Bayesian Optimization for Efficient Accelerator Synthesis

ATEFEH MEHRABI, Duke University
ANINDA MANOCHA, Princeton University
BENJAMIN C. LEE, University of Pennsylvania
DANIEL J. SORIN, Duke University

Accelerator design is expensive due to the effort required to understand an algorithm and optimize the design. Architects have embraced two technologies to reduce costs. High-level synthesis automatically generates hardware from code. Reconfigurable fabrics instantiate accelerators while avoiding fabrication costs for custom circuits. We further reduce design effort with statistical learning. We build an automated framework, called Prospector, that uses Bayesian techniques to optimize synthesis directives, reducing execution latency and resource usage in field-programmable gate arrays. We show in a certain amount of time that designs discovered by Prospector are closer to Pareto-efficient designs compared to prior approaches. Prospector permits new studies for heterogeneous accelerators.

CCS Concepts: • **Hardware → High-level and register-transfer level synthesis**; **Modeling and parameter extraction**;

Additional Key Words and Phrases: High-level synthesis, design space exploration, FPGA, Bayesian optimization

ACM Reference format:
Atefeh Mehrabi, Aninda Manocha, Benjamin C. Lee, and Daniel J. Sorin. 2020. Bayesian Optimization for Efficient Accelerator Synthesis. *ACM Trans. Archit. Code Optim.* 18, 1, Article 4 (December 2020), 25 pages.
https://doi.org/10.1145/3427377

4

1 INTRODUCTION

As Dennard scaling ends, the pursuit of specialized hardware for energy efficiency has become prevalent. Accelerators have been developed for various applications, including machine learning [8, 63], robotics [43], web search [49], and others [26, 36]. Accelerators derive their energy efficiency from specializing datapath and control for specific functionality [21]. Many platforms, including datacenters [49], instantiate accelerators on field-programmable gate arrays (FPGAs) to realize their benefits while avoiding the costs of custom circuits.

The need for accelerators has renewed interest in high-level synthesis (HLS) [27], a toolflow in which designers specify functionality in a high-level language and automated tools produce

This work is supported by National Science Foundation grants CCF-1149252, CCF-1337215, SHF-1527610, and AF-1408784. This work is also supported, in part, by the Semiconductor Research Corporation's Global Research Collaboration (GRC) program under task 2821.001.
Authors' addresses: A. Mehrabi and D. J. Sorin, Department of ECE, Duke University, PO Box 90291, Durham, NC 27708; emails: atefeh.mehrabi@duke.edu, sorin@ee.duke.edu; A. Manocha, Princeton University; email: amanocha@princeton.edu; B. C. Lee, University of Pennsylvania; email: leebcc@seas.upenn.edu.

This work is licensed under a Creative Commons Attribution International 4.0 License.

© 2020 Copyright held by the owner/author(s).
1544-3566/2020/12-ART4
https://doi.org/10.1145/3427377

register transfer level (RTL) code. Such toolflows offer large productivity gains when compared to manually writing RTL, a time-consuming process that requires expertise in digital design. But challenges remain because designers must rely on HLS directives to guide synthesis and produce quality RTL [65].

Directives are hints to the HLS tool, indicating that some code locations could be optimized to improve performance. Designers of complex accelerators need to tune varied directives at different code locations, which define a vast design space. The size of the space depends on the number of lines of code, the choice of directives (e.g., loop unrolling), and the choice of settings for a given directive (e.g., unrolling factor). Even worse, the effects of multiple directives interact to affect performance in subtle and unexpected ways. Given the scale of the challenge, designers need automated solutions that generate many RTL variants and explore trade-offs between cost and performance yet deliver the desired accelerator functionality.

We have developed Prospector [39], a framework for synthesizing efficient accelerators with optimization directives. While Prospector supports both ASICs and FPGAs, this article focuses on FPGAs given the growing interest in reconfigurable accelerators. The framework coordinates the placement and configuration of directives, seeking low execution time and efficient resource usage. Prospector achieves these goals in two ways. First, it encodes the design space so that statistical models can capture accelerator performance and FPGA costs (e.g., flip-flops, lookup tables, block RAMs, and digital signal processors) more effectively. Second, as HLS measurements are expensive, it samples the design space in order to reveal optimal designs more efficiently.

Prospector uses Bayesian optimization, a method starting to find success in digital design [37, 52], to judiciously collect data, incrementally train models, and efficiently optimize designs. Efficient data collection and analysis is critical because evaluating each point of the design space involves costly synthesis and place-and-route. We show that Prospector efficiently reveals design optima by running HLS measurements on a small percentage (e.g., $< 1\%$) of the whole design space. Such capabilities reflect Bayesian optimization's particular strengths, which are absent in popular search heuristics, such as genetic algorithms and simulated annealing.

This article builds on the original Prospector work [39] in three ways. First, we provide more depth and details on both the statistical basis for Prospector and the components of the Prospector framework. Second, we extend the evaluation of Prospector with new experiments that provide additional insight into Prospector's behaviors and relationship to prior work. Third, we develop a case study of Prospector usage, in which Prospector improves heterogeneity gains by finding more accurate Pareto-optimal design variants.

The following summarizes our contributions:

- **Effective Search Algorithms.** Prospector places and configures optimization directives by modeling their effect on accelerator performance and FPGA resource usage. Concise design encodings and intelligent design sampling permit the use of Bayesian optimization for HLS.
- **Efficient Resource Usage.** Prospector discovers designs that meet performance targets using fewer FPGA resources. Compared to classic approaches, Prospector discovers designs that require 35% fewer FPGA resources on average (Table 4).
- **Broad Pareto Frontiers.** Prospector captures high-dimensional trade-offs between performance and FPGA cost and finds more accurate Pareto frontiers. The distance between the Pareto frontier and the true frontier for Prospector is 0.57× (less distant) of the average distance for alternative approaches (Figure 11).
- **Heterogeneous Accelerator Design.** Prospector supports heterogeneous mixes of accelerator designs, allowing the FPGA to use trade-offs between fast but large and small but slow accelerators to respond when workload mixes evolve and resource allocations change.

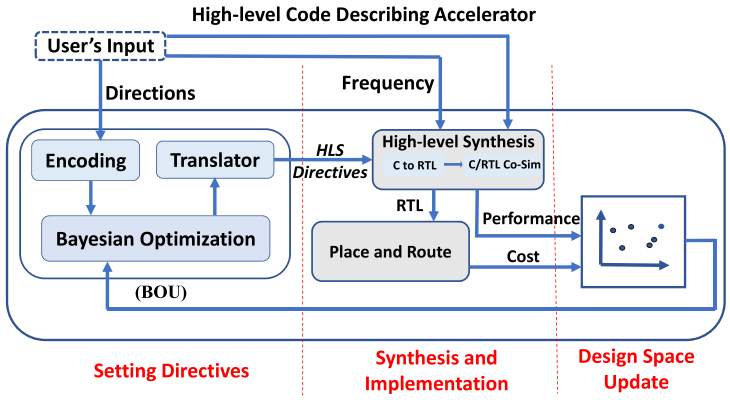


Fig. 1. Prospector framework.

2 THE PROSPECTOR FRAMEWORK

We optimize accelerator design flows that use HLS to target FPGAs. HLS reduces design effort by compiling behavioral descriptions into hardware descriptions at the RTL [12, 44]. Behavioral descriptions support high levels of abstraction that allow architects to express design intent efficiently. HLS produces RTL, allowing architects to simulate performance, estimate resource utilization, and verify functionality. RTL results are used for FPGA implementations and more accurate resource usage estimates via place-and-route.

We develop a statistical framework, called Prospector, for identifying synthesis parameters that best optimize an accelerator design. We leverage Bayesian optimization, which builds a probabilistic model that approximates an unknown function and serves as its surrogate [56]. Bayesian techniques are particularly effective when the function is a black box and evaluating the function is costly. The technique iteratively samples parameter values, evaluates the function with those values, and updates the probabilistic model. Over multiple iterations, the optimization approaches optimal parameter values $\hat{x} = \arg\max_{x \in X} f(x)$ for function f.

We consider the HLS toolflow (i.e., simulate, synthesize, place-and-route[1]), an unknown function to be modeled and optimized. The function's inputs specify the placement and configuration of synthesis directives [11, 12]. The function's outputs quantify design quality, which include execution time and multiple measures of FPGA resource utilization. Although we can invoke the HLS toolflow to evaluate the function for any set of inputs, evaluations are prohibitively expensive when exploring large, complex design spaces.

Figure 1 summarizes the Prospector framework. Its input is high-level source code that describes accelerator functionality. Its outputs are RTL implementations that reflect varied performance and cost trade-offs. Within Prospector, the Bayesian optimization unit (BOU) explores the design space iteratively. In each iteration, the BOU controls the choice of synthesis directives and the HLS toolflow converts source code to RTL. After multiple iterations, Prospector identifies synthesis directives and accelerator designs that balance execution time and FPGA resource utilization.

2.1 Design Space

Architects use directives to specify optimizations and guide HLS. For example, software pipelining increases throughput by executing instructions from multiple loop iterations simultaneously [50]. Memory allocation increases throughput by distributing data across independent memory banks

[1]Place-and-route is a user option in Prospector, and we assume it is chosen in this article.

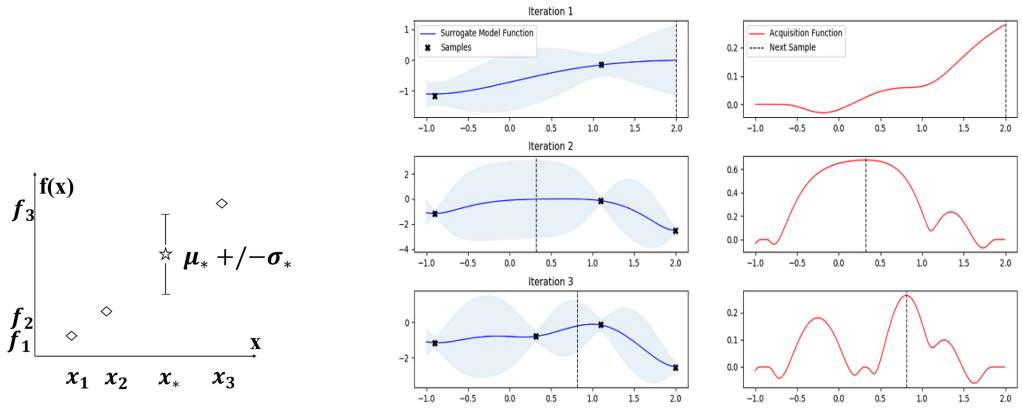


Fig. 2. Gaussian process with measurements for x_1, x_2, x_3 and prediction for x_* (left). Bayesian optimization updates the model as data is acquired (center). The acquisition function selects data measurements based on expected benefits (right).

[47]. Determining which directives are effective requires time-consuming simulation. The problem is complicated by two inter-related questions:

- **Placement.** Which code locations should be targeted by an optimization?
- **Configuration.** What values should an optimization's arguments take?

Both placement and configuration heavily influence optimization effectiveness and design quality. The design space size increases exponentially with the number of loops, arrays, and functions targeted for acceleration. An architect may need to try many directives because interactions between the different optimizations are difficult to anticipate. If each trial were to invoke HLS, the computation for simulation, synthesis, and place-and-route would require days if not weeks. To understand the magnitude of the optimization problem, consider a benchmark like 2mm that adds product matrices from two matrix multiplications. The spectrum of unrolling factors and pipeline initiation intervals for its nested loops (shown later in Table 2) produces a parameter space with over 15,000 points.

Tuning the HLS parameter space is particularly important for reconfigurable platforms. FPGAs are cost-effective and have been deployed in numerous high-performance settings such as datacenters [10, 24, 49]. Their reconfigurability is necessary when user demands vary across time, applications evolve quickly, and data inputs affect implementation choices [49]. But reconfigurability is also challenging when architects seek to simplify application development while retaining accelerator efficiency [23]. Responsive reconfigurability requires methods to identify and realize the best accelerator design from large design spaces.

2.2 Statistical Model

The Gaussian process is a statistical model of an unknown function. For each input, the model estimates the function's output and the uncertainty around that estimate. The model is trained by sampling inputs, evaluating the function, and obtaining outputs. These sampled measurements supply data that refine estimates for unobserved outputs. Moreover, they reduce uncertainty around estimates for the corresponding outputs. As data become available, the Gaussian process updates its model of the function to produce increasingly accurate and confident estimates.

Figure 2 illustrates the Gaussian process. Suppose we obtain data $\{(x_1, f_1), (x_2, f_2), (x_3, f_3)\}$ by evaluating the function. We model output f_* for previously unobserved input x_* using a Gaussian

random variable with mean μ_* and standard deviation σ_*. Intuitively, μ_* should be near the outputs for inputs similar to x_* and σ_* should decrease when more outputs are observed for inputs similar to x_*.

The Gaussian process uses samples as training data to construct a multivariate Gaussian model. The kernel function K constructs the covariance matrix for the joint Gaussian distribution of the function outputs (f) based on the function inputs (x). Many kernel functions exist, but the squared exponential kernel is popular because it models high correlation between outputs f_i and f_j when inputs x_i and x_j are similar.

$$\begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix} \sim N\left[\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \quad \begin{pmatrix} K_{11} & K_{12} & K_{13} \\ K_{21} & K_{22} & K_{23} \\ K_{31} & K_{32} & K_{33} \end{pmatrix}\right]$$

$$K_{ij} = exp\left(-\frac{(x_i - x_j)^2}{2}\right) => \begin{cases} K_{ij} = 0 \ |x_i - x_j| \to \infty \\ K_{ij} = 1 \ x_i = x_j \end{cases}$$

2.3 Data Acquisition

Gaussian processes and Bayesian optimization use data sparingly. An acquisition function incrementally selects inputs for which the target function should be evaluated, producing a sequence of implemented designs (x, f). Early in the procedure, the acquisition function favors exploration and selects inputs for which the function's output is uncertain. Later, it favors exploitation and selects inputs for which the function's output is likely closer to the optimum. Thus, data collection reflects prior measurements and statistical uncertainty about the function's output.

Figure 2 illustrates updates to the Gaussian process and the acquisition function's estimate of utility from data. This example selects the input based on expected improvement over the current optimum modeled by the Gaussian process. Many acquisition functions have been proposed to select the next evaluation based on varied goals such as probability of improvement, expected improvement, GP confidence bounds, etc. [7, 59]. Prospector uses the PESMO acquisition function for multi-objective Bayesian optimization. PESMO selects inputs to reduce the estimated Pareto frontier's entropy [22]. The Pareto frontier includes points for which we cannot further optimize one dimension in the objective without harming another dimension. PESMO is an effective acquisition function when synthesizing accelerators for reconfigurable substrates because the accelerator consumes resources in multiple dimensions. The optimization seeks to reduce both execution time and FPGA resource utilization, measured in terms of the number of flip-flops (FFs), lookup tables (LUTs), digital signal processors (DSPs), and block RAMs (BRAMs).

2.4 Prediction and Optimization

As the acquisition function supplies data to train the Gaussian process, predictions become more accurate. Suppose we wish to predict the function's output f_* at x_*, a previously unobserved input. The Gaussian process assumes that f_* follows a Gaussian distribution with mean $m^* = 0$ and variance $K_{**} = 1$ and extends the multivariate Gaussian distribution. To aid the explanation, let us define covariance for prior inputs K, covariance for the target input K_*, and variance for the target input K_{**}.

$$\begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_* \end{pmatrix} \sim N\left[\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad \begin{pmatrix} K_{11} & K_{12} & K_{13} & K_{1*} \\ K_{21} & K_{22} & K_{23} & K_{2*} \\ K_{31} & K_{32} & K_{33} & K_{3*} \\ K_{*1} & K_{*2} & K_{*3} & K_{**} \end{pmatrix}\right]$$

$$K = \begin{bmatrix} K_{11} & K_{12} & K_{13} \\ K_{21} & K_{22} & K_{23} \\ K_{31} & K_{32} & K_{33} \end{bmatrix}; K_* = \begin{bmatrix} K_{1*} \\ K_{2*} \\ K_{3*} \end{bmatrix}; K_{**} = 1$$

We calculate the posterior conditional distribution of f_* given data $f = \{f_1, f_2, f_3\}$ from the procedure for calculating marginal and conditional distributions for multivariate Gaussians [42].

$$p(f_*|f) \sim N(\mu_*, \sigma_*) = N(m^* + K_*^T K^{-1} f, K_{**} - K_*^T K^{-1} K_*)$$

In Bayesian terminology, the posterior mean μ_* updates the prior m^* to reflect observed outputs f. The posterior standard deviation σ_* updates the prior K_{**} to reflect covariance between observed outputs K.

The mean μ_* predicts the output and the standard deviation σ_* assesses prediction uncertainty. Accuracy increases and uncertainty decreases as the training process evaluates the function for inputs similar to x_*. The estimate for μ_* is a linear combination of fs that assigns greater weights when the corresponding xs are similar to x_*. The estimate for σ_* decreases as similarities between x_* and x increase.

Bayesian optimization and the acquisition function use predictions to assess design quality, assess model uncertainty, and guide data acquisition in the search for inputs that optimize the output. In this article, the inputs encode optimization directives used for HLS. The outputs describe figures of merit in accelerator design. We use separate Gaussian processes for execution time, FF usage, LUT usage, DSP usage, and so forth. This formulation permits multi-dimensional Bayesian optimization, which is important when navigating trade-offs for accelerators synthesized on FPGAs.

2.5 Directive Encodings

We refine Gaussian processes by iteratively synthesizing and evaluating a sequence of designs (x, f).

Inputs. We define input x to be a vector of variables, each of which encodes the use of a synthesis directive. First, we encode directive placement using a binary variable, indicating whether a directive is applied, for each code location that could be optimized. Next, we encode directive configuration using a categorical variable to describe the optimization's mode and integer variables to specify the optimization's tunable argument.

Outputs. The synthesis of input x produces output f. We define f to be a vector of metrics relevant to our optimization objective. These metrics include execution time and utilization for each type of FPGA resource including FFs, LUTs, DSPs, and BRAMs.

Encoding Directive Placement. Users specify the code locations for which they wish to explore optimization directives. For instance, users can select labels for loops in the high-level code. Based on user specifications, Prospector defines parameters for the number of code locations L and the number of possible directives D for each location. We construct a bit map for each directive. The map's length equals the number of locations optimized. The result is D maps with L bits each. Each bit indicates whether a directive $d \in [1, D]$ is active at location $l \in [1, L]$. We convert each directive's map into an integer between zero and $2^L - 1$. For example, if each of three loops could be unrolled, we encode the unrolling strategy with a three-bit map that corresponds to an integer between zero and seven. Thus, the bitmaps permit Gaussian processes to model and explore directive placement. Because some directives do not apply to some locations (e.g., function inlining does not apply to loop locations), users can provide separate sets of L and D values for different types of locations. Placement is monitored either via this separate integer parameter or by considering inactive state as one possible configuration value for all locations. For example, an

```
loopjj:for (jj = 0; jj < row_size; jj += block_size)
    loopkk:for (kk = 0; kk < row_size ; kk += block_size)
        loopi: for (i = 0; i < row_size ; ++i)
            loopk: for (k = 0 ; k < block_size; ++k)
                i_row = i * row_size;
                k_row = (k + kk) * row_size;
                temp_x = m1[i_row + k + kk];
                loopj: for (j = 0; j < block_size; ++j)
                    mul = temp_x * m2[k_row + j + jj];
                    prod[i_row + j + jj] += mul;
```

Fig. 3. bbgemm code.

```
LocationUnroll = 2, LocationPipeline = 1
Pipeline_i = 3, Pipeline_j = 2, Pipeline_k = 5
Unroll_i = 4, Unroll_j = 8, Unroll_k = 4
```

Fig. 4. Input (x) selected by PESMO acquisition function.

```
set_directive_unroll -factor 4 bbgemm/loopk
set_directive_pipeline -II 2 bbgemm/loopj
```

Fig. 5. Input (x) translated into HLS directives.

unroll factor of 1 means the unrolling directive is applied to repeat the loop by a factor of one, which technically means the loop is not unrolled.

Encoding Directive Configuration. We specify the range of possible values for each directive's categorical and numerical settings. Prospector explores and samples directive configurations within these ranges. For example, a loop could be unrolled by $2x$, where $x \in [1:4]$, or pipelined with an initiation interval $x \in [2:5]$. Note that we sample parameters for directive placement and configuration independently. The sampled unrolling factor for a code location has an effect only if the sampled bitmap activates unrolling for that location.

Example. Figure 3 presents code for blocked matrix multiplication, which iterates over matrix blocks and block elements in five nested loops. We optimize three code locations (e.g., loops i, j, k), each with two possible directives (e.g., loop unrolling, pipelining). In each optimization step, the framework samples an input, translates the input into directives, and runs the HLS toolflow.

In Figures 4 and 5, the `LocationUnroll` bitmap indicates that the second loop is unrolled ($2_{10} = 010_2$) and the `LocationPipeline` bitmap indicates that the innermost loop is pipelined ($1_{10} = 001_2$). The input specifies initiation intervals for each loop in case the sampled bitmap applies pipelining and specifies unrolling factors for each loop in case the sampled bitmap applies unrolling. Thus, our strategy for encoding HLS inputs models and optimizes directive placement and configuration simultaneously.

Scalability. Our encoding produces concise inputs for Bayesian optimization. This efficiency allows the framework to explore both directive placement with categorical, discrete variables as well as directive configuration with numerical, continuous variables. The encoding produces vectors with $D \times (1 + L)$ elements, using integers to determine whether D directives are enabled at various locations and $D \times L$ integers to specify numerical configurations for those directives. Expanding the range of configurations increases the range of values for the corresponding $D \times L$ elements but does not lengthen the vector.

In contrast, prior work is limited by its reliance on binary flags alone to indicate whether directives are applied at code locations [37]. Such a one-hot encoding requires a rapidly increasing number of binary flags and one vector element per possible use of HLS directives. For example, unrolling a target loop by two and by four is treated as two separate directives, each with its own binary flag at each location. The encoded vector length increases with the number of permissible directives, numerical settings, and code locations. Because the encoding does not scale, prior work focuses on directive placement and excludes directives' numerical settings.

Constraining Exploration. A multi-objective acquisition function addresses limitations from single-objective analysis, which optimizes a single-dimension subject to constraints on all other dimensions. An example of single-objective analysis is latency minimization subject to constraints on the number of LUTs used [37]. Although this approach accounts for multiple figures of merit, the designer must specify reasonable constraints on all but one dimension prior to data collection. If specified constraints are too lax or stringent, Bayesian optimization is likely to return results that do not reside on the multi-dimensional Pareto frontier.

Prospector is robust even when the designer provides no guidance about or constraints on reasonable directive usage. When evaluation reports high execution time or resource utilization, the acquisition function is less likely to sample from the corresponding neighborhood of inputs in future iterations. When synthesis fails to generate RTL or warns about long simulation times, Prospector assigns high execution times to the corresponding outputs, discouraging further exploration of similar directive configurations. For example, HLS generates warnings when pipelining the outer-most loop in blocked matrix multiplication. Despite these pitfalls, Prospector converges to synthesis inputs that optimize design quality.

Although Prospector does not require domain-specific guidance, it can explore inputs more efficiently when the designer supplies such guidance. Guidelines might constrain particular combinations of directives or specify dependencies between directives, thereby pruning optimizations known to perform poorly. For example, unrolling every loop in blocked matrix multiply makes little sense. Using one directive for a loop might preclude the use of others on the same loop. Prospector can ensure the acquisition function selects inputs that satisfy user constraints before generating HLS directives.

3 EXPERIMENTAL METHODS

Benchmarks. Prospector supports any benchmark written in a high-level language that interfaces with HLS tools. We evaluate application kernels from two benchmark suites, PolyBench [48] and MachSuite [51], which cover a wide range of functionality and complexity. We explore design spaces for fdtd-2d, 2mm, and heat-3d from PolyBench and fft, bbgemm, and stencil-3d from MachSuite.

Bayesian Optimization Unit (BOU). The BOU generates new values for target directives at each step in the optimization procedure. The BOU integrates our encoding and translation mechanisms with Spearmint,[2] a software package that implements Gaussian processes and acquisition functions. The encoding unit converts a user-defined space of HLS directives and code locations into input parameters for Spearmint. Spearmint's acquisition function selects parameter values, which the translation unit converts into a TCL script that describes how directives are to be located and configured. HLS uses the script to generate a hardware description. This hardware description is used for place-and-route to profile its performance and assess its costs. Data from the design profile updates Spearmint's Gaussian process and influences the acquisition function's subsequent selections.

[2]Spearmint: https://github.com/HIPS/Spearmint/tree/PESM.

Table 1. Directives' Ranges That Define the Design Space

Directive	Parameter	Values
Loop Unrolling	Factor	$\forall 2x \mid x \in [0 : M]$
Loop Pipelining	Initiation Interval	$\forall x \mid x \in [1 : 7]$
Array Partitioning	Factor	$\forall 2^x \mid x \in [1 : N]$
Function Inlining	Setting	on/off
Allocation	Instance and Limit	instance $\in$ [add,mul], $\forall x \mid x \in [1 : N_{add}]$, $\forall 2x \mid x \in [1 : N_{mul}]$

The values of M, N differ across benchmarks. See Table 2.

Table 2. Design Space Summary

Benchmark	Optimization Targets	Range	Design Space Size
fdtd-2d	4 loops	M=9	6,561
2mm	4 loops, 1 array	M=12, N=1	15,625
fft	9 loops, 3 arrays, 3 functions, 2 allocations	M=8, N_{add}=10, N=2, N_{mul}=8	36,000
bbgemm	2 loops, 1 array	M=16, N=4	960
stencil-3d	2 loops, 1 array	M=16, N=4	960
heat-3d	4 loops	M=16	2,304

High-Level Synthesis and Place-and-Route. We use Xilinx Vivado HLS, which produces RTL from high-level code. Vivado HLS simulates our benchmark's C code to determine functionality and then synthesizes RTL in three steps. First, Vivado HLS determines which operations occur in each cycle based on clock rate. Second, it binds hardware resources to operations according to optimization directives and the targeted FPGA. Third, Vivado HLS extracts the finite state machine that controls the RTL, storage, and I/O. The final RTL result is fed into the Vivado Design Suite to perform place-and-route and generate the bitstream to program the FPGA.

Directives. Table 1 defines a set of directives: loop unrolling, loop piplelining, array partitioning, function inlining, multiplier allocation, and adder allocation. This set includes the most effective directives found in our experiments, which align with those in prior work [53]. These directives define a design space with thousands of points for which exhaustive analysis would require days of synthesis and simulation.

Table 2 reports the number of loops and arrays that are targeted by our optimizations, the range of parameter values that were considered, and the total size of the design space. For a robust experimental evaluation, we also explore the entire design space with exhaustive search. In practice, Prospector does not require exhaustive search, and broadening the design space to include more directives is possible.

First, directives for loop unrolling and pipelining specify the target loop and values for the unrolling factor and initiation interval. Unrolling generates the specified number of copies for the loop body, improving parallelism by using more hardware. Pipelining starts the next iteration before the current one finishes, improving parallelism with concurrent execution. The initiation interval specifies how many cycles separate the start of two loop iterations. Second, directives for array partitioning specify the array and number of partitions. Breaking large arrays into smaller ones or single registers permits concurrent access to multiple block RAMs. Third, function inlining removes the hierarchy of the function to potentially improve logic usage between the function and its caller. Fourth, allocation directives restrict the maximum number of instances of a specific operation/function that can be used.

We use two additional directives to guide the use of loop and array directives. First, the "dependence" directive guides the handling of loop-carried dependencies. In some scenarios, such as variable-dependent array indexing, Vivado HLS's automatic analysis of dependencies may be conservative and leave parallelism (e.g., loop pipelining) unexploited. The dependence directive indicates when loop-carried dependencies are false. For example, the directive is helpful where loop-carried dependencies do not exist in source code and loop pipelining is an important performance optimization. The directive removes dependencies from the code region and allows HLS to pursue parallelism more aggressively. Prospector automatically considers applying the dependence directive on loops requested by users. Second, the "resource" directive guides an operation's implementation, identifying particular compute or memory units for operations or data structures.

For all designs, with whatever directive settings are used, safe parallelism is ensured automatically by the last HLS stage, C/RTL co-simulation, which verifies the RTL. In standard HLS toolflow, after the HLS tool produces RTL from C code, the RTL's functional correctness is verified using a testbench that compares the C code and RTL code output on a reference dataset. If a specific directive setting violates the rules and produces incorrect outputs, the RTL's functionality will not match the C code's. If verification fails, Prospector discards the design point and continues with its exploration.

We consider comprehensive design spaces, defined by many directives and settings, that include sub-optimal designs. Prospector defines design space boundaries generously for two reasons. First, an automated framework for design space exploration cannot know *a priori* the point of diminishing marginal returns from an optimization directive and the most effective boundaries for the space. Moreover, the framework should not need the designer to specify tight boundaries for exploration, a difficult task given how multiple, interdependent directives affect performance and cost.

4 EVALUATION

We evaluate Prospector's ability to find optimal design points and reveal the Pareto frontier. We first perform an exhaustive characterization of the design space by running every possible design point through HLS and place-and-route, measuring execution time and FPGA utilization. These measurements produce the **Golden Pareto Frontier**, which we take as the optimal baseline because it reveals the best designs after trying all possible configurations of D directives on L locations. We determine how closely Prospector and alternative heuristics compare against these optima. The total number of points in each design space for exhaustive search is in Table 2.

Because Prospector relies on search, we compare its single- and multi-dimensional variants against popular search heuristics for design space exploration. The baseline heuristics account for multiple dimensions in different ways. Some heuristics, like genetic algorithm and random search, account for multiple dimensions easily. Other heuristics, like simulated annealing, account for multiple dimensions ineffectively. Evaluated methods include:

- **Prospector.** Models and optimizes one or multiple objective function(s), including latency[3] and multiple dimensions of FPGA usage, with Gaussian processes. Prospector searches for Pareto optima based on objective(s). Prospector denotes optimization for all five dimensions of our problem (latency, FFs, LUTs, DSPs, and BRAMs), while Prospector-kD denotes optimization for $k < 5$ dimensions.
- **Random Search.** Samples uniformly at random from the design space, evaluating these samples based on design objectives [4]. The sampling procedure is independent of the number of design objectives, and we do not require multiple variants for multiple objectives.

[3]Improving latency leads to higher throughput, which is used in Section 5.

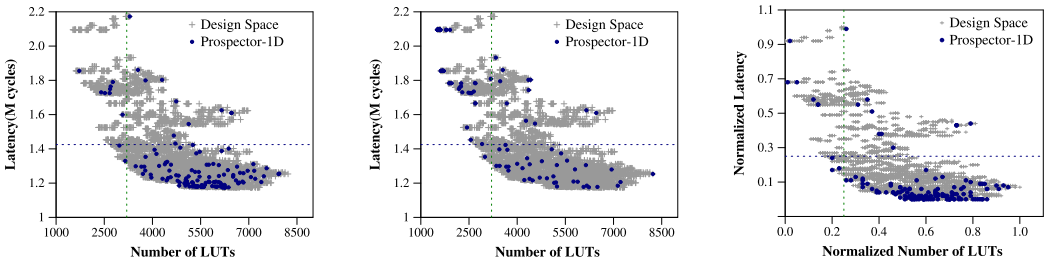


Fig. 6. Latency, LUT usage across iterations of Prospector-1D when minimizing (a) latency, (b) latency-LUT product, and (c) latency-LUT product with normalized, scaled measures. Data for fdtd-2d.

- **Simulated Annealing.** Samples designs that are likely to improve upon prior measurements. Samples are increasingly focused as iterations progress [54]. Simulated annealing accounts for multiple dimensions inefficiently. Simulated annealing could use an energy function that accounts for multiple objectives, but as we discuss in Section 4, combining FPGA performance and cost metrics into a single objective is both challenging and inefficient. So, we evaluate simulated annealing to account for a single objective, latency. But even in this setting, simulated annealing cannot find low-latency, Pareto-optimal designs and Prospector-1D outperforms it.
- **Genetic Algorithm.** Samples design populations by using the fittest designs from the previous generation of samples [13]. Searches for Pareto optima based on latency and cost. We use DEAP [17] framework to implement our genetic algorithm. This genetic algorithm baseline accounts for multiple dimensions. The genetic algorithm is a natural fit when optimizing multiple objectives and, like Prospector, we evaluate it to optimize latency and multiple dimensions of FPGA usage.

For all heuristics, each search iteration includes (1) parameter selection, (2) design synthesis, and (3) performance and cost evaluation. The runtime per iteration is completely dominated by HLS and RTL generation, common to all heuristics. For fairness, we allocate heuristics the same number of search iterations (or equivalently, the same amount of wall-clock time) and compare the resulting design quality.

4.1 Visualizing Pareto Efficiency

We start evaluating Prospector by applying Prospector-1D. Figure 6(a) indicates that Prospector-1D explores the design space to minimize latency without regard for cost. Although most design samples lie within the 25th percentile from minimum latency (bottom), most also exceed the 25th percentile from minimum LUT usage (right); see dotted lines. This outcome arises from the narrow optimization objective. Figure 7(a) indicates that Prospector-1D identifies designs that converge toward optimal latency but not the optimal number of LUTs.

Designers often care about more than one objective (i.e., cost and latency). Architects sometimes account for two metrics by optimizing their product or putting constraints on one when optimizing the other [37], but defining constraints is a burden on the user. Figure 6(b) applies Prospector-1D on the LUT-latency product. As shown in Figure 7(b), product optimization is a fragile solution. LUT usage is reduced, but latency remains high when minimizing this fused metric. LUT usage dominates the product and optimization procedure. One could normalize and rescale metrics to overcome the range difference between latency and LUTs. Figure 6(c) applies Prospector-1D on

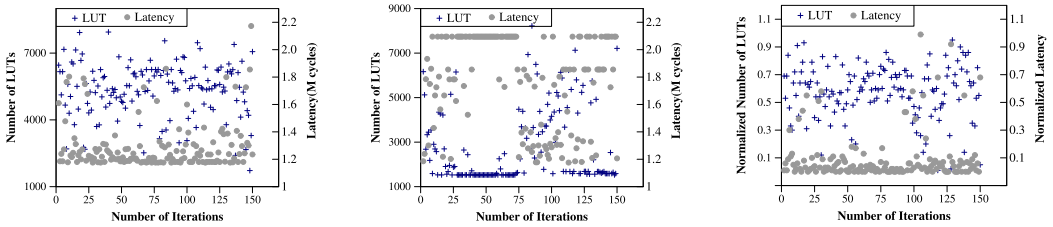


Fig. 7. Latency, LUT usage across iterations of Prospector-1D when minimizing (a) latency, (b) latency-LUT product, and (c) latency-LUT product with normalized, scaled measures. Data for fdtd-2d.

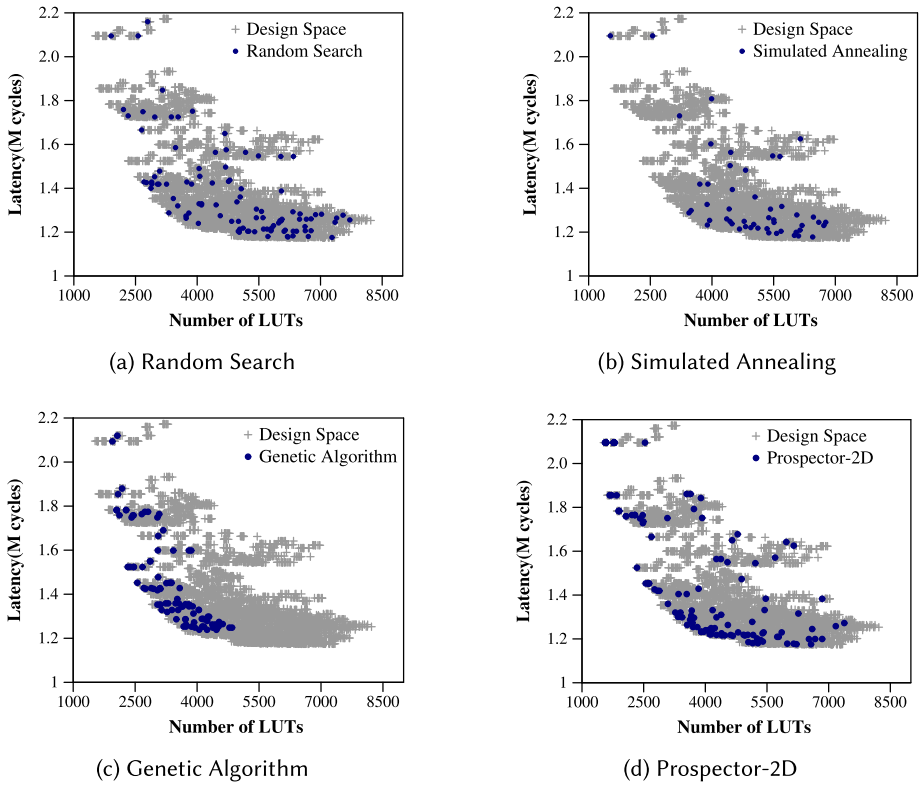


Fig. 8. Exploring fdtd-2d design space with random search, simulated annealing, genetic algorithm, and Prospector-2D.

the LUT-latency product with normalized measures. Figure 7(c) indicates that doing so does not solve the issue. Latency is reduced, but LUT usage remains high. These challenges increase with the number of dimensions in the design space.

Figures 8 and 9(a)–(c) indicate that alternative methods produce narrower trade-offs or Pareto sub-optimal designs. In Figure 8 for fdtd-2d, random search and simulated annealing are ineffective at identifying Pareto optima. Although the genetic algorithm identifies many low-latency and low-cost designs, measurements are concentrated in the lower-left quadrant and only partially reveal the Pareto frontier. In Figure 9 for fft, random search and simulated annealing fail even when

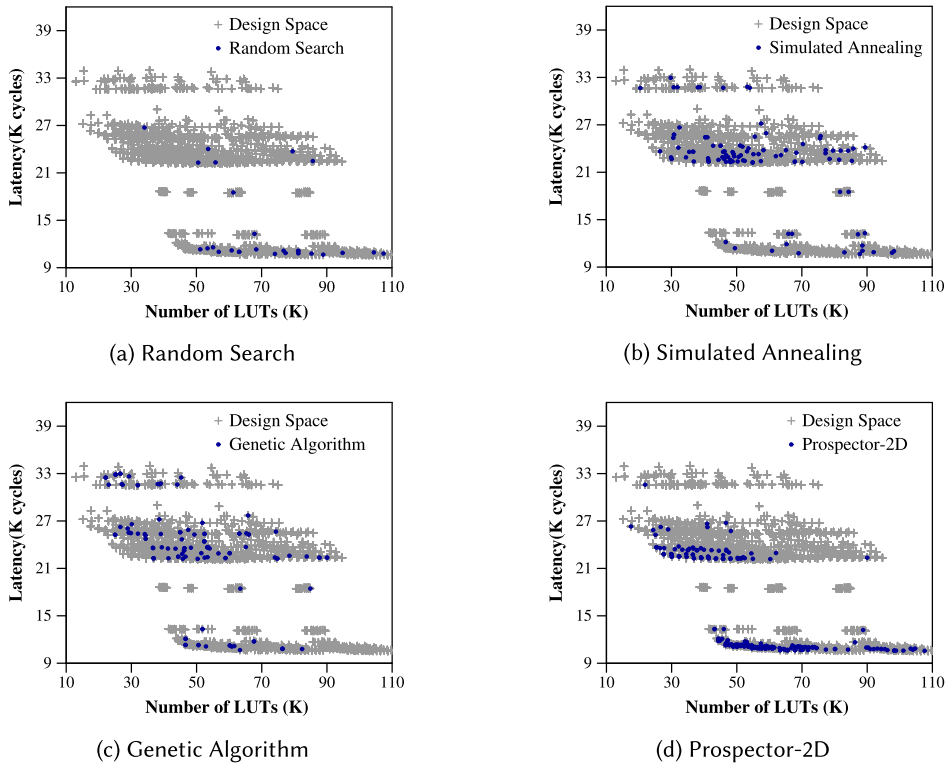
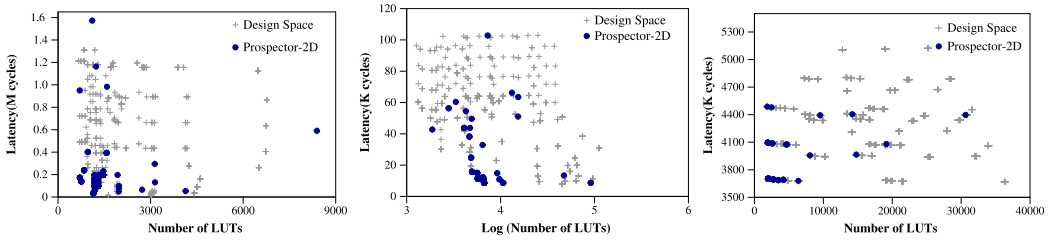


Fig. 9. Exploring fft design space with random search, simulated annealing, genetic algorithm, and Prospector-2D.

optimizing one objective. Random search and simulated annealing select very few points close to the Pareto frontier. Although the genetic algorithm identifies some points on the Pareto frontier, it samples disproportionately from high-latency, high-cost regions and misses Pareto optima with similar latency and lower costs. Prospector overcomes this limitation when modeling and optimizing multiple objectives simultaneously.

We evaluate the ability of design frameworks to reveal the Pareto frontier, which is essential to reasoning about design trade-offs. A good Pareto frontier identifies a broad spectrum of designs for which no other design improves one metric without harming another. We find that popular techniques—simulated annealing, random search, and genetic algorithm—do not accurately describe the Pareto frontier and do not reveal efficient trade-offs. These techniques converge to optima poorly or slowly, get stuck in local minima, or restrict the optimization to only parts of the design space. We find that Prospector can navigate multiple objectives, producing low-latency designs that use resources efficiently.

Figures 8(d) and 9(d) show how Prospector-2D for latency and LUT usage reveals the broad latency-LUT Pareto frontier for fdtd-2d and fft. Figure 10 also demonstrates Prospector-2D results for more benchmarks. One might hypothesize that, while Prospector-1D is insufficient, perhaps Prospector-2D suffices and there is little or no need for higher-dimensional optimization. However, although Prospector-2D outperforms Prospector-1D, its results are Pareto sub-optimal and fall short of coordinated analysis across all dimensions. Neglecting other FPGA resources in



(a) bbgemm (b) stencil (c) heat-3d

Fig. 10. Prospector-2D for latency and LUTs.

Table 3. Each Design is Pareto Optimal in 2D Analysis but, in the 5D Space, Is Actually Sub-optimal and Obscures Interactions between FPGA Resource Types

Design	2D Pareto	FFs	LUTs	DSPs	Cycles (M)
1	Latency-DSPs	4,926	7,289	32	1.175
2	Latency-LUTs	2,927	5,048	32	1.185
3	Latency-FFs	3,199	5,151	32	1.180
4	Latency-FFs	2,768	4,808	32	1.204
5	Latency-DSPs	3,810	5,536	20	1.194
6	Latency-LUTs	2,659	4,670	32	1.214
7	Latency-FFs	2,620	4,108	20	1.220

Data for fdtd-2d.

Prospector-2D results in missing sophisticated resource interactions and opportunities to use the FPGA more efficiently.

Table 3 details these limitations by presenting Prospector-2D's "Pareto-optimal" designs, which do not actually satisfy criteria for optimality and miss interactions between resources in the 5D space. First, Prospector-2D misses Pareto optima within the neglected dimensions of the 5D space. We can find designs that incur the same costs for one resource but lower costs for another resource. For example, Design 5 is not Pareto optimal because Design 7 performs nearly identically and uses the same number of DSPs, but reduces the number of LUTs and FFs by 25% and 31%, respectively. We consider designs that differ in latency by less than 50K cycles or 0.5ms to be equally good. Design 5 is Pareto dominated, yet Prospector-2D discovers it when optimizing latency and DSP cost while neglecting other resources. Design 7 is truly Pareto optimal and discovered by Prospector.

Second, Prospector-2D misses opportunities to substitute and exchange resources in the pursuit of performance. Substitution effects are important for FPGAs, because competition for shared resources may require flexible resource requests for an accelerator. For example, Designs 5 and 6 illustrate the possibility of substituting DSPs for LUTs and FFs. We could reduce the number of DSPs by 37% and increase the number of LUTs and FFs by 18% and 43% with nearly equal latency. Optimization in higher dimensions is more likely to discover substitutability, mitigating resource bottlenecks and revealing multiple paths to the same performance.

Finally, Prospector-2D misses complementary resource demands that require coordinated allocation. Points in the 2D Pareto frontiers indicate that LUTs and FFs are neither substitutes nor independent. LUTs and FFs are often used in related proportions such that if LUT usage changes, so does FF usage. Comparing Design 1 to Designs 2, 3, and 4, we find that LUTs and FFs are reduced together by 30% to 40%. Comparing Design 5 to Designs 6 and 7, we find that LUTs and FFs are

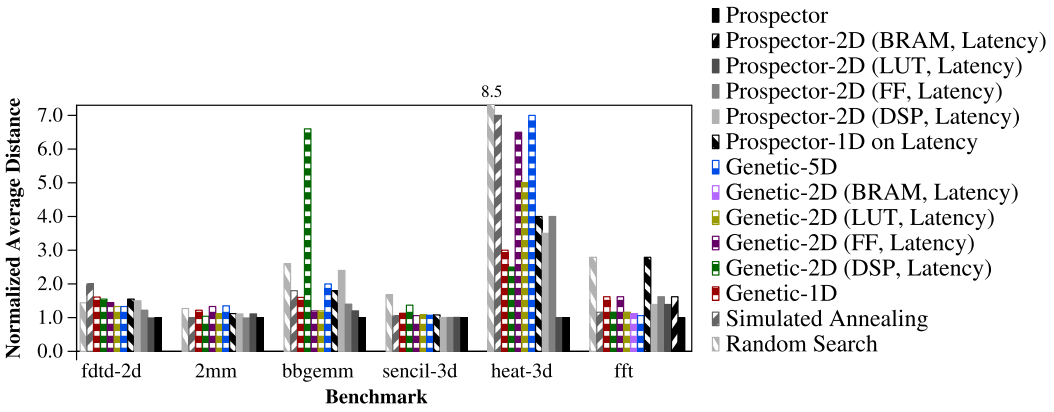


Fig. 11. Distances between golden and estimated Pareto frontiers, normalized to that from Prospector.

reduced together by 16% to 32%. These conclusions are missing from 2D analysis and require 5D analysis.

4.2 Quantifying Pareto Efficiency

We assess goodness by measuring the distance between two Pareto frontiers. We calculate the Euclidean distance from every design on the source frontier to the closest point on the destination frontier, producing a number of distances equal to the number of design points in the source. Note that each design point is represented by a five-dimensional vector that quantifies latency and usage for four FPGA resources. Our measures build on related work that explored several indicators to evaluate multi-dimensional Pareto frontiers [6].

Other simpler metrics such as counting the number of Pareto-optimal designs found could be misleading. Each design space has several points with similar efficiency or cost. Even if a search heuristic does not find the Pareto optimum, it may find other designs nearly as good. Counting exact matches is far too conservative and neglects the value of finding good designs that are slightly sub-optimal. One might define a threshold that specifies how close a design must be to the Pareto frontier in order to be counted, but defining appropriate thresholds for diverse benchmarks and design spaces is challenging.

When measuring distances, the source is the golden frontier identified by exhaustive evaluation of the design space, and the destination is an estimated frontier identified by Prospector or other heuristics. For each point on the golden frontier, we measure the shortest distance to any point on the estimated frontier. We mitigate the inconsistency between ranges measured for latency and FPGA resource usage by normalizing values so that they are in [0,1].

Figure 11 compares each heuristic by showing the distance of its designs to the golden Pareto frontier, which reflects the best designs from exhaustive search.[4] We normalize distances to those for Prospector to accommodate multiple scales and visualize results more effectively.

Prospector most accurately reveals the Pareto frontier by reporting the shortest distances to the golden frontier. The average normalized distance for alternative approaches is 1.74, meaning that alternative approaches find Pareto frontiers that are 1.74× more distant from the golden frontier than Prospector's. Conversely, Prospector's distance from the golden frontier is 0.57× that of the average distance of the alternatives. Optimization in fewer dimensions or using alternative

[4]HLS automatically allocates FFs and BRAMs, sometimes using only FFs. The BRAM-latency bar is shown only for fft, which required BRAMs.

Table 4. Optima from Other Algorithms Use More Resources Than Prospector, Given the Same Target Latency and Budget for Optimization Time

Scheme	Low Latency	Medium Latency	High Latency
Prospector 2D Latency-LUT	(0, 13, 8)	(0, 64, 38)	(0, 35, 25)
Prospector 2D Latency-FF	(0, 13, 8)	(25, 32, 16)	(0, 22, 15)
Prospector 2D Latency-DSP	(25, 86, 65)	(25, 55, 32)	(0, 138, 89)
Prospector 1D	(25, 74, 56)	(0, 134, 82)	(0, 191, 123)
Genetic Algorithm 5D	(25, 20, 9)	(0, 42, 19)	(0, 24, 8)
Genetic Algorithm 2D Latency-LUT	(25, 31, 18)	(0, 20, 12)	(0, 32, 18)
Genetic Algorithm 2D Latency-FF	(25, 26, 23)	(0, 7, -1)	(0, 32, 22)
Genetic Algorithm 2D Latency-DSP	(20, -10, -2)	(25, 28, 20)	(150, 59, 60)
Genetic Algorithm 1D	(25, 20, 11)	(0, 0, 1)	(0, 15, 8)
Simulated Annealing	(25, 68, 50)	(25, 180, 93)	(150, 201, 157)
Random Search	(25, 16, 13)	(0, 17, 13)	(0, 53, 39)

Results shown are percentage increases in resource usage (%DSP, %FF, %LUT). fdtd-2d benchmark.

heuristics is less accurate and reports greater distance to the golden Pareto frontier. Although Prospector-2D appears to produce accurate frontiers when visualizing 2D projections, distances between its 2D frontiers and the golden reference in 5D could be high.

In Figure 11, for heat-3d, Prospector's advantage is substantial due to the design space shape and structure. Relative to other benchmarks, heat-3d's design space is characterized by discrete regions. Designs are farther from each other and no dense cluster of similar designs exists. Therefore, when a search heuristic selects a sub-optimal design, the design's distance to the golden frontier is larger. Alternative heuristics make inefficient choices and incur higher costs for each mistakenly selected design from the space, missing opportunities to explore and model the space accurately.

Even though distances are the most important metric, they can be difficult to interpret. Table 4 shows how Prospector often identifies designs that use far fewer FPGA resources than those identified by other methods. The table compares resource usage for three performance targets (low, medium, high) that correspond to the 25th, 50th, and 75th percentiles in the golden frontier's latency distribution. For example, Prospector-1D's designs require up to 191% more resources than Prospector's. These points illustrate the likelihood of considerable resource inefficiencies when designing with Prospector's alternatives. Alternative approaches require 35% more FPGA resources than Prospector, when overheads are averaged over resource types and heuristics in Table 4.

5 CASE STUDY IN HETEROGENEITY

Prospector has many applications, and in this case study we illustrate how Prospector can be used to aid a system in using an allocation of FPGA resources to accelerate a computational kernel. Our case study explores classic questions in the design of large-small systems and the balance between serial performance and parallel throughput. Our study of FPGA-based accelerators is analogous to those that explore heterogeneous processor cores [19, 28, 29, 45].

Problem Statement. How should a computational kernel use an allocation of FPGA resources? Answering this question is challenging for two reasons. First, the kernel must choose between several accelerator implementations. Prospector supports this choice by exploring the design space and quantifying trade-offs between performance and FPGA resource usage. By revealing the Pareto frontier, Prospector focuses the kernel's choices to the most efficient implementations. In this case study, each kernel considers two accelerators, one that is faster and uses more resources (denoted large) and another that is slower and uses fewer resources (denoted small). We obtain these designs

Table 5. Latency and Resource Usage Trade-offs between Small and Large Designs from the Pareto Frontier

Benchmark	Latency (cycles)	DSP	BRAM	FF	LUT
fdtd-2d (pareto-optimal large)	1175242	32	0	4317	6642
fdtd-2d (pareto-optimal small)	2095242	8	0	911	1527
2mm (pareto-optimal large)	230044	168	0	62565	49129
2mm (pareto-optimal small)	262404	17	0	5207	3917
bbgemm (pareto-optimal large)	16389	48	0	1713	1629
bbgemm (pareto-optimal small)	1212417	3	0	331	664
stencil (pareto-optimal large)	7906	795	0	17617	45121
stencil (pareto-optimal small)	96314	6	0	518	1268
heat-3d (pareto-optimal large)	3670301	8	0	52363	36327
heat-3d (pareto-optimal small)	4485661	0	0	3620	1864
fft (pareto-optimal large)	10348	1861	148	80771	172428
fft (pareto-optimal small)	32556	114	16	11101	12990

by defining deadlines of latency on the Pareto frontier. Table 5 reports latency and resource usage trade-offs between small and large accelerators from the Pareto frontier of our benchmarks.

Second, the kernel must balance the tension between a single task's performance versus multiple tasks' throughput. This balance is increasingly important when FPGAs serve a stream of requests for acceleration from multiple processor cores. Ideally, with unlimited FPGA resources, each processor core would be paired with a large accelerator so that a task would always find a capable accelerator ready and available. With FPGA resource constraints, however, such resource allocations are infeasible.

In practice, we should implement a few large accelerators when few tasks are expected. And we should implement many small accelerators when task arrival rates are high, thereby reducing the risk of tasks arriving when all accelerators are busy. In this setting, heterogeneous architectures can improve system performance by switching between large and small configurations across time (temporal heterogeneity) or by instantiating a mix of large and small accelerators at the same time (spatial heterogeneity).

5.1 System Model

Task Execution. The system is organized around a queue of tasks, each of which requests computation on a processor core and an FPGA accelerator. When a core finishes its previous task, it dequeues the next task. The core computes for the task until it reaches a region targeted for acceleration (e.g., matrix multiply). Then, the core requests support from the most capable accelerator available. It sends data and control signals to the FPGA and waits for results. Finally, the core uses the results to complete its task. When all relevant accelerators are serving other cores, the region targeted for acceleration must execute on the general-purpose core. We model diverse task queues to generate varying demands on FPGA accelerators. We populate the task queue in two steps. First, we specify a benchmark of interest, determine the frequency (from 10% to 100%) with which it appears in the queue, and place the corresponding number of tasks at random locations in the queue. Second, we populate the remaining queue entries uniformly at random from our benchmark suite. Regardless of task intensities, all queued tasks are assumed to have already arrived, and throughput is measured as the number of tasks completed per second. The queued tasks ensure that the

processor cores are fully utilized yet exercise the FPGA with requests for accelerators separated by diverse inter-arrival times. We use a queue of 100,000 tasks in our experiments.

System Architecture. We consider a chip with eight out-of-order x86 cores and an integrated Xilinx Zynq FPGA. Each core and the FPGA has its own private L1 cache, and they all share a multi-banked L2 cache. The cores and FPGA communicate via cache-coherent shared memory (e.g., Intel HARP [20], IBM CAPI [61]), which uses a MESI two-level protocol and a mesh topology. Such tight integration reduces communication costs due to faster last-level cache accesses and simplifies the programming model when compared to DMA-based systems [9].

System Simulation. We evaluate system throughput using hybrid simulation [40]. First, we simulate one core and an integrated FPGA. We use PAAS, which extends gem5 with FPGA models [5, 34]. PAAS takes as inputs the CPU's x86 executable and the FPGA's Verilator executable [60], which compiles Verilog for Prospector's selected design and estimates performance. PAAS reports task performance without acceleration and with heterogeneous acceleration. In addition, synthesis reports describe each accelerator's FPGA resource usage. We record these performance and area statistics in the Single Core Performance Table (SCPT).

In each experiment, the workload is executed with a particular configuration of small and large accelerators programmed onto the FPGA. We do not reconfigure the FPGA dynamically. Rather, the mix of accelerators is programmed once and executes all tasks in the workload. Across experiments, we vary the combination of small-large accelerators and the workload intensities to measure the effect on throughput.

Second, we model multi-core behavior with a discrete event simulator (DES). Upon initialization, all cores and accelerators are idle and available for computation. In each time step, the DES dequeues and assigns the next task to an available core and its most capable accelerator. If no accelerator is available, the task computes on the core alone. The DES marks the time when cores and accelerators become available (i.e., current time plus recorded execution time from the SCPT's corresponding entry).

The DES efficiently captures first-order effects. The simulator details the assignment of tasks to computational resources, recording the number and type of tasks executed on the small and large accelerators as well as those executed solely on the core because no accelerator was available. Furthermore, the DES reports the start and end time for every task, which permits us to measure task throughput from varied accelerator and FPGA configurations.

5.2 Opportunities for Heterogeneity

We explore three scenarios in which having heterogeneous accelerators offers benefits. Naturally, if heterogeneous accelerators are useful, then Prospector's ability to efficiently find Pareto-optimal accelerators is beneficial. Without loss of generality and for clarity, these examples are limited to small amounts of heterogeneity. First, we motivate heterogeneity for FPGA accelerators. Second, we show that Prospector, which more accurately reveals Pareto trade-offs, produces greater benefits from heterogeneity when compared to alternative approaches. We present results for fdtd-2d and heat-3d; results for other kernels are similar.

Temporal Heterogeneity. Temporal heterogeneity describes an FPGA that reconfigures itself to use large or small accelerators, but not both, between workloads. Considering the FPGA size, it can hold either four large (4L) or eight small (8S) fdtd-2d accelerators. For heat-3d, the FPGA can hold either three large (3L) or eight small (8S) accelerators. Table 6 shows each task's end-to-end latency on the CPU, with the small FPGA accelerator, and with the large FPGA accelerator. The latencies for accelerators include the time to transfer data between the CPU and the accelerator.

Figures 12 and 13(a) plot throughput as a function of task intensity (i.e., percentage of queued tasks that demand acceleration for the same kernel). The figures illustrate trade-offs between serial

Table 6. Execution Time for fdtd-2d and Heat-3d Benchmarks on CPU, with Small FPGA Accelerator, and with Large FPGA Accelerator

Benchmark	Latency on CPU (sec)	Latency with Small Accelerator (sec)	Latency with Large Accelerator (sec)
fdtd-2d	102.5	67.63	47.88
heat-3d	4995.6	1727.8	1159.8

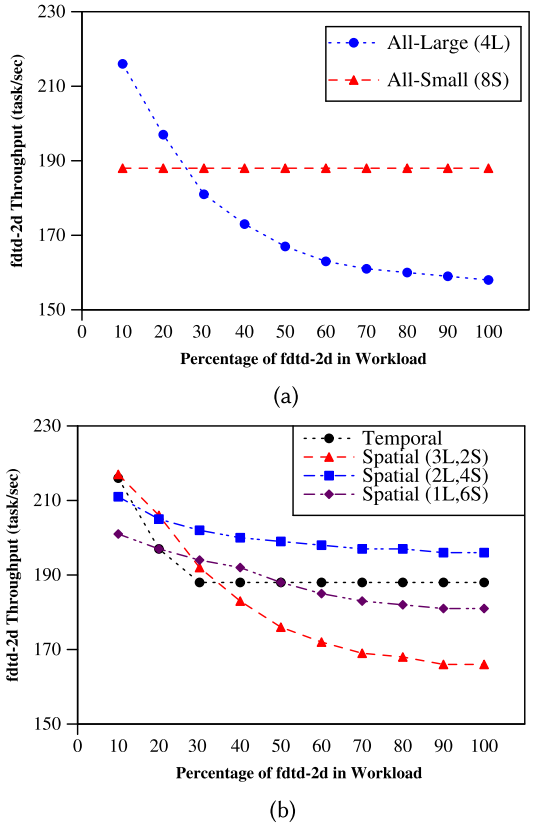


Fig. 12. Throughput given increasing workload intensity, measured by the percentage of queued tasks that demand the same accelerator type. Data for representative fdtd-2d workload.

performance and parallel throughput. Lower intensities generate less task parallelism, and the all-large configuration provides higher throughput via larger speedups. Higher intensities correspond to greater task parallelism, and the all-small configuration performs best. An FPGA that instantiates many small accelerators reduces the likelihood that a task dequeues, fails to find an available accelerator, and computes on the processor core. For throughput, accelerating all tasks moderately is preferable to accelerating a few tasks significantly. The figures' crossover points make the case for adapting to task arrival rates and switching between all-large and all-small.

The frequency of FPGA reconfiguration, between all-small or all-large accelerators, depends on workload mix and intensity. Reconfiguration is required only when mix and intensity change significantly. Given that our task queue involved a large number of tasks (100,000), the total execution time of each queue could take days. Thus, we consider reconfiguration for phases that are

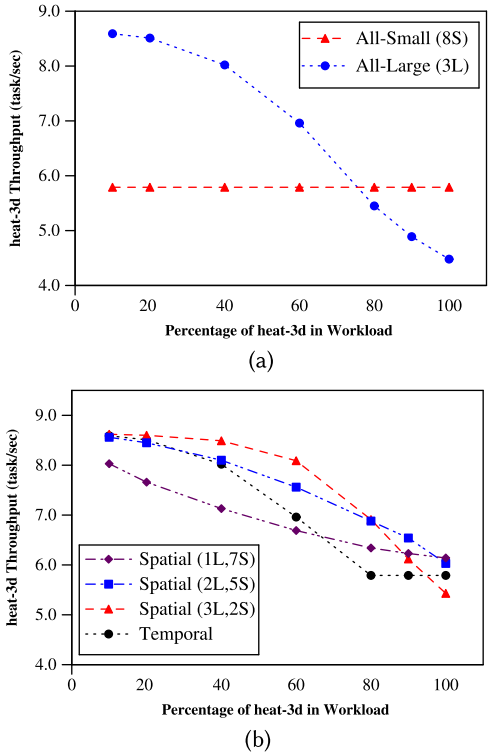


Fig. 13. Throughput given increasing workload intensity, measured by the percentage of queued tasks that demand the same accelerator type. Data for representative heat-3d workload.

long enough that reconfiguration latency (100s of milliseconds) is amortized over long workload phases.

Spatial Heterogeneity. Spatial heterogeneity describes an FPGA that simultaneously deploys a mix of large and small accelerators. Figures 12 and 13(b) show the benefits of spatial heterogeneity, compared to a baseline (denoted "Temporal") that optimally deploys the all-large or all-small configuration based on task intensity. At low task intensities, spatially heterogeneous mixes that favor large accelerators perform best. At high intensities, those that favor small accelerators perform best. Spatially heterogeneous mixes outperform all-large or all-small configurations by as much as 24% when 30% to 100% of queued tasks demand the same accelerator for the fdtd-2d kernel. For heat-3d, heterogeneous mixes outperform all-large or all-small configurations by as much as 20% when 60% to 90% of the queued tasks demand the same accelerator.

Differing FPGA Budgets. Heterogeneous accelerators can also be useful when targeting FPGAs with different resources. Table 7 (top) holds fdtd-2d intensity at 80%, varies FPGA resource budgets, uses Prospector to determine the accelerator mixes that fit, and reports throughput. The large and small accelerators are defined as the designs that achieve the 25th and 75th latency percentiles on the Pareto frontier. We normalize throughput to that from eight large accelerators supporting eight processor cores to show throughput penalties due to resource limits. As FPGA resources decrease, from Budget 1 to 8, the number of (spatially) heterogeneous mixes that fit also decreases. We highlight the mix that maximizes throughput for each budget.

Given generous FPGA resources, the system accommodates all-large configurations. However, with scarce resources in a multi-programmed FPGA, heterogeneity's benefits are significant. As

Table 7. Throughput, Normalized to that from Eight Large Accelerators, Given Decreasing FPGA Resource Budgets

Budget 1		Budget 2		Budget 3		Budget 4		Budget 5		Budget 6		Budget 7		Budget 8	
(8L,0S)	**1**	(7L,0S)	0.92	(6L,0S)	0.85	(5L,0S)	0.78	(4L,0S)	0.71	(3L,0S)	0.64	(2L,0S)	0.57	(1L,0S)	0.50
(7L,1S)	0.98	**(6L,2S)**	**0.96**	(5L,2S)	0.89	(4L,2S)	0.82	(3L,2S)	0.75	(2L,2S)	0.68	(1L,2S)	0.60	**(0L,2S)**	**0.53**
(6L,2S)	0.96	(5L,3S)	0.94	**(4L,4S)**	**0.92**	(3L,5S)	0.85	(2L,6S)	0.78	(1L,6S)	0.71	**(0L,6S)**	**0.64**		
(5L,3S)	0.94	(4L,4S)	0.92	(3L,5S)	0.90	**(2L,6S)**	**0.88**	(1L,6S)	0.81	**(0L,6S)**	**0.74**				
(4L,4S)	0.92	(3L,5S)	0.90	(2L,6S)	0.88	(1L,7S)	0.86	**(0L,8S)**	**0.85**						
(3L,5S)	0.90	(2L,6S)	0.88	(1L,7S)	0.86	(0L,8S)	0.85								
(2L,6S)	0.88	(1L,7S)	0.86	(0L,8S)	0.85										
(1L,7S)	0.86	(0L,8S)													
(0L,8S)	0.85														

Budget 1		Budget 2		Budget 3		Budget 4		Budget 5		Budget 6		Budget 7		Budget 8	
(6L,2S)	**0.92**	**(6L,0S)**	**0.85**	**(5L,1S)**	**0.82**	**(4L,1S)**	**0.75**	**(3L,1S)**	**0.68**	**(2L,1S)**	**0.61**	**(1L,1S)**	**0.53**	**(0L,1S)**	**0.46**
(5L,3S)	0.89	**(5L,2S)**	**0.85**	(4L,2S)	0.78	(3L,2S)	0.71	(2L,2S)	0.64	(1L,2S)	0.57	(0L,2S)	0.50		
(4L,4S)	0.85	**(4L,4S)**	**0.85**	(3L,4S)	0.78	(2L,4S)	0.71	(1L,3S)	0.60	(0L,3S)	0.53				
(3L,5S)	0.82	(3L,5S)	0.82	(2L,5S)	0.75	(1L,5S)	0.68	(0L,5S)	0.60						
(2L,6S)	0.78	(2L,6S)	0.78	(1L,6S)	0.71	(0L,6S)	0.64								
(1L,7S)	0.74	(1L,7S)	0.74	(0L,7S)	0.67										
(0L,8S)	0.71	(0L,8S)	0.71												

Data for representative fdtd-2d workload. Bold entries are the best-performing allocation(s) at a given budget from Prospector (top) and Prospector-2D on latency and flip-flops (bottom).

the budget constrains the number of large accelerators, the system improves performance by substituting a few large accelerators with several smaller ones. From Budget 4 onwards, all-small outperforms all-large, and there is a trade-off between many small accelerators and a few big ones. For example, in Budget 4, the best spatial mix outperforms the all-large and all-small configurations by 10%.

Table 7 (bottom) presents the same experiment with Prospector-2D. Its large and small accelerators are larger than Prospector's, such that fewer accelerators fit within each budget level. Moreover, Prospector-2D's designs perform worse. Thus, using small and large accelerators found by an alternative approach leads to lower throughput under each budget.

In summary, the performance benefits of heterogeneity, both temporal and spatial, show the importance of diverse RTL libraries that support flexible FPGA configurations. We expect performance benefits to increase further with diverse RTL libraries that look beyond the two large and small configurations and accommodate a richer set of performance and cost trade-offs.

6 RELATED WORK

Black-Box Optimization: Black-box optimizations like Bayesian optimization and reinforcement learning are interesting candidates in approaching design space exploration problems for objective functions with an unknown closed-form relation to inputs. Bayesian optimization has been applied to hardware design in two separate studies. First, Bayesian optimization tunes directives to minimize latency [37]. This work explores only the placement, but not configuration, of HLS directives due to its limits in design encoding. Furthermore, it fuses metrics into a single objective (i.e., latency-LUT product), which we show is problematic, to leverage standard data collection procedures. Second, Bayesian optimization tunes neural network workloads, optimizing both network hyperparameters and some continuous hardware parameters (e.g., operand bit width) [52].

This study employs Aladdin's pre-RTL models [57]. It does not consider the HLS directives needed to instantiate accelerators on reconfigurable hardware.

Bayesian optimization (BO) and reinforcement learning (RL) share some similarities in black-box optimization—both are frameworks for online and incremental learning. BO is composed of two main components. First, models (e.g., Gaussian processes) estimate probability distributions on outcomes based on past observations to optimize design. Second, the acquisition function (e.g., expected improvement) determines how data is acquired to update models. RL is also composed of two parts. First, models (e.g., Q-table, neural network) estimate outcomes based on past observations to optimize design and policy. Second, strategies (e.g., experience replay) determine how data is acquired to update models. We use BO instead of RL for several reasons. BO's Gaussian processes and acquisition functions are a natural fit for design space exploration and have been applied extensively in other domains. In contrast, RL's policy and reward functions would need customization for the HLS design space. Formulating HLS design space exploration as an RL problem could be an interesting future direction.

Design Space Exploration: First, statistical learning constructs efficient predictive models that act as surrogates for unwieldy design flows. Machine learning has been used in a variety of domains such as compiler auto-tuning for optimization purposes [3, 64]. Architects also train models from sampled measurements [14, 25, 35, 38, 41, 55, 66]. Regression can predict performance and power from sparse simulations of microarchitectural design spaces [30, 31]. These methods sample the parameter space, evaluate the corresponding design, and learn models for design quality.

Machine learning models for design space exploration require training data and predictive accuracy. Prior work proposed using regression trees to predict performance and area from discrete HLS directive configurations for ASICs [35]. Acquiring training data and iteratively refining the model requires a larger number of synthesis iterations, which will increase the search time. In addition to the training challenge, our design space is harder to model given how interactions between directive placement and configuration dictate multiple measures of FPGA cost (as discussed in Section 4).

Second, pre-RTL frameworks support early-stage ASIC design (e.g., Aladdin [57]), but the architect must still perform design space exploration, relying on expert design or tuned directives to identify efficient implementations. Our study differs from Aladdin in its focus on intelligent design space exploration, on generating RTL, and on instantiating accelerators on FPGAs.

Third, heuristics search the design space defined by tunable knobs. Random search (RS) samples designs randomly [4]. Some heuristics, such as simulated annealing (SA) [54, 58] and gradient descent, judiciously sample designs likely to improve upon prior measurements but may become stuck in local minima [33, 52, 59]. Genetic algorithm (GA) uses the fittest designs in a population of samples to produce the next generation of samples [1, 2, 13, 15, 32, 46]. Prospector outperforms these techniques for HLS and FPGAs' large parameter spaces.

Finally, analytical models explore simpler parameter spaces but do not extend easily to directive placement and configuration for multiple interdependent code targets. Some models focus on a restricted design space such as memory organization [18, 62]. Others study directives for a single code target [68]. Yet others optimize directives to reduce usage for specific FPGA resource types such as BRAMs and DSPs [67]. None of these approaches model directive placement and configuration for multiple HLS code targets and comprehensive FPGA design spaces. Researchers have also constructed lattices for HLS design space exploration [16]. Lattices are promising but thus far have considered only two objectives, latency and area, rather than the multiple objectives required for FPGA implementation. They have also been applied to configuring a limited number of directives at fixed code locations, leading to design space sizes that are only 20% of those in our study.

7 CONCLUSION

Prospector uses multi-dimensional Bayesian optimization to efficiently search large accelerator design spaces defined by HLS directives. Despite the plethora of possible HLS directive placements and configurations and despite the multiple metrics optimized, Prospector efficiently finds Pareto-optimal designs. Prospector is much more effective than alternative search heuristics. We highlight Prospector's usefulness by producing a heterogeneous mix of accelerators that balances latency and throughput as well as responds to evolving workload demands and resource allocations.

REFERENCES

[1] G. Ascia, V. Catania, A. Di Nuovo, M. Palesi, and D. Patti. 2007. Efficient design space exploration for application specific systems-on-a-chip. *Journal of Systems Architecture* 53, 10 (2007), 733–750.

[2] G. Ascia, V. Catania, and M. Palesi. 2005. A multiobjective genetic approach for system-level exploration in parameterized systems-on-a-chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24, 4 (2005), 635–645.

[3] A. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano. 2018. A survey on compiler autotuning using machine learning. *ACM Computing Surveys (CSUR)* 51, 5 (2018), 1–42.

[4] J. Bergstra and Y. Bengio. 2012. Random search for hyper-parameter optimization. *Journal of Machine Learning Research* 13, 1 (2012), 281–305.

[5] N. Binkert, B. Beckmann, G. Black, S. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. Hill, and D. Wood. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

[6] P. Bosman and D. Thierens. 2003. The balance between proximity and diversity in multiobjective evolutionary algorithms. *IEEE Transactions on Evolutionary Computation* 7, 2 (2003), 174–188.

[7] E. Brochu, V. Cora, and N. De Freitas. 2010. A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599* (2010).

[8] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. 2014. DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*.

[9] Y. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei. 2016. A quantitative analysis on microarchitectures of modern CPU-FPGA platforms. In *Proc. of the 53rd Annual Design Automation Conference*.

[10] E. Chung, P. Milder, J. Hoe, and K. Mai. 2010. Single-chip heterogeneous computing: Does the future include custom logic, FGPAs, and GPGPUs? In *Proc. International Symposium on Microarchitecture (MICRO'10)*.

[11] J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang. 2006. Platform-based behavior-level and system-level synthesis. In *Proc. International SOC Conference (SOCC'06)*.

[12] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. 2011. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 4 (2011), 473–491.

[13] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (2002), 182–197.

[14] C. Dubach, T. Jones, and M. O'Boyle. 2008. Microarchitectural design space exploration using an architecture-centric approach. In *Proc. International Symposium on Microarchitecture (MICRO'08)*.

[15] S. Eyerman, L. Eeckhout, and K. De Bosschere. 2006. Efficient design space exploration of high performance embedded out-of-order processors. In *Proc. Design Automation & Test in Europe Conference*, Vol. 1. IEEE, 1–6.

[16] L. Ferretti, G. Ansaloni, and L. Pozzi. 2018. Lattice-traversing design space exploration for high level synthesis. In *2018 IEEE 36th International Conference on Computer Design (ICCD'18)*. IEEE, 210–217.

[17] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. 2012. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research* 13, 1 (2012), 2171–2175.

[18] A. Ghosh and T. Givargis. 2004. Cache optimization for embedded processor cores: An analytical approach. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 9, 4 (2004), 419–440.

[19] P. Greenhalgh. 2011. Big. little processing with arm cortex-a15 & cortex-a7. *ARM White Paper*.

[20] P. Gupta. 2016. Accelerating datacenter workloads. In *Proc. International Conference on Field Programmable Logic and Applications (FPL'16)*.

[21] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. Lee, S. Richardson, C. Kozyraki, and M. Horowitz. 2010. Understanding sources of inefficiency in general-purpose chips. In *Proc. International Symposium on Computer Architecture (ISCA'10)*.

[22] D. Hernández-Lobato, J. Hernández-Lobato, A. Shah, and R. Adams. 2016. Predictive entropy search for multi-objective Bayesian optimization. In *Proc. International Conference on Machine Learning*.

[23] J. Hoe. 2016. Technical perspective: FPGA compute acceleration is first about energy efficiency. *Communications of the ACM* 59, 11 (2016), 113–113.

[24] M. Huang, D. Wu, C. Yu, Z. Fang, M. Interlandi, T. Condie, and J. Cong. 2016. Programming and runtime support to blaze FPGA accelerator deployment at datacenter scale. In *Proc. ACM Symposium on Cloud Computing*.

[25] P. Joseph, K. Vaswani, and M. Thazhuthaveetil. 2006. A predictive performance model for superscalar processors. In *Proc. International Symposium on Microarchitecture (MICRO'06)*.

[26] J Koenig, D. Biancolin, J. Bachrach, and K. Asanovic. 2017. A hardware accelerator for computing an exact dot product. In *Proc. 24th Symposium on Computer Arithmetic (ARITH'17)*.

[27] D. Koeplinger, R. Prabhakar, Y. Zhang, C. Delimitrou, C. Kozyrakis, and K. Olukotun. 2016. Automatic generation of efficient accelerators for reconfigurable hardware. In *Proc. International Symposium on Computer Architecture (ISCA'16)*.

[28] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. 2003. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proc. International Symposium on Microarchitecture (MICRO'03)*.

[29] R. Kumar, D. Tullsen, P. Ranganathan, N. Jouppi, and K. Farkas. 2004. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Proc. International Symposium on Computer Architecture (MICRO'04)*.

[30] B. Lee and D. Brooks. 2006. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*.

[31] B. Lee and D. Brooks. 2007. Illustrative design space studies with microarchitectural regression models. In *Proc. International Symposium on High-Performance Computer Architecture (HPCA'07)*.

[32] B. Lee and D. Brooks. 2008. Efficiency trends and limits from comprehensive microarchitectural activity. In *Proc. International Conference on Architectural Support Programming Languages and Operating Systems (ASPLOS'08)*.

[33] B. Lee and D. Brooks. 2008. Roughness of microarchitectural design topologies and its implications for optimization. In *Proc. International Symposium on High Performance Computer Architecture (HPCA'08)*.

[34] T. Liang, L. Feng, S. Sinha, and W. Zhang. 2017. PAAS: A system level simulator for heterogeneous computing architectures. In *Proc. International Conference on Field Programmable Logic and Applications (FPL'17)*.

[35] H. Liu and L. Carloni. 2013. On learning-based methods for design-space exploration with high-level synthesis. In *Proc. Design Automation Conference (DAC'13)*.

[36] X. Liu, Xingyu, and Y. Deng. 2014. Fast radix: A scalable hardware accelerator for parallel radix sort. In *Proc. International Conference on Frontiers of Information Technology (FIT'14)*.

[37] C. Lo and P. Chow. 2016. Model-based optimization of high level synthesis directives. In *Proc. International Conference on Field Programmable Logic and Applications (FPL'16)*.

[38] A. Mahapatra and B. Schafer. 2014. Machine-learning based simulated annealer method for high level synthesis design space exploration. In *Proc. Electronic System Level Synthesis Conference (ESLsyn'14)*.

[39] A. Mehrabi, A. Manocha, B. Lee, and D. Sorin. 2020. Prospector: Synthesizing efficient accelerators via statistical learning. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE'20)*. IEEE, 151–156.

[40] D. Meisner, J. Wu, and T. Wenisch. 2012. BigHouse: A simulation infrastructure for data center systems. In *Proc. International Symposium on Performance Analysis of Systems and Software (ISPASS'12)*.

[41] P. Meng, A. Althoff, Q. Gautier, and R. Kastner. 2016. Adaptive threshold non-pareto elimination: Re-thinking machine learning for system level design space exploration on FPGAs. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE'16)*. IEEE, 918–923.

[42] K. Murphy. 2012. *Machine Learning: A Probabilistic Approach*. MIT Press.

[43] S. Murray, W. Floyd-Jones, Y. Qi, G. Konidaris, and D. Sorin. 2016. The microarchitecture of a real-time robot motion planning accelerator. In *Proc. International Symposium on Microarchitecture (MICRO'16)*.

[44] R. Nane, V. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. 2015. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 10 (2015), 1591–1604.

[45] Nvidia. 2011. Variable SMP- A multi-core CPU architecture for low power and high performance. *Whitepaper*. Retrieved from http://www.nvidia.com.

[46] M. Palesi and T. Givargis. 2002. Multi-objective design space exploration using genetic algorithms. In *Proc. 10th International Symposium on Hardware/software Codesign*. ACM, 67–72.

[47] C. Pilato, P. Mantovani, G. DiGuglielmo, and L. Carloni. 2014. System-level memory optimization for high-level synthesis of component-based SoCs. In *Proc. International Conference on Hardware/Software Codesign and System Synthesis*.

[48] L. Pouchet. 2012. Polybench: The polyhedral benchmark suite. Retrieved from http://www.cs.ucla. edu/pouchet/software/polybench.
[49] A. Putnam, A. Caulfield, E. Chung, D. Chiou, and K. Constantinides. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proc. International Symposium on Computer Architecuture (ISCA'14)*.
[50] B. Rau. 1994. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. Interanational Symposium on Microarchitecture (MICRO'94)*.
[51] B. Reagen, R. Adolf, Y. Shao, G. Wei, and D. Brooks. 2014. Machsuite: Benchmarks for accelerator design and customized architectures. In *Proc. International Symposium on Workload Characterization (IISWC'14)*.
[52] B. Reagen, J. Hernández-Lobato, R. Adolf, M. Gelbart, P. Whatmough, G. Wei, and D. Brooks. 2017. A case for efficient accelerator design space exploration via Bayesian optimization. In *Proc. International Symposium on Low Power Electronics and Design (ISLPED'17)*.
[53] B. Reagen, Y. Shao, G. Wei, and D. Brooks. 2013. Quantifying acceleration: Power/performance trade-offs of application kernels in hardware. In *Proc. International Symposium on Low Power Electronics and Design (ISLPED'13)*.
[54] B. Schafer, T. Takenaka, and K. Wakabayashi. 2009. Adaptive simulated annealer for high level synthesis design space exploration. In *Proc. International Symposium on VLSI Design, Automation and Test (VLSI-DAT'09)*.
[55] B. Schafer and K. Wakabayashi. 2012. Machine learning predictive modelling high-level synthesis design space exploration. *IET Computers & Digital Techniques* 6, 3 (2012), 153–159.
[56] B. Shahriari, K. Swersky, Z. Wang, R. Adams, and N. De Freitas. 2015. Taking the human out of the loop: A review of Bayesian optimization. *Proceedings of the IEEE* 104, 1 (2015), 148–175.
[57] Y. Shao, B. Reagen, G. Wei, and D. Brooks. 2014. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *Proc. International Symposium on Computer Architecture (ISCA'14)*.
[58] K. Smith, R. Everson, J. Fieldsend, C. Murphy, and R. Misra. 2008. Dominance-based multiobjective simulated annealing. *IEEE Transactions on Evolutionary Computation* 12, 3 (2008), 323–342.
[59] J. Snoek, H. Larochelle, and R. Adams. 2012. Practical Bayesian optimization of machine learning algorithms. In *Proc. Advances in Neural Information Processing Systems (NIPS'12)*.
[60] W. Snyder. 2013. Verilator: Open simulation-growing up. *DVClub Bristol*.
[61] J. Stuecheli, B. Blaner, C. Johns, and M. Siegel. 2015. CAPI: A coherent accelerator processor interface. *IBM Journal of Research and Development* 59, 1 (2015), 7:1–7:7.
[62] R. Szymanek, F. Catthoor, and K. Kuchcinski. 2004. Time-energy design space exploration for multi-layer memory architectures. In *Proc. Conference on Design, Automation and Test in Europe-Volume 1*. IEEE Computer Society, 10318.
[63] C. Wang, L. Gong, Q. Yu, X. Li, Y. Xie, and X. Zhou. 2016. DLAU: A scalable deep learning accelerator unit on FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36, 3 (2016), 513–517.
[64] Z. Wang and M. O'Boyle. 2018. Machine learning in compiler optimization. *Proceedings of IEEE* 106, 11 (2018), 1879–1901.
[65] F. Winterstein, S. Bayliss, and G. Constantinides. 2013. High-level synthesis of dynamic data structures: A case study using Vivado HLS. In *Proc. International Conference on Field-Programmable Technology (FPT'13)*.
[66] S. Xydis, G. Palermo, V. Zaccaria, and C. Silvano. 2014. SPIRIT: Spectral-aware Pareto iterative refinement optimization for supervised high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34, 1 (2014), 155–159.
[67] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He. 2017. COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications. In *Proc. 36th International Conference on Computer-Aided Design*.
[68] G. Zhong, A. Prakash, Y. Liang, T. Mitra, and S. Niar. 2016. Lin-analyzer: A high-level performance analysis tool for FPGA-based accelerators. In *Proc. 53rd Annual Design Automation Conference*. ACM, 136.

Received February 2020; revised August 2020; accepted September 2020