Nautilus: Fast Automated IP Design Space Search Using Guided Genetic Algorithms

Michael K. Papamichael
Carnegie Mellon University

Peter Milder
Stony Brook University

James C. Hoe
Carnegie Mellon University

ABSTRACT

Today's offerings of parameterized hardware IP generators permit very high degrees of performance and implementation customization. Nevertheless, it is ultimately still left to the IP users to set IP parameters to achieve the desired tuning effects. For the average IP user, the knowledge and effort required to navigate a complex IP's design space can significantly offset the productivity gain from using the IP. This paper presents an approach that builds into an IP generator an extended genetic algorithm (GA) to perform automatic IP parameter tuning. In particular, we propose extensions that allow IP authors to embed pertinent designer knowledge to improve GA performance. In the context of several IP generators, our evaluations show that (1) GA is an effective solution to this problem and (2) our modified IP author guided GA can reach the same quality of results up to an order of magnitude faster compared to the basic GA.

1. INTRODUCTION

The use of IPs has become an indispensable part of modern hardware design flows. Instead of designing every component in a chip from scratch, designers can build entire chips or portions thereof by leveraging existing IP blocks, often developed by third parties. This practice greatly reduces the development time and cost of individual submodules within a larger chip. Over the years, IP blocks, which started as basic primitives (adders and multipliers), have now grown to complex IP blocks and even sophisticated on-demand design generators (e.g., [16, 14, 11]). A single IP block could be responsible for multiple millions of transistors in a chip.

To maintain performance and energy efficiency, today's IP blocks have to be highly parameterized to allow tailoring an instantiation to match specific application and user requirements. The flexibility of user customization however leads to the formation of a vast complex design space that has to be navigated by the IP user. The sheer number and details of the parameters are burdens to handle and a source for error. Moreover, many low-level module-specific parameters are cryptic and incomprehensible to an average IP user who is not already deeply familiar with the specific domain pertaining to the IP (e.g., signal processing, arithmetic units, on-chip interconnects). All of the above result in suboptimal outcomes from an otherwise more-than-capable IP generator.

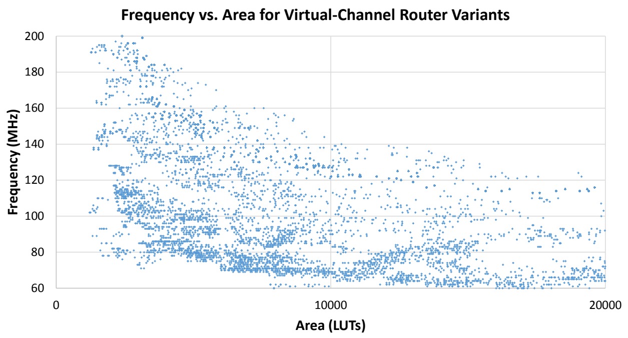

Figure 1: LUT usage and maximum frequency for approximately 30,000 router design points based on FPGA synthesis results.

The Scale of the Problem. Consider the top-level router module of the Stanford Open Source Network-on-Chip Router project [4], a highly-parameterized state-of-the-art router IP block, which exposes 42 parameters (not including any additional sub-module parameters). The design space of a single router already spans multiple billions of possible design points; this does not even consider the countless ways these routers can be arranged and connected to form a network. To give a sense of what this design space can look like, Figure 1 plots FPGA LUT usage and maximum frequency across approximately 30,000 design points—all interchangeable at a functional-level from an IP user's perspective—that belong to a subset of the full design space formed by only 12 out of the 48 parameters.

As another example, we used the publicly available CONNECT NoC IP generator [13] to generate a large collection of different network configurations (router design + network topology) targeting a commercial 65nm technology. Figure 2 plots how power and area relate to peak network bisection bandwidth (a network performance metric), across a variety of different 64-endpoint NoC configurations (different colors represent different topology families). Note that, once again, all of these design instances are only a small subset out of the myriad of potential NoC configurations, which are all

DAC'15, June 07 - 11, 2015, San Francisco, CA, USA
Copyright 2015 ACM 978-1-4503-3520-1/15/06 ...$15.00.
http://dx.doi.org/10.1145/2744769.2744875

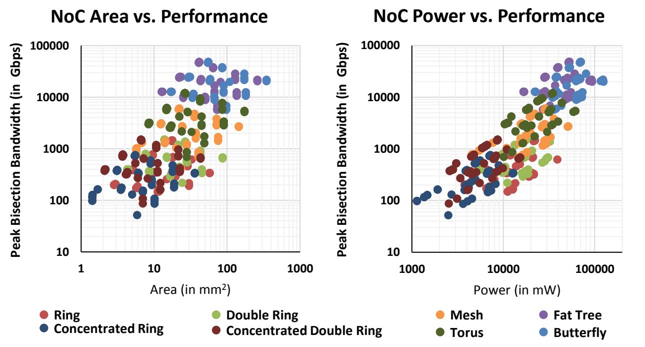

Figure 2: Area, power and performance for various 64-endpoint CONNECT NoCs targeting a commercial 65nm ASIC technology node.

interchangeable from an IP application perspective; an IP user could pick any of these to satisfy the functional-level connectivity requirements of his or her application. The fact that these design points exhibit 2–3 orders of magnitude of variation across all presented metrics (power, area, performance), highlights the need to be able to efficiently and quickly navigate the design space and the criticality of picking a design point that makes the right trade off and fits the constraints of the project.

Our Solution: Nautilus. The sheer scale and vastness of design spaces of parameterized IPs, such as the ones presented above, and the fact that evaluating each design point can be very costly (requiring long runs of CAD and/or simulation tools, each of which can take several minutes to hours) makes exhaustive search prohibitive and motivates the need for automated fast and efficient navigation of the design space. To this end, we present Nautilus, an IP author guided design space exploration engine. This automatic design tuning approach is especially fitting in the context of parameterized IP generators which are already software-driven active objects.

At the core of Nautilus is a modified genetic algorithm (GA) that allows embedding of IP author knowledge pertaining to the IP design space. This knowledge, coming in the form of "hints," captures the IP author's intuition about how IP parameters relate to the various metrics of interest; the goal is to help steer the optimization search process more quickly toward profitable directions. In this paper, we offer a taxonomy of key classes of IP author knowledge and discuss how to incorporate them into GAs. A particularly important issue in embedding IP author guidance into a GA is to balance the strength of the author's guidance (which will be imperfect) and the stochastic nature of the underlying GA, which is critical for overcoming local optima and for handling design regions that may defy the author's intuition.

We present an evaluation of the Nautilus approach in the context of two hardware IP generators, targeting Networks-on-Chip and fast Fourier transforms (FFTs). The results show that GAs are effective in the automatic tuning of IP parameters when optimizing for a number of key design metrics (e.g., resource usage, efficiency), reducing the number of design points that have to be evaluated by orders of magnitude compared to a naive brute force approach. The results further show that the Nautilus guided GA, with the help of IP author knowledge, can further accelerate the GA search process, reaching the same quality of results as a baseline GA with up to an order of magnitude fewer design points evaluated. This is significant in light of the fact that each evaluation typically corresponds to minutes to hours of EDA execution time depending on the complexity of the IP.

Paper outline. The rest of this paper is organized as follows. Section 2 provides background on genetic algorithms (GAs) and describes how a baseline GA can be used to perform IP optimization. Section 3 introduces Nautilus and describes our extensions to GA that incorporates author hints. Section 4 reports our evaluation methodology and experimental results. Finally, we discuss related work in Section 5 and provide a discussion and conclusions in Section 6.

2. BACKGROUND: GENETIC ALGORITHMS

Genetic algorithms (GAs) are a class of stochastic adaptive optimization algorithms based on the ideas of evolution, natural selection and genetics (e.g., [3]). An initial "population," which consists of randomly selected points in the solution space, is allowed to evolve over a number of generations. During each generation, samples of the population can mutate or combine with each other (crossover) and at the end of each generation, the samples are evaluated and assigned a fitness score. The most "fit" samples resulting from this process form the next generation.

Applying GA to solve a new optimization problem consists of three main steps. The first step is defining the genetic representation, i.e., expressing each possible solution in the solution space as series of genes, called a genome. Once this mapping has been established, the second step is to define and implement the genetic operators, such as mutation and crossover, that can be applied to the population. Mutation operations modify genes of individual members of the population, while crossover operations combine subsets of the genes of two existing members of the population to produce a new member (i.e., "breeding"). Finally, the third step is defining a fitness (a.k.a. objective) function that assigns a fitness score to each sample in the population. This fitness score is used during the ranking and selection process of the GA that determines which members of the population will survive to form the next generation, where they will have a chance to further mutate and breed.

The quality of results and runtime of a GA algorithm depends on several factors, including:

- **Population size.** A large initial population increases the solution space coverage, but also increases the amount of work that the algorithm performs during each generation. Since successive generations depend on each other, the population size effectively caps the available parallelism during the evaluation phase of the algorithm that calculates the fitness scores.

- **Mutation rate.** The mutation rate controls the probability of mutations. A low mutation rate restricts the algorithm to localized search around existing solutions (exploitation), while a high mutation rate allows the algorithm to make larger leaps and potentially overcome local optima to reach unexplored portions of the design space (exploration). As is also the case with other stochastic optimization algorithms, striking a good balance between exploration and exploitation is an important aspect of tuning a GA to a particular problem.

- **Fitness function.** The fitness function is used to assign a score to each sample in the population which is used at the end of each generation for ranking the available samples and selecting the ones that will make it on to the next generation. The fitness function is used to guide the evolution process and is one of the most central elements of a GA. Not only is it used to pick different optimization goals, but it can also be adapted to constrain the algorithm to only explore specific portions of the solution space (e.g., by assigning very low scores to solutions lying in regions of the design space that are not of interest or should be avoided).

GAs for IP optimization. In this work we use genetic algorithms to automatically tune IP parameters for a given optimization goal. In the context of IP optimization, the initial population consists of potential design instances with different low-level parameter configurations, each corresponding to a distinct point in the design space. Mutations correspond to changing a parameter value, and crossovers combine parameter settings of different samples in the design space. Depending on the type of IP and the metric being optimized, "fitness" can take many different forms and even incorporate multiple metrics, which allows for great flexibility. For instance, in the case of a Network-on-Chip router, fitness can correspond to FPGA resource usage, throughput, energy efficiency or even a custom-defined composite function that can combine these metrics in arbitrary ways.

Evaluating GAs. We are mainly interested in two metrics when evaluating a GA: 1) runtime, i.e., how long it takes for the GA algorithm to run, and 2) quality of results, i.e., how good of a solution the GA finds. In the context of IP parameter optimization, runtime is directly tied to the number of fitness function evaluations, since each evaluation requires running computationally expensive CAD tools (e.g., FPGA/ASIC synthesis or place-and-route tools) and/or simulations. The quality of results can be either expressed as the fitness score of the best solution in the population or as a percentage with respect to the best scoring sample for the specific optimization (if that is known). It is important to understand the goal of Nautilus is not to find the absolute best design point. In real usage scenarios, we want to help an average IP user find a good design point that is within some threshold of what the IP generator can offer. The goal is to do much better than what an average IP user could do by trial-and-error or, worse yet, by taking the default. Thus for evaluation, we examine how many distinct design points have to be evaluated (at the cost of up to hours each) in order to reach a desired quality of results.

3. INCORPORATING AUTHOR KNOWLEDGE

Overview. Traditional GAs, such as the baseline GA described above, explore the design space in a random fashion, assuming no knowledge of how individual genes or parameters relate to optimization goals or affect the fitness of the samples in the population; each point in the design space is equally likely to be visited. This oblivious nature of the baseline GA makes it ideal for exploring unknown or highly unpredictable non-convex solution spaces. Compared to naive brute force design space exploration approaches, such as exhaustive search or random sampling, using a GA already marks a significant improvement in terms of the number of design points that have to be evaluated until a desirable one is found.

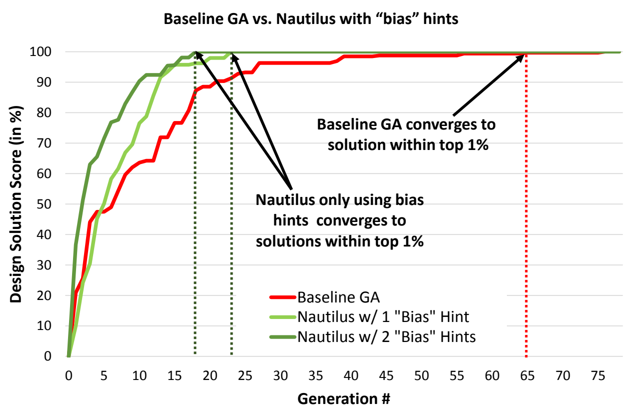

Figure 3: Baseline GA vs. Nautilus only using 1 or 2 "bias" hints.

The baseline GA might be a good approach when an IP user is dealing with an IP he or she is unfamiliar with. However, it is wasteful to forgo the wealth of knowledge the *author* of the IP possesses about the design space. In Nautilus, we want the IP authors to embed knowledge about the design space as an integral act of creating an IP. This IP author knowledge can drastically improve the GA search and optimization process, even if this knowledge is limited and comes in the form of partial or approximate hints.

To implement Nautilus we modified an existing open-source GA implementation, called PyEvolve [1], and extended it to support various forms of IP author or domain-expert hints pertaining to all or a subset of IP parameters. The idea behind these hints is to skew the search process towards specific regions of the design space.[1]

To illustrate the effect of hints, Figure 3 compares how the "fitness" (average of 20 runs) of a set of FFT hardware designs evolves over time using a baseline GA and Nautilus using a varying number of hints. In this example the baseline GA takes 56 generations to find a solution within the top 1%, while Nautilus can reach the same quality of results within 15 to 23 generations, depending on how many hints are provided.

Supported Classes of Hints. Below we describe some of the supported classes of Nautilus hints and explain their effect with respect to the baseline GA behavior and how they are used to guide the search process. It is important to keep in mind that the goal of Nautilus is to provide infrastructural support for different classes of hints; the exact instances are specific to the given IP generator and metric. The IP author's specialized knowledge allows him/her to provide hints as a part of creating an IP. The IP author is free to supply as many or few hints as desired; if it lacks sufficient hint information, Nautilus will fall back to using default values or employ the baseline GA. Unless specified otherwise, each hint needs to be supplied per metric of interest and per IP parameter, i.e., the IP author's knowledge about how IP parameters relate to high-level metrics is captured by a vector of hints, such as the ones described here.

[1]Note that these hints are incorporated in a probabilistic manner, maintaining the stochastic nature of GA, which is still free to explore the full design space and overcome local optima.

- **Importance:** The importance hint assigns values from 1 to 100 to each parameter that captures how drastically the parameter is expected to affect the metric being optimized. In Nautilus, the parameters' relative importance skews which genes are more likely to be picked for mutation during a genetic operation. This accelerates the algorithm, because it can directly focus on the parameters most likely to matter, wasting less time experimenting with others.

- **Importance Decay:** The "importance decay" hint takes a value from 0 to 1 and allows Nautilus to gradually adjust the relative importance differences of parameters. The idea is to allow for the importance of some parameters to "decay" over time (at the rate defined by the "importance decay" hint) as the algorithm progresses through generations.

  This hint can be used to introduce a *temporal aspect* to Nautilus, allowing it to initially focus on parameters believed to be "important" to coarsely navigate towards promising regions of the design space and then gradually shift focus to experimenting with less "important" parameters to perform more localized fine-tuning within those promising regions.

- **Bias and Target:** While the two previous hints affect which genes get selected to mutate, the bias and target hints affect the values that these genes will be assigned during each genetic operation. Each parameter can either be assigned a bias hint or a target hint (but not both). Bias takes values from -1 to 1 for each parameter and captures the correlation between the parameter and the metric being optimized. A positive or negative bias means that increasing the parameter will increase or decrease the metric being optimized, respectively. The target is a more direct hint that allows the IP author to specify that good (or balanced) solutions are known to cluster around a particular value. Bias and target can be used to guide Nautilus in a coarse manner, e.g., towards a specific direction, or in a more fine grained manner, i.e., towards a specific region in the design space. The bias and target hints cause Nautilus to behave in a much more "directed" fashion compared to a baseline GA; when correctly set by an expert who understands the design space, they can allow the algorithm to find efficient regions of the design space quickly.

- **Confidence:** The confidence hint can be viewed as a high-level knob that controls how much trust Nautilus should place in the author hints. In other words, this determines how "guided" the algorithm will be. Setting low confidence values will make the algorithm behave more similarly to the baseline GA, while setting high confidence values along with strongly-guided hints (e.g., setting target values of high bias values) will cause the algorithm to perform very directed optimization that starts to resemble convex optimization methods such as gradient descent. Confidence allows Nautilus to more easily incorporate heuristics or even low-confidence experimental hints that might be purely based on "gut feeling" without breaking the search or optimization process.

Finally, in addition to the hints described above, Nautilus includes some additional low-level auxiliary settings that, e.g., determine the "stepping" of the algorithm or define ordering relationships among values that a specific parameter can take (e.g., order different allocator options with respect to clock frequency or area). These settings control subtle aspects of the algorithm or are used to ensure smooth operation of the algorithm under non-trivial design spaces (e.g., sparsely populated design spaces that included infeasible points or regions).

In our targeted usage scenario, these hints are calibrated by the IP author during the IP development phase and are packaged and provided along with Nautilus as part of the IP (preferably in the form of an IP generator). However, in the absence of an IP "expert" these hints can also be set directly by a knowledgeable IP user. Additionally, an IP user could try sweeping each IP parameter independently and then observe how the various metrics of interest respond to estimate approximate hint values.

4. EVALUATION

We evaluate Nautilus with varying degrees of guidance against a baseline GA using a publicly available [4] highly-parameterized state-of-the-art Virtual-Channel Network-on-Chip (NoC) router IP, as well as the Spiral FFT IP design generator; for the remainder of this section we will refer to these IP as "NoC" and "FFT".

4.1 Methodology

As a preparatory step, we map a large portion of each IP's design space consisting of comparable—from an IP user perspective—design instances. The resulting datasets consist of approximately 12,000 design instances for the FFT IP (varying 6 parameters) and 30,000 design instances for the router IP (varying 9 parameters). For each design we run FPGA synthesis and/or simulations for each design instance to characterize it with respect to hardware implementation metrics (e.g., area, frequency), metrics specific to the IP domain (e.g., SNR values for the FFT IP), and composite metrics (e.g., throughput-per-LUT). FPGA synthesis results were obtained using Xilinx XST 14.7 targeting a Xilinx Virtex-6 LX760T FPGA (part xc6vlx760). This characterization step was done "offline" (using a dedicated cluster with 200+ cores running non-stop for about 2 weeks) to produce the datasets that we used to evaluate Nautilus.

Both Nautilus and our baseline GA implementation are based on modified versions of the PyEvolve genetic algorithm framework [1]. For each IP we define queries (e.g., optimize for throughput/area) and then compare the baseline GA with Nautilus in terms of the computational cost to run the query and quality of results (with respect to the given query). Unless otherwise noted, for both the baseline GA and for Nautilus, we use an initial population of 10 samples, a mutation rate of 0.1 (this means that each gene that belongs to a sample has a 10% chance of mutating during each generation), and run for 80 generations. Results are averaged over 40 runs for each experiment to compensate for the "noisy" nature of the stochastic process.

In the case of FFT, the Nautilus engine is expert-guided as the hints are provided from a member of the Spiral development team. For the NoC IP we estimated hints by synthesizing 80 designs (less than 0.3% of the design space) and observing trends; this is equivalent to an IP user (or some other non-expert) supplying the hints using limited empirical knowledge or gut intuition about the IP.

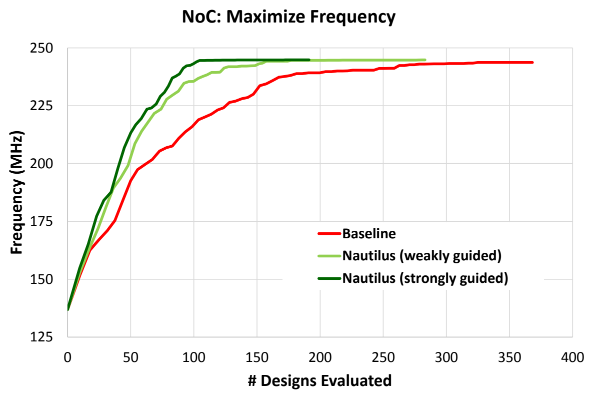

Figure 4: Maximizing frequency in the NoC design space.

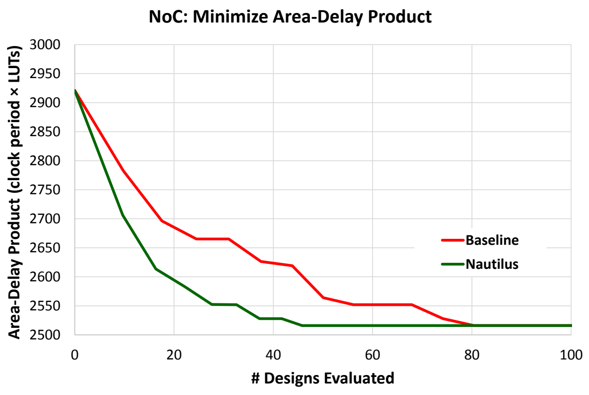

Figure 5: Minimizing the Area-Delay product in the NoC design space.

4.2 Results

NoC. We first look at the NoC results, where Nautilus is guided by non-expert hints. We compare two Nautilus variants ("strongly guided" and "weakly guided"[2]) against the baseline GA. Figure 4 shows the result of a query aimed at finding designs in the NoC design space that can achieve the highest frequency. The y-axis shows the maximum frequency for the best sample in each algorithm's population and essentially captures how the quality of results changes as the algorithms progress. The x-axis shows the cumulative number of synthesis jobs needed for each of the three techniques, reflecting the computational cost of the query. All three techniques are run for 80 generations of the GA. Note however that the Nautilus lines require fewer designs to be synthesized (because the GA revisits previously-synthesized results as it converges), allowing their lines to stop after fewer designs. Both the strongly-guided and weakly-guided configurations of Nautilus approach good solutions much faster than the baseline GA. The baseline GA requires about 2.8x and 1.8x the number of synthesis jobs to converge to a solution within 1% of the best solution.

Figure 5 shows the results for our second NoC query, which aims at minimizing the area-delay product of a design. Here, results are shown only for the first 20 generations, because both techniques converged to the optimal solution within this time. While our previous query only used hints related to frequency, this query also incorporates hints related to the importance and bias of IP parameters that affect area, such as virtual-channel buffer depth. In this case, Nautilus achieves similar quality of results with about half the number of synthesis runs required by the baseline. It is interesting to note that even the non-expert guided Nautilus performs significantly better than the baseline, offering much better quality of results for the same computational cost or the same quality of results after significantly fewer synthesis jobs.

FFT. We next turn to the FFT results. As mentioned earlier, in this case Nautilus is "expert-guided", i.e., a developer of the FFT IP generator set the hints. Figure 6 shows the result of a query aimed at minimizing the number of LUTs used by an implementation from the FFT dataset.

Here we see that all three methods eventually converge on a same result (about 540 LUTs), but the Nautilus designs converge much more quickly and require far fewer designs

[2]The strongly and weakly guided lines differ only in the "confidence" hint.

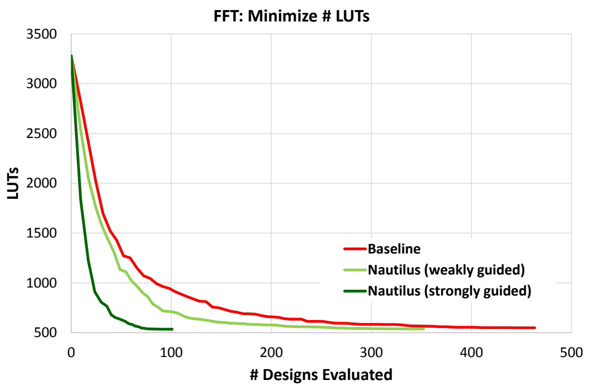

Figure 6: Minimizing the number of LUTs in the FFT design space.

to be synthesized: the strongly guided Nautilus strategy converges on the optimal design using an average of 101 synthesis runs, while the baseline GA requires 463 designs to be synthesized (on average) to reach an equivalent result. If we relax the goal to 1,071 LUTs (twice the minimum), we see that the strongly guided Nautilus technique is able to meet the goal synthesizing 23.6 designs (on average), while the baseline GA requires synthesizing an average of 78.9 designs to reach the same quality result.[3]

Figure 7 aims to search the FFT design space for a design that uses a composite metric to maximize the ratio of throughput to logic area consumed: throughput in million samples per second (MSPS) divided by the number of LUTs. In this case, once again the strongly and weakly guided Nautilus variants find significantly better solutions in less time; for example, the strongly guided Nautilus strategy is able to reach 1.45 MSPS per LUT using 61.6 synthesis runs (on average), while the baseline GA requires more than 8x synthesis runs (501.4 on average) to reach the same value. Moreover, Nautilus is able to reach high-quality solutions exhibiting more than 1.5 MSPS per LUT, which the baseline is never able to approach even after having explored a much larger portion (>5x) of the design space.

Overall, Nautilus outperformed the baseline GA, both when guided by a non-expert (NoC), and especially when given IP-expert guidance (FFT). Nautilus consistently produces higher quality of results at much lower computational

[3]For comparison, if random sampling was used, it would take on average 11,921 synthesis runs to find a design meeting this goal.

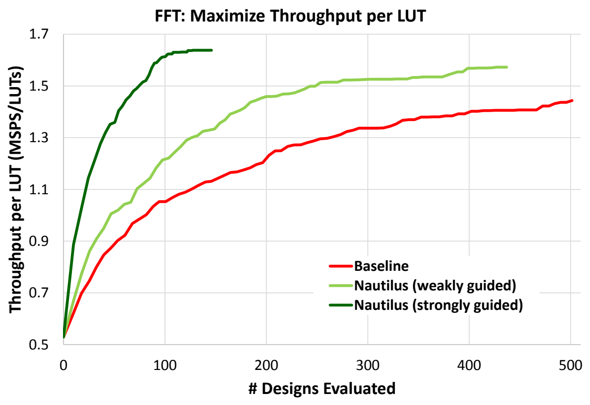

Figure 7: Maximizing throughput per LUT in the FFT design space.

cost across multiple optimization queries in two separate IP domain spaces.

5. RELATED WORK

GA in EDA. Stochastic methods such as genetic algorithms have been used for a variety of aspects of hardware exploration, from the circuit level (e.g., low-level VLSI layout optimization [8] and yield-aware circuit sizing [17]), to the system level (such as [15] and [7], in which genetic algorithms are used to optimize aspects of hardware-software codesign). [12] uses GAs to perform exploration across a space of closely-related processor systems, which can be evaluated in only a few seconds per design. Other approaches focus on high-level synthesis, such as [10, 5], which use genetic algorithms and Monte Carlo methods (respectively) for HLS optimization. Lastly, simulated annealing has long been used in physical design automation problems (e.g., [2]).

Active Learning. Another important class of related work is research on *active learning*, a family of learning techniques that iteratively evaluate potentially useful points within a set. Several recent works [18, 19, 6, 9] use active learning techniques to model the entire Pareto-optimal set of design points across a multi-objective space; specifically, [18, 19] consider costs and performance of generated IPs. These approaches differ from our work in that they aim to understand all design points that give a Pareto-optimal trade-off among any of the design characteristics. As we are considering design spaces with tens of thousands of possible designs and on the order of ten cost and performance metrics, this generalized approach would become extremely difficult; instead we aim to provide a system that can answer a given query about this complicated design space using as few synthesis steps as possible.

6. CONCLUSION

IP-based design can improve hardware designer productivity. However, to avoid inefficiency, one must not only select IPs of the correct functionality but also fine-tune the IPs to match the overall project's design tradeoffs across a myriad of implementation- and application-level goals. While a highly parameterized IP can provide the needed customization flexibility, we have observed in today's most complicated IPs and IP generators, that just to understand and correctly set an IP's parameters can become unmanageable to the average IP user (who may lack both the highly domain-specific knowledge surrounding the IP and a detailed understanding of the IP's inner workings). This paper offers a genetic algorithm-based approach to automatically tune IP parameters on behalf of the IP user to meet the IP user's optimization goals. The central novelty lies in our extensions to GA to allow the IP authors to embedded different classes of hints about the design space to improve the quality and speed of design space exploration. For two example IP generators, our evaluations show that GA is indeed an effective way to automatically tune IP parameters and that embedding IP author knowledge can reduce the number of designs points that need to be evaluated by as much as a factor of eight. This is a significant improvement in real usage scenarios where evaluating a design point requires costly synthesis and characterization by EDA tools.

Acknowledgments. Funding for this work was provided in part by NSF CCF-1012851.

7. REFERENCES

[1] Pyevolve. `http://pyevolve.sourceforge.net`.
[2] C. J. Alpert, D. P. Mehta, and S. S. Sapatnekar. *Handbook of algorithms for physical design automation*. CRC Press, 2008.
[3] T. Back. An overview of evolutionary algorithms for parameter optimization. *Evolutionary Computation*, 1(1):1–23, 1993.
[4] D. U. Becker. *Efficient Microarchitecture for Network-on-Chip Routers*. PhD thesis, 2012.
[5] D. Bruni, A. Bogliolo, and L. Benini. Statistical design space exploration for application-specific unit synthesis. In *Design Automation Conference (DAC)*, pages 641–646, 2001.
[6] P. Campigotto, A. Passerini, and R. Battiti. Active learning of Pareto fronts. *IEEE Transactions on Neural Networks and Learning Systems*, 25(3):506–519, Mar. 2014.
[7] R. Dick and N. Jha. MOGAC: a multiobjective genetic algorithm for hardware-software cosynthesis of distributed embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(10), 1998.
[8] H. Esbensen and E. Kuh. Design space exploration using the genetic algorithm. In *IEEE International Symposium on Circuits and Systems*, volume 4, pages 500–503, 1996.
[9] J. Knowles. ParEGO: a hybrid algorithm with on-line landscape approximation for expensive multiobjective optimization problems. *IEEE Transactions on Evolutionary Computation*, 10(1):50–66, Feb. 2006.
[10] V. Krishnan and S. Katkoori. A genetic algorithm for the design space exploration of datapaths during high-level synthesis. *IEEE Transactions on Evolutionary Computation*, 10(3):213–229, 2006.
[11] P. Milder, F. Franchetti, J. C. Hoe, and M. Püschel. Computer generation of hardware for linear digital signal processing transforms. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 17(2):15, 2012.
[12] M. Palesi and T. Givargis. Multi-objective design space exploration using genetic algorithms. In *CODES*, 2002.
[13] M. K. Papamichael. CONNECT NoC Generation Framework. `http://users.ece.cmu.edu/~mpapamic/connect/`.
[14] M. K. Papamichael and J. C. Hoe. CONNECT: Re-Examining Conventional Wisdom for Designing NoCs in the Context of FPGAs. In *FPGA*, 2012.
[15] D. Saha, R. S. Mitra, and A. Basu. Hardware software partitioning using genetic algorithm. In *International Conference on VLSI Design*, pages 155–160, 1997.
[16] O. Shacham. *Chip multiprocessor generator: automatic generation of custom and heterogeneous compute platforms*. PhD thesis, 2011.
[17] S. K. Tiwary, P. K. Tiwary, and R. A. Rutenbar. Generation of Yield-Aware Pareto Surfaces for Hierarchical. In *Design Automation Conference (DAC)*, pages 31–36, 2006.
[18] M. Zuluaga, A. Krause, P. Milder, and M. Püschel. "Smart" design space sampling to predict Pareto-optimal solutions. In *Proceedings of International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, 2012.
[19] M. Zuluaga, A. Krause, G. Sergent, and M. Püschel. Active learning for multi-objective optimization. In *International Conference on Machine Learning*, volume 28, 2013.