60	Virtex 7K160T
Design2	17847	29012	6.55	Virtex 7K70T
Design3	18204	28361	2.00	Virtex 7K160T
Design4	26098	17242	2.65	Virtex 7VX330T
Design5	27873	38261	4.30	Virtex 7VX330T

We select 23 tunable parameters from the VTR-7.0 manual [16] and 9 tunable parameters from Vivado flow covering logic synthesis, packing, placement and routing. The list of parameters for VTR and Vivado are shown in Table 1 and Table 4, respectively.

4.1 Tuning VTR

We first compare DATuner with a parallel baseline that performs a static partitioning scheme, denoted as *Static-Part* in Section 4.1.1. Then we compare DATuner with a serial baseline running OpenTuner on one machine, denoted as *Ser-MAB* in Section 4.1.2.

4.1.1 Comparison with Static Partitioning

For the case of static partitioning, we choose three parameters from the VTR parameter list as the pivots for partitioning and partition the solution space into eight subspaces. We empirically choose the partitioning pivots to be `alpha_t`, `allow_unrelated_clustering`, and `base_cost_type`, which usually have large impact on the timing quality.

In Figure 4, we compare DATuner with Static-Part, where both methods use eight machines for parallel searching. We conduct 100 iterations of searching for 10 VTR benchmarks and show the best-found frequency at different iterations. DATuner achieves better QoR as well as faster convergence

Table 4: List of tunable Vivado configuration parameters.

Parameter	Value	Stage
OptDirective	{Explore, ExploreSequentialArea, AddRemap, ExploreArea, Default}	logic synthesis
PlaceDirective	{Explore, ExtraNetDelay_high, ExtraNetDelay_medium, ExtraNetDelay_low, ExtraPostPlacementOpt, WLDrivenBlockPlacement, LateBlockPlacement, SSI_SpreadLogic_high, SSI_SpreadLogic_low, AltSpreadLogic_low, AltSpreadLogic_medium, AltSpreadLogic_high, ExtraTimingOpt, Default}	placement
Fanout_opt	{on,off}	post-placement
Placement_opt	{on,off}	post-placement
Critical_cell_opt	{on,off}	post-placement
Critical_pin_opt	{on,off}	post-placement
Retime	{on,off}	post-placement
Rewire	{on,off}	post-placement
RouteDirective	{Explore, HigherDelayCost, Default}	routing

rate than Static-Part for eight out of the ten designs. Besides, DATuner shows a significant improvement on average QoR than Static-Part, further demonstrating the benefits of dynamic partitioning. From our experiments, the SP trees learnt by DATuner at runtime through dynamic partitioning are indeed very different across different designs.

Table 5 provides a more detailed comparison between the best configurations found by DATuner and static partitioning in terms of the clock frequency, runtime, and area increase over the results from the default setting. It is not surprising to see frequency improvements from both dynamic and static partitioning, which indicates effectiveness of using MAB-guided search ensembles. In addition, DATuner with dynamic partitioning outperforms the static scheme in the majority of designs in terms of both frequency and LUT counts. The runtime overheads are also similar.

4.1.2 Comparison with Serial Search

We further compare DATuner with Ser-MAB in Figure 5. For fair comparison, we constrain DATuner and Ser-MAB with the same amount of compute efforts, where Ser-MAB runs on one machine for 800 iterations and DATuner runs eight search instances of 100 iterations each. While each MAB instance makes use of an ensemble of heuristic searches to avoid getting easily stuck in local optima, our

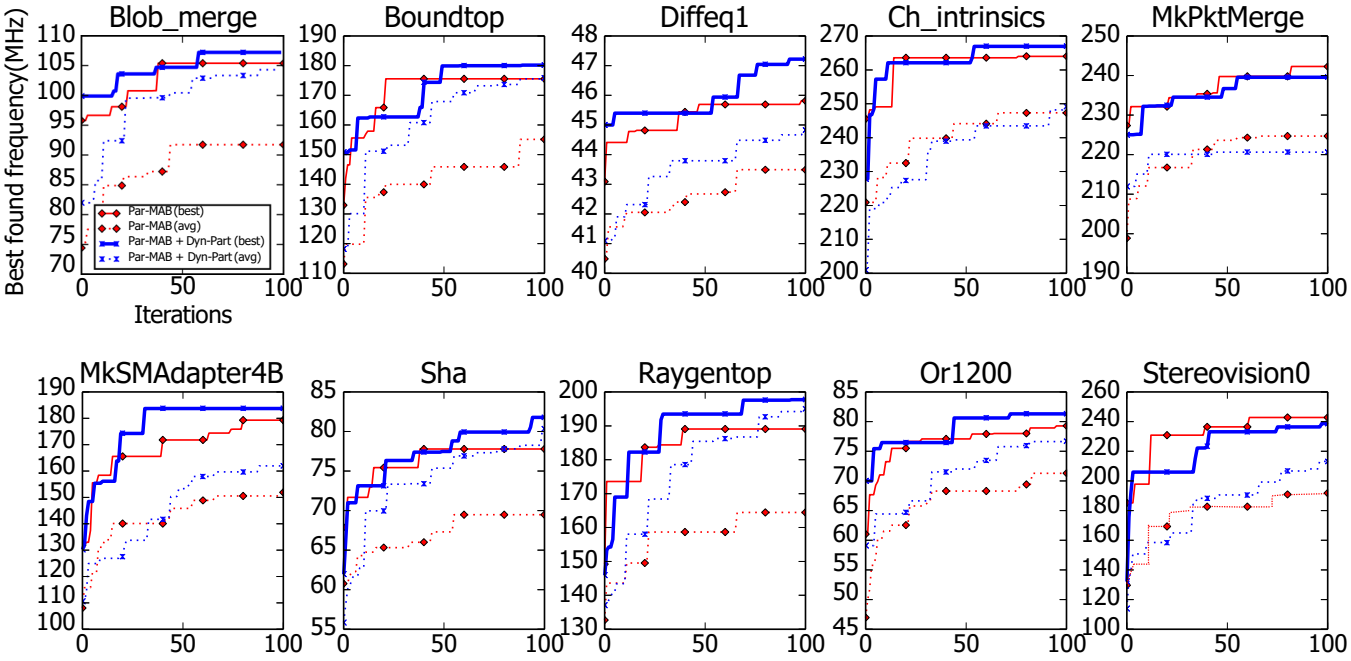

Figure 4: Comparison of dynamic and static partitioning schemes for VTR tuning — `Par-MAB` uses the Static-Part parallelization scheme and `Par-MAB+Dyn-Part` is the DATuner method. `best` is the best frequency found up to the current iteration and `avg` is the average frequency of all design points found up to the current iteration.

Table 5: Profiles of the best-found VTR configurations by DATuner and static partitioning — Freq., RT, and LUT are the ratios of the best configuration over default on clock frequency, runtime, and LUT count, respectively.

Circuit	DATuner			Static-Part		
	Freq. ratio	RT ratio	LUT ratio	Freq. ratio	RT ratio	LUT ratio
Blob_merge	1.11	5.22	1.01	1.09	2.54	1.01
Boundtop	1.17	3.85	1.08	1.14	3.21	1.06
Diffeq1	1.16	1.16	1.06	1.13	1.23	1.36
Ch_intrinsics	1.06	3.51	0.97	1.06	4.59	1.73
MkPktMerge	1.09	0.89	1.20	1.14	0.68	1.00
MkSMAdapter4B	1.04	1.86	0.99	1.01	0.69	1.00
Sha	1.12	2.28	1.00	1.06	1.69	1.00
Raygentop	1.12	3.62	0.99	1.07	2.09	1.42
Or1200	1.08	0.95	1.00	1.06	2.18	1.12
Stereovision0	1.18	1.74	1.29	1.21	2.04	1.24
Avg.	1.11	2.51	1.06	1.09	2.10	1.19

parallelization scheme further enables multiple MAB search instances to explore additional promising regions that cannot be quickly reached by a single search. In Figure 6, we attempt to visualize the search trajectory of one VTR design through dimensionality reduction. Here the black dash line captures the search trajectory of Ser-MAB, which is mostly trapped in one region. Other colored lines represent search trajectories of different search instances instantiated by DATuner, where they are simultaneously exploring different promising regions of the design space. We believe that this partly explains the results in Figure 5 where Ser-MAB results in a worse QoR than that from DATuner in six out of the ten VTR designs.

In Figure 7, we further compare DATuner with Ser-MAB in terms of the runtime to achieve a specific QoR target. We set the QoR target to be 2% improvement in design frequency over the default design, and measure the speedup in runtime of DATuner (with eight search instances) over Ser-MAB. DATuner reaches the target QoR 11X faster than Ser-MAB.

4.2 Tuning Vivado

We have also applied DATuner to resolve the timing closure problem for five large industry designs, which are listed in Table 3. These designs are specified with very tight timing constraint, and are known as challenging benchmarks in terms of meeting timing.

In addition to showing the timing improvement over the default settings of Vivado, we also experiment with the Vivado exploration mode, a tool option that explores various

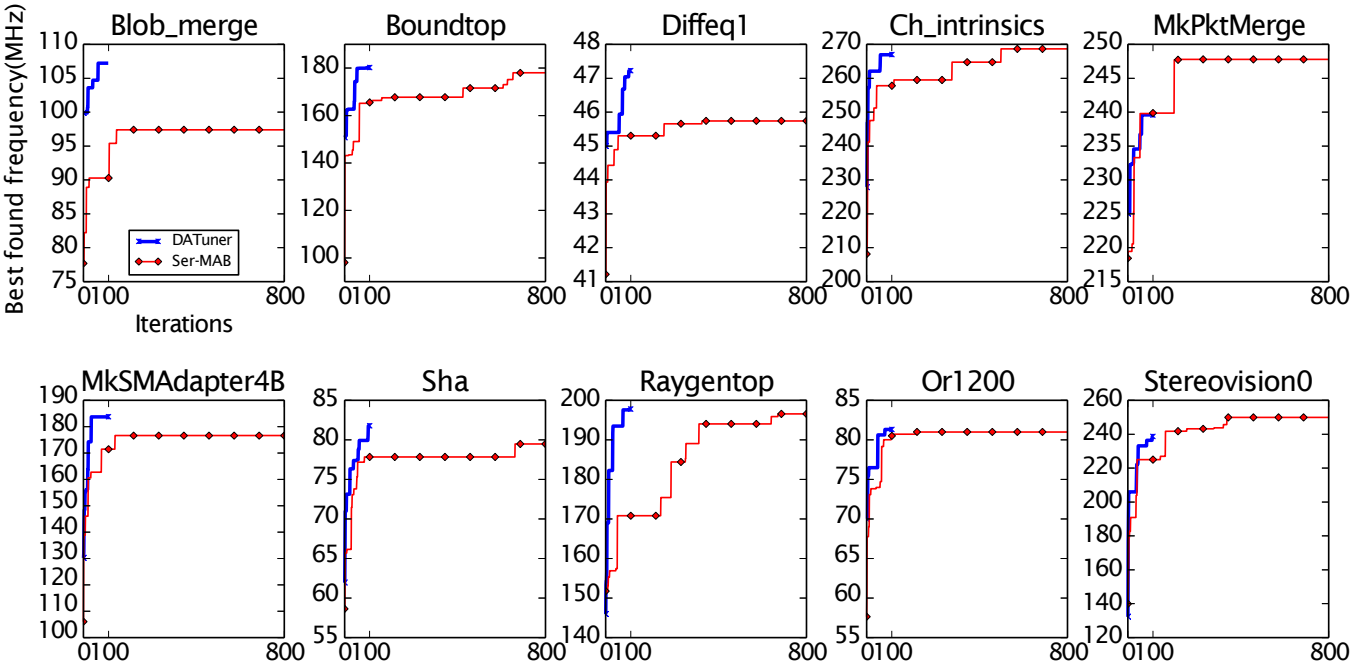

Figure 5: Comparison of DATuner and sequential search (Ser-MAB) for VTR tuning — We use the same total number of search iterations for DATuner and Ser-MAB.

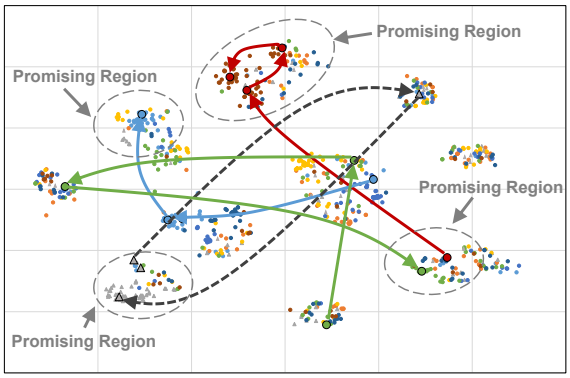

Figure 6: Search trajectory of VTR tuning for Diffeq1 — We use t-SNE package [23] to reduce dimensionality of the search space for VTR design Diffeq1 from 23 to 2 for the sake of visualization. Different colors represent samples of different search engines. Arrows indicate search time sequence. The black dash line represents trajectory of Ser-MAB. Other colored lines capture traces of DATuner. Ser-MAB is stuck in one promising region while DATuner leads to multiple promising regions.

optimizations in the placement and routing stages for improving timing. Figure 8 shows that this mode improves the WNS for four designs, but still fail to meet the timing constraint. In contrast, DATuner helps close timing for all designs.

We also study the profiles of the best-found configurations by DATuner. Figure 9 shows that average similarity

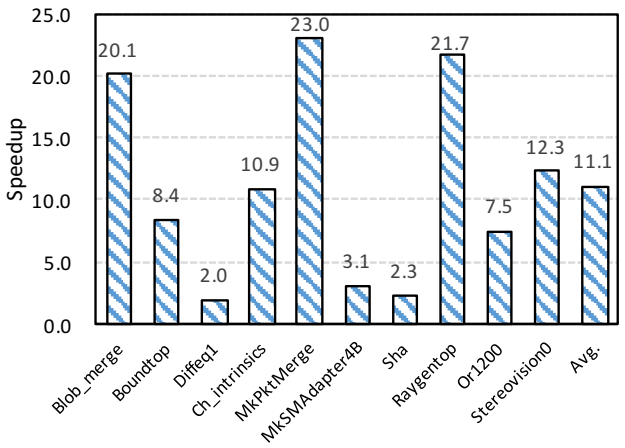

Figure 7: Runtime improvement using DATuner for VTR tuning — Compared with Ser-Search, DATuner reaches the same frequency target 11.1X faster on average using eight machines.

among the five designs is only 46%, indicating that the one-size-fits-all solution indeed does not exist. Table 6 further shows the ratios of worst negative slack (WNS), runtime, and area between the best-found configurations and default. On average, we reduce WNS by 11% and increase the runtime by a factor of 3.7X. The resource utilization is almost the same as the default configuration.

5. Related Work

Intelligent design space exploration and autotuning-based techniques have been proposed in the domain of high-

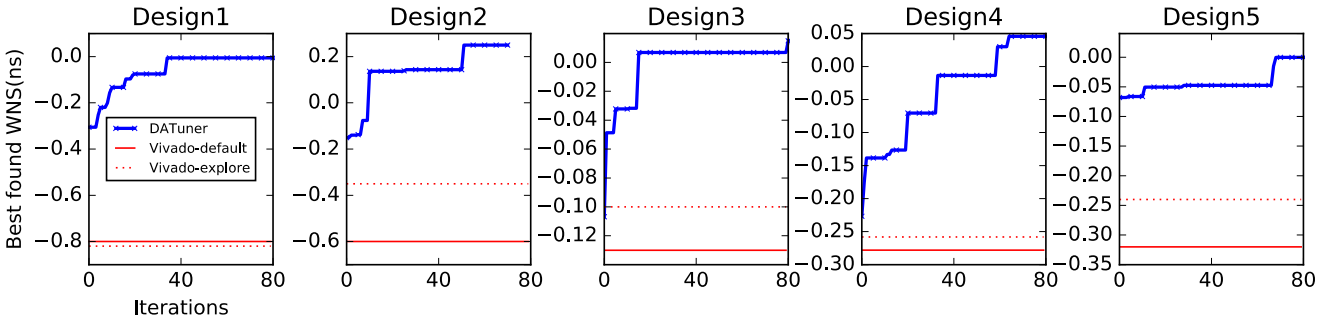

Figure 8: Vivado tuning by DATuner for large-scale industry designs — These designs fail to meet timing with both `Vivado-default` and `Vivado-explore` modes, whereas DATuner closes timing for all of them.

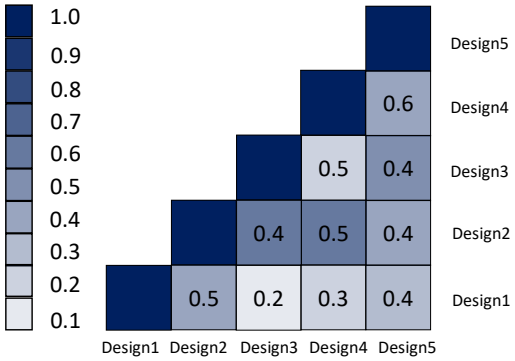

Figure 9: Similarity matrix of best-found configurations for Vivado designs — the average similarity is 46%.

Table 6: Profiles of the best-found Vivado configurations by DATuner — WNS, RT, LUT, and FF are the ratios of the best configuration over the default one in terms of worst negative slack, runtime, LUT count, and flip-flop count, respectively.

Circuit	WNS ratio	RT ratio	LUT ratio	FF ratio
Design1	0.77	2.38	0.83	1.00
Design2	0.93	6.80	1.00	1.00
Design3	0.94	3.26	1.00	1.00
Design4	0.88	4.30	1.03	1.00
Design5	0.90	1.62	1.00	1.00
Avg.	0.89	3.67	0.97	1.00

performance computing (e.g., stencil computation [10] and matrix computation [21]) and compiler optimization (e.g., a domain-specific compiler for image processing applications [18] and the compiler for Java Virtual Machine [12]).

Similar research efforts have recently emerged in the EDA field to configure CAD tool options to improve design quality automatically. Xilinx SmartXplorer [1] and Altera Design Space Explorer [2] use predefined or user-specified configuration bundles for design space exploration. They support parallel CAD runs with fixed sets of parameters and automatically report the best solution found. Unlike the fixed heuristics used in these tools, DATuner dynamically determines the best search method for the current design.

InTime [13, 14] is a commercial autotuning tool for FPGA timing closure based on the naïve Bayesian classifier, and has recently been extended to include other machine learning techniques as well [24]. InTime builds a database of configurations from a series of preliminary runs and learns to predict the next set of CAD tool options to improve timing results, achieving 30% improvement in timing result compared to vendor-supplied design space exploration tools. The authors in [17] also propose machine learning techniques such as linear regression and random forest to autotune the performance and power consumption of FPGA designs. We note that the learning-based sampling and classification techniques used in InTime [14] and [17] are complementary to our proposal. It is possible to integrate these methods into DATuner as an additional arm in the MAB algorithm. Besides timing closure, our framework can also be applied to other EDA tools.

OpenTuner [4] leverages the MAB method to dynamically select the best searching technique from an ensemble of searching strategies to tune global compiler switches, which improves the performance of software compilers such as GCC by to 2.8x over the baseline using `gcc -o3`. Our work builds on OpenTuner, and targets hardware synthesis rather than software compiler optimizations, and investigates parallelization techniques to speed up the tuning process.

6. Conclusions

In this paper we propose DATuner, a parallel autotuning framework for FPGA compilation using the multi-arm bandit technique. To mitigate the high runtime cost incurred by the complex CAD optimization process, we devise an efficient parallelization scheme that enables many MAB-based autotuners to explore the design space simultaneously. Concretely, DATuner dynamically partitions solution space into promising subspaces based on information gains, and allocates compute resource among subspaces to balance the exploration of unknown subspaces and the exploitation of subspaces with known high-quality solutions. Applications

of DATuner on VTR and Xilinx Vivado tools have demonstrated promising improvements in quality and convergence rate across a variety of academic and industry designs.

7. Acknowledgements

This research was supported in part by DARPA Young Faculty Award D15AP00096, Intel Corporation under the ISRA Program, a research gift from Xilinx, Inc., National Natural Science Foundation of China (NSFC) Grant 61520106004, Beijing Natural Science Foundation (BJNSF) Grant 4142022 and Chinese Scholarship Council.

References

[1] SmartXplorer for ISE Project Navigator Users Tutorial. *Xilinx Inc.*, 2010.

[2] Quartus II Handbook Volume 1: Design and Synthesis. *Altera Corporation*, 2014.

[3] Vivado Design Suite User Guide. *Xilinx Inc.*, 2015.

[4] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe. OpenTuner: An Extensible Framework for Program Autotuning. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2014.

[5] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, 2002.

[6] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire. The Nonstochastic Multiarmed Bandit Problem. *SIAM Journal on Computing*, 2002.

[7] D. Chen, J. Cong, and P. Pan. FPGA Design Automation: A Survey. *Foundations and Trends in Electronic Design Automation*, 1(3):139–169, 2006.

[8] W. Chen, Y. Wang, and Y. Yuan. Combinatorial Multi-Armed Bandit: General Framework and Applications. *Intl Conf. on Machine Learning(ICML)*, 2013.

[9] V. F. Farias and R. Madan. The Irrevocable Multiarmed Bandit Problem. *Operations Research*, 59:383, 2011.

[10] J. D. Garvey and T. S. Abdelrahman. Automatic Performance Tuning of Stencil Computations on GPUs. *Int'l Conf. on Parallel Processing (ICPP)*, pages 300–309, Sept 2015.

[11] J. Gittins, K. Glazebrook, and R. Weber. Multi-Armed Bandit Allocation Indices. 2011.

[12] S. Jayasena, M. Fernando, T. Rusira, C. Perera, and C. Philips. Auto-Tuning the Java Virtual Machine. *Parallel and Distributed Processing Symposium Workshop (IPDPSW)*, pages 1261–1270, 2015.

[13] N. Kapre, B. Chandrashekaran, H. Ng, and K. Teo. Driving Timing Convergence of FPGA Designs through Machine Learning and Cloud Computing. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2015.

[14] N. Kapre, H. Ng, K. Teo, and J. Naude. InTime: A Machine Learning Approach for Efficient Selection of FPGA CAD Tool Parameters. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2015.

[15] L. Lai, H. Jiang, and H. V. Poor. Medium Access in Cognitive Radio Networks: A Competitive Multi-Armed Bandit Framework. *Asilomar Conf. on Signals, Systems and Computers*, 2008.

[16] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed, K. B. Kent, J. Anderson, J. Rose, and V. Betz. VTR 7.0: Next Generation Architecture and CAD System for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, June 2014.

[17] A. Mametjanov, P. Balaprakash, C. Choudary, P. D. Hovland, S. M. Wild, and G. Sabin. Autotuning FPGA Design Parameters for Performance and Power. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, pages 84–91, 2015.

[18] R. T. Mullapudi, V. Vasista, and U. Bondhugula. PolyMage: Automatic Optimization for Image Processing Pipelines. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 429–443, 2015.

[19] S. Ontanón. The Combinatorial Multi-Armed Bandit Problem and its Application to Real-Time Strategy Games. *Artificial Intelligence and Interactive Digital Entertainment Conference*, 2013.

[20] C. E. Shannon. A Mathematical Theory of Communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 2001.

[21] O. Spillinger, D. Eliahu, A. Fox, and J. Demmel. Matrix Multiplication Algorithm Selection with Support Vector Machines. 2015.

[22] L. Tran-Thanh, S. Stein, A. Rogers, and N. R. Jennings. Efficient Crowdsourcing of Unknown Experts using Bounded Multi-Armed Bandits. *Artificial Intelligence*, 214:89–111, 2014.

[23] L. Van der Maaten and G. Hinton. Visualizing Data Using t-SNE. *Journal of Machine Learning Research*, 2008.

[24] Q. Yanghua, N. Kapre, H. Ng, and K. Teo. Improving Classification Accuracy of a Machine Learning Approach for FPGA Timing Closure. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2016.