Scalable Auto-Tuning of Synthesis Parameters for Optimizing High-Performance Processors

Matthew M. Ziegler[1], Hung-Yi Liu[2*], and Luca P. Carloni[2]
[1] IBM T. J. Watson Research Center, Yorktown Heights, NY, USA
[2] Department of Computer Science, Columbia University, NY, USA

ABSTRACT

Modern logic and physical synthesis tools provide numerous options and parameters that can drastically impact design quality; however, the large number of options leads to a complex design space difficult for human designers to navigate. By employing intelligent search strategies and parallel computing we can tackle this parameter tuning problem, thus automating one of the key design tasks conventionally performed by a human designer. In this paper we present a novel learning-based algorithm for synthesis parameter optimization. This new algorithm has been integrated into our existing autonomous parameter-tuning system, which was used to design multiple 22nm industrial chips and is currently being used for 14nm chips. These techniques show, on average, over 40% reduction in total negative slack and over 10% power reduction across hundreds of 14nm industrial processor macros while reducing overall human design effort. We also present a new higher-level system that manages parameter tuning of multiple designs in a scalable way. This new system addresses the needs of large design teams by prioritizing the tuning effort to maximize returns given the available compute resources.

1. INTRODUCTION

Modern logic and physical synthesis tools have advanced to the point where they can often achieve a quality of results similar to that of experienced human designers employing custom methodologies. These synthesis tools also have significant advantages compared to semi-custom / custom methodologies in terms of turn-around-time and human design effort. The improvements in quality of results and reductions in design cost have led to an industry shift to synthesis-centric methodologies for even the highest performance digital processor designs. However, this comes at the cost of an increased complexity. Modern tools present many options and parameters that can drastically impact the design quality. The result is a parameter design space that is complex for even experienced human designers to navigate and daunting for novice designers. Fortunately, this problem is well-suited for automated techniques, as long as intelligent *design space exploration (DSE)* algorithms are employed.

The industrial synthesis tool-flow we employ has over 1000 parameters [1]. These parameters span the logic and physical synthesis space and the control settings for modifying the synthesis steps, such as: logic decomposition, technology mapping, placement, estimated wire optimization, power recovery, area recovery, and/or higher effort timing improvement. The parameters also vary in data type (Boolean, integer, floating point, and string). Considering that an exhaustive search of only 20 Boolean-type parameters leads to over one million combinations, it is clear that intelligent search strategies are required.

* Hung-Yi Liu is now with the Intel Design Technology & Solutions Group, Hillsboro, OR, USA.

ISLPED '16, August 08-10, 2016, San Francisco Airport, CA, USA
 ISBN 978-1-4503-4185-1/16/08…$15.00
DOI: http://dx.doi.org/10.1145/2934583.2934620

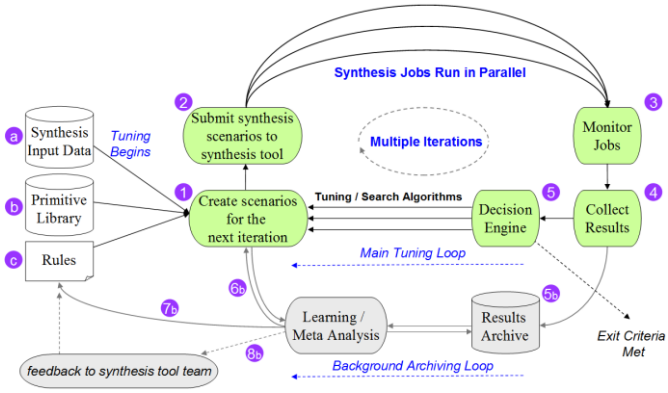

Figure 1. Architecture of the STS process, which employs a parallel and iterative tuning process to optimize macros [2].

In prior work [2], we showed how *SynTunSys (STS)*, a novel synthesis-parameter tuning system, can provide timing and power improvements for industrial high-performance processors, e.g., the IBM z13 mainframe system [3]. During the design of this processor synthesis parameter tuning was run on over 200 macros and provided, on average, a 60% improvement on internal macro slack, a 36% improvement in total negative slack, and a 7% power reduction after one pass of synthesis parameter tuning. These macros consisted of various types of digital logic spanning from datapath to finite state machines. Further, although our specific application was a high-performance server chip, any synthesizable digital logic can benefit from this approach.

STS is a program that adds an additional level of abstraction between synthesis tools and the human designer. STS takes control of the synthesis parameter tuning process, i.e., job submission, results analysis, and next-step decision making, automating a key portion of a human designer's decision process. Although the version of STS employed at the 22nm node was effective, VLSI design challenges continue to require strides in quality of results and design efficiency. Addressing these challenges led us to make the following new contributions:

Contribution 1: We developed an adaptive online learning algorithm for DSE and we integrated it into STS. This enhancement yields an additional 20% of total negative slack improvement beyond our original algorithm on 22nm macros.

Contribution 2: We applied the new enhanced version of STS to hundreds of macros for a new 14nm industrial processor, obtaining, an average reduction of over 40% in total negative slack and over 10% in power.

Contribution 3: We addressed the challenge of tuning multiple macros to achieve the best return-on-investment (ROI) given limited compute resources by developing the *STS Scheduler (STSS)*, which can manage many STS tuning runs to maximize the ROI of parameter tuning across multiple macros.

2. STS System Overview

As shown in Figure 1, STS consists of a main tuning loop that constructs synthesis scenarios consisting of synthesis parameter settings (Step (1)),

Table 1. An example of primitive names and primitive descriptions.

Primitive Name	Primitive Description
restruct_a	Logic restructuring to reduce area
restruct_t	Logic restructuring to improve timing
area_he	High effort area reduction
wireopt_t	Wire optimization for timing
wireopt_c	Wire optimization for congestion

submits and monitors synthesis jobs (2-3), analyzes the results (4), and iteratively refines the solutions (5). A second background loop archives the results of all runs from all macros, users, and projects. This archive is a database that can be mined for historical trends across projects and to provide feedback in terms of the performance of synthesis parameters. For a complete presentation of the STS system, please see reference [2].

Each STS iteration submits multiple scenarios in parallel to a compute cluster. Our cluster employs the Platform LSF workload management system, which is commercial software based on the Utopia project [4]. A user can typically submit ~50 scenarios in parallel, although this number can vary depending on the specific macro's compute requirements, i.e., memory, CPU count, and runtime.

STS targets the design flow steps that convert RTL to a placed layout, based on timing information derived with estimated wires and congestion analysis. Thus the synthesis input is an RTL description, a physical abstract view providing macro boundaries, pin locations, and timing assertions (Figure 1 (a)). In this section we consider tuning only a single macro. Section 5 expands the discussion to tuning multiple macros.

The industrial design cycle for a high-performance processor typically requires many frequent updates to the synthesis inputs. Hence, it is critical that STS reaches high-quality solutions in only a few iterations, i.e., low-latency optimization. Low-latency drives many of the system architecture decisions, which translate into: submission of multiple parallel scenarios, condensing of parameters to primitives (see Section 2.1), and use of novel parallel and fast online learning algorithms.

STS is typically run for about 3-5 iterations, leading to a little over a 3-5x runtime (latency) increase versus a single synthesis run. Over the 3-5 iterations, approximately 100-200 scenarios are run. Although this overhead may seem costly, within the scope of a large design project it is quite tolerable and provides a high ROI for the following reasons: 1) STS is a fully autonomous system that does not require human designer effort once initiated. 2) It is not necessary to run STS every time a macro is synthesized but only at certain points in the design cycle to locate customized parameters for a specific macro; during subsequent synthesis runs, the STS scenarios can be reused to avoid runtime overhead.

2.1 STS Components & Terminology

To allow a detailed description of the decision engine algorithms in Section 3, we introduce terminology specific to STS.

Primitives: To reduce the ~1000 multi-valued parameter space up front, we recast this DSE problem to have a space of about 100 Boolean parameters. This design space reduction involves a one-time offline effort to create a library of *primitives*. A primitive contains one or more synthesis parameters set to specific values. Table 1 shows an example of a small primitives library, although the actual library in our case consists of ~300 primitives. In general, a primitive targets a singular action. Thus, an STS primitive is a binary decision, whereas setting many parameters individually may require many more decisions. The choice of tuning primitives instead of parameters directly leads to faster tuning while possibly missing some points in the design space; however, we believe this compromise is beneficial given the need for low-latency.

Scenarios: A scenario is a complete parameter setting to launch a synthesis job. STS creates scenarios consisting of one or more primitives. Selecting primitives to construct high-quality scenarios is non-trivial, motivating the need for intelligent decision algorithms like those presented in Section 3.

Cost Function: The STS cost function is a key setting conveying the optimization goals. It converts multiple design metrics into a single cost number, allowing cost ranking of scenarios. Examples of available metrics include: multiple timing metrics, power consumption, congestion metrics, area utilization, electrical violations, runtime, etc. The selected metrics are assigned weights to signify their relative importance. STS provides a number of reference cost functions which combine timing, power, and congestion metrics. Designers can use these cost functions directly or to build custom functions.

The overall cost function is then a "normalize weighted sum" of the m selected metrics, expressed as:

$$Cost = \sum_{i=1}^{m} W_i \cdot Norm(M_i) \qquad \begin{array}{l} where: \\ W_i = weight_i \\ M_i = metric_i \end{array} \qquad (1)$$

where $Norm(Mi)$ is the normalized Mi across all the scenario results in a STS run. A default balanced cost function does not favor any particular metric, i.e., all the selected weights W_i's are equal.

3. DECISION ENGINE ALGORITHMS

The decision engine and tuning algorithms are key STS components that determine the total number of iterations and scenarios to be run during each iteration. The low-latency requirement drives the need to develop custom STS decision algorithms that can reach high-quality solutions in a few iterations. The decision engine is also a modular component that can be upgraded independently, allowing incremental refinement.

Given a set of primitives, a decision algorithm returns a list of lowest-cost scenarios with respect to the designer-defined cost function (Equation (1)) and generates a set of new scenarios for the next iteration. In practice, the optimal scenarios are macro-specific for these reasons: 1) the heuristic nature of the underlying synthesis algorithms requires different settings based on a macro's logic and 2) the objectives in a specific cost function will call for unique parameter settings.

3.1 The Base Decision Algorithm

The initial STS decision algorithm, which we call the *Base* algorithm and specify in Algorithm 1, is a pseudo-genetic algorithm involving a survival of the fittest comparison (*sensitivity test*), followed by a dense search using the top primitives, as illustrated in Figure 2.

The Base algorithm begins with STS launching an initial iteration (i=0) consisting of one scenario for each given primitive, i.e., a 1-hot sensitivity test. Then, it selects the best N (lowest-cost N) scenarios as a "survivor set" and proceeds to iteration 1 (i=1). Iteration 1 generates a stream **S1** of more complex scenarios, consisting of combinations of primitives from the survivor set (e.g. b, j, d, f in Figure 2). The most common configuration is to generate all possible combinations of i+1 primitives for each iteration i. The algorithm works on the premise that all primitives are *complementary*; therefore, combining the survivors would yield optimal or near-optimal scenarios. In the example of Figure 2, the number of primitives is 10 and the size N of the survivor set is set to 4.

Although the Base algorithm fully searches the design space of the survivor set, its practical size is often constrained by the available compute resource requirements, i.e., a large survivor set may lead to too many parallel scenarios in one iteration. To mitigate this problem and expand the survivor set, an **S2** stream of scenarios can also be added to the i=1 iteration. The **S2** scenarios, which are rule-based guesses such as combining the M (M>N) lowest cost primitives (M = 5 and 6 in Figure 2), were a late addition to the Base algorithm to cover the known deficiency and provide a bridge until a more sophisticated algorithm could be implemented.

3.2 The Learning Algorithm

Within the framework of the Base algorithm (i.e., a sensitivity test followed by iterative combination of scenarios), we present an enhanced decision algorithm (see Algorithm 2), which we call the *Learning* algorithm to better address the deficiencies described in the previous section. The Learning algorithm selects a given number k of scenarios in each iteration

Algorithm 1: Base
Input:
P: set of primitives
f(**S**): cost function (Equation (1)) on a set **S** of scenarios
N: size of Survivor Set ($1 \le N \le |\mathbf{P}|$)
M: size of "Stream 2" scenarios ($N < M < |\mathbf{P}|$)
Output:
S*: sorted list of low-cost scenarios
// sensitivity test (i = 0)
1: **S** ← {|**P**| scenarios, each including a unique primitive}
2: run |**S**| synthesis jobs with **S**
3: **S*** ← sort(**S**) with f(**S**)
4: Survivor Set **S'** ← {combine top-N scenarios in **S***}
// tuning iterations (i > 0)
5: **for** (i ← 1; $i < N$-1; i++)
6: | **if** (i = 1) *// prepare "Stream 2" scenarios*
7: | | **S** ← {the top-n scenarios in **S'** | n in {N, M, M+1}}
8: | **else**
9: | | **S** ← Φ
10: | **S** ← **S** U {$C(N,i+1)$ scenarios: all combinations of i+1 survivor primitives}
11: | run |**S**| synthesis jobs with **S**
12: | **S*** ← sort(**S*** U **S**) with f(**S*** U **S**)
13: **return S***

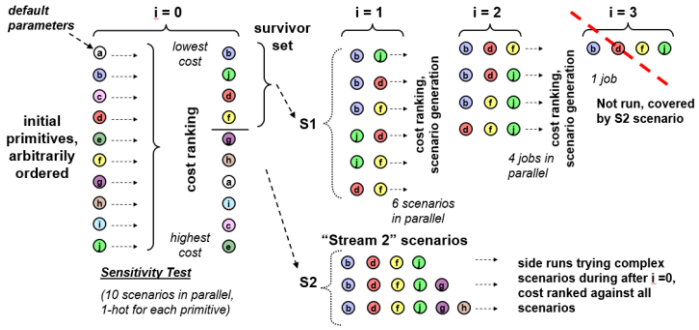

Figure 2. The Base decision algorithm, where each colored bubble represents a primitive, a horizontal sequence of adjacent bubbles represents a scenario, and i denotes the iteration number.

as parallel synthesis jobs (i.e. maximized utilization of compute resource), and dynamically adapts to the k scenarios that are more likely to return lower costs (i.e. adaptive exploration).

Figure 3 illustrates the main idea of the Learning algorithm. Following the sensitivity test (i=0) on the given primitives, the Learning algorithm *estimates* the cost of an unknown composite scenario by taking the average cost of its contributing scenarios as a cost predictor. For instance, to estimate the cost of a scenario that comprises three primitives (b, j, d in Figure 3), the Learning algorithm calculates the average cost of the three contributing scenarios that respectively comprise the three primitives. Furthermore, the Learning algorithm can "look ahead" by a combination order O>1, which allows combining up to O prior scenarios for cost estimation (O=3 in Figure 3). This look-ahead predictor allows *complementary* scenarios such as "restruct_t + wireopt_t + wireopt_c" (see description in Table 1) to be discovered for a cost function favoring timing and routability, earlier in tuning iteration i=1 in the Leaning algorithm, as opposed to iteration i=2 in the Base algorithm (compare Line 10 in Algorithm 1 and Lines 4-5 in Algorithm 2). The Learning algorithm uses the look-ahead predictor to learn the *inter-scenario* interaction (Algorithm 2, Lines 8-9).

After the cost estimation, the Learning algorithm selects the top-k composite scenarios with the lowest estimated costs to form a *potential* set and then submits k parallel synthesis jobs with the selected scenarios.

Algorithm 2: Learning
Input:
P: set of primitives
f(**S**): cost function (Equation (1)) on a set **S** of scenarios
k: size of Potential Set ($1 \le k \le |\mathbf{P}|$)
M: size of "Stream 2" scenarios ($N < M < |\mathbf{P}|$)
I: max # of tuning iterations
O: combination order ($O \ge 2$)
β: parameter controlling cost estimation ($0 < \beta < 1$)
Output:
S*: sorted list of low-cost scenarios
// sensitivity test (i = 0)
1: same as in the Base algorithm (let N equal k)
// tuning iterations (i > 0)
2: **for** (i ← 1; $i \le I$; i++)
3: | initialize set **S** as in Lines 6-9 of the Base algorithm
4: | **for** (j ← 2; $j \le O$; j++) *// look-ahead combination*
5: | | **S** ← **S** U {$C(k,j)$ scenarios: all combinations of j scenarios from **S***}
6: | $\alpha \leftarrow \beta^{(i-1)}$ *// α decreases as # iterations increases*
7: | **foreach** scenario s in **S** // major learning iterations
| | *// to learn the inter-scenario interaction*
8: | | $\mathbf{S_c}$ ← set of the contributing scenarios of s
9: | | Coarse-Cost ← average $f(\mathbf{S_c})$
| | *// to learn the inter-primitive interaction*
10: | | $\mathbf{S_r}$ ← set of the reference scenarios of s
11: | | Fine-Cost ← average $f(\mathbf{S_r})$
| | *// dynamically weigh the two types of costs*
12: | | estimated cost of s ← α×Coarse-Cost + (1-α)×Fine-Cost
13: | Potential Set **S'** ← {top-k scenarios in **S** with the lowest estimated costs}
14: | run |**S'**| synthesis jobs with **S'**
15: | **S*** ← sort (**S*** U **S'**) with f(**S*** U **S'**)
16: **return S***

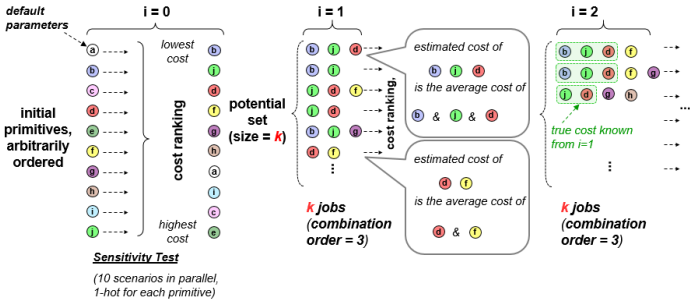

Figure 3. Illustration of the Learning decision algorithm.

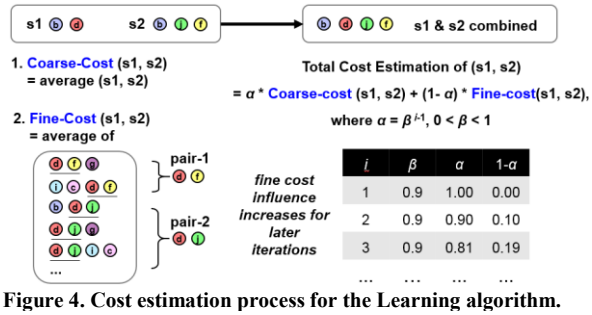

Figure 4. Cost estimation process for the Learning algorithm.

Since the size k of the potential set is constrained, with a combination order greater than 1 the Learning algorithm can filter out non-promising scenarios early in the tuning loop and allocate instead the synthesis budget to the more promising scenarios. This estimation-selection-submission process repeats for every tuning iteration until an exit criterion is

Table 2: Comparison of Learning3+ and Base algorithms across a 12 macros test suite that is representative of IBM 22nm z13 processor.

	Timing			Power	Route-ability
	Worst Slack	Latch to Latch Slack	Total Negative Slack	Total Power	Route Score
Improvement	(%)	(%)	(%)	(%)	(%)
Base	24%	43%	42%	11%	46%
Learning3+	35%	70%	62%	10%	43%
12 macro sum	(ps)	(ps)	(ps)	(a. u.)	(a. u.)
default	-256	-105	-77505	1125	129
Base	-195	-60	-45104	1003	69
Learning3+	-167	-32	-29138	1007	74

met (e.g., max iteration *i* is reached).

Furthermore, the Learning algorithm leverages the iterative process to continuously refine its cost-estimation accuracy on *non-complementary* combinations. Specifically, at any iteration *i*, whenever a composite scenario, say "restruct_t + area_he", was predicted good (i.e. low timing and area costs) and selected for synthesis, but the synthesis result turns out to be not good (mediocre weighted cost because of conflicting underlying optimization mechanisms), then the algorithm can *learn* the actual effectiveness of combining scenarios "restruct_t" and "area_he". Therefore, at any future iterations, it will demote any composite scenario that involves "restruct_t + area_he". In summary, the Learning algorithm uses cost estimation to avoid non-promising scenarios and refines its estimation after learning actual synthesis results.

Moreover, to better estimate the cost based on *non-trivial* contributing scenarios (i.e., scenarios comprising more than one primitive), the Learning algorithm includes a fine-grained cost estimation (see Figure 4). For instance, given two scenarios, **s1** = (b + d) and **s2** = (b + j + f), the algorithm not only regards the average cost of **s1** and **s2** as the *coarse-cost*, but also considers a *fine-cost*. The fine-cost aims to learn the *inter-primitive* interaction (such as d + j), in contrast to the coarse-cost that targets the inter-scenario interaction (such as **s1** + **s2**). To this end, we calculate the fine-cost of **s1** and **s2** using the average cost of their *reference scenarios*, which are the scenarios that have been run in the previous iterations and each include a pair of primitives, such that one primitive (e.g. d) comes from **s1** and the other (e.g. j or f) from **s2.** See pair-1 and pair-2 in Figure 4 for illustration, where the scenarios listed to the left of pair-1 and pair-2 are the reference scenarios for **s1** and **s2**. Hence, the fine-cost of **s1** and **s2** is the average cost of these reference scenarios. Thus, the previous example of non-complementary "restruct_t + area_he" combination can also be considered when estimating the cost of **s1** + **s2**, if primitive d is "restruct_t" and primitive j (or f) is "area_he". The learning of inter-primitive interaction is specified at Lines 10-11 in Algorithm 2.

Overall, the Learning algorithm's cost-estimation function (Line 12 in Algorithm 2) is a dynamic weighted sum of the coarse- and fine-cost with a changing weighting factor α for the coarse-cost and a factor (1-α) for the fine-cost, where $0 \leq \alpha \leq 1$. For determining α we used the formula $\alpha = \beta^{(i-1)}$ where $0 < \beta < 1$ and *i* is the current iteration number. That is, we favor the fine-cost more in the later tuning iterations as there are more references scenarios available.

4. EXPERIMENTAL RESULTS

4.1 Learning vs. Base Algorithm Results

Here we compare the Base and Learning algorithms using a test suite of macros that are representative of the IBM 22nm z13 processor from [3]. The new Learning algorithm provides a number of configuration settings that can be tuned to improve performance. Prior to the comparison with the Base algorithm (whose size of the survivor set is 6 and number of tuning iterations is thereby 5), we fine-tuned the Learning algorithm parameters (we skip these details due to space limitations). The resulting algorithm is called *Learning3+*, whose size of the potential set, max number of tuning iterations, combination order, and weight-changing parameter β are 20, 5, 3, and 0.9, respectively.

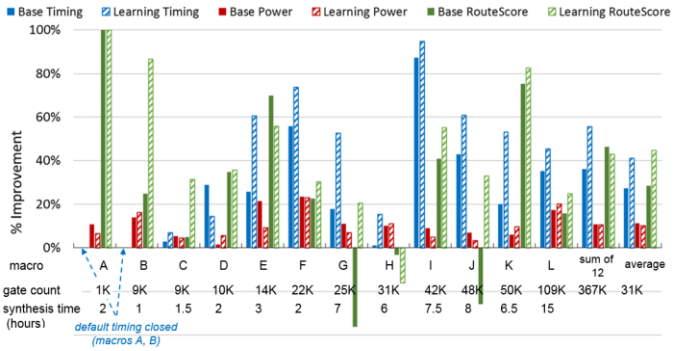

Figure 5. Macro-by-macro breakdown of Learning3+ vs. Base.

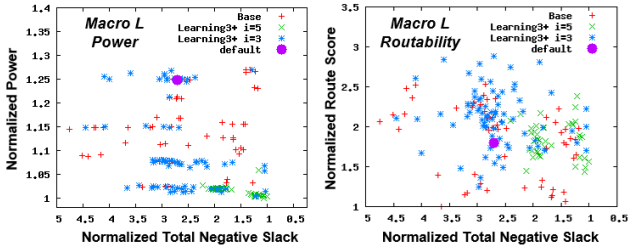

Figure 6. Power and Routability results for the largest test suite macro.

The Base and Learning3+ comparison employ the same default cost function, which is balanced across timing, power and congestion metrics. The test suite consists of 12 macros, ranging in size from 1K to 110K gates, with an average size of 31K gates (the same average macro size as the processor). We use a large number of initial primitives (49) to emulate a realistic design scenario. For both algorithms, the maximum scenario count per iteration is 20 (after the sensitivity test, i.e., i>0) and the total iteration count is 5 plus the sensitivity test.

The exploration results are summarized in Table 2. For this study we did not route the macros and, therefore, we report only post-physical synthesis statistics. In lieu of routing, we include a routability metric called *route-score*, in which a lower value denotes less congestion (i.e., a more routable macro). Overall, the results of Table 2 are in the same average improvement range for the Base algorithm as with the processor design from [2]. However, the Learning3+ algorithm achieves an additional 20% total negative slack improvement over the Base results as well as significant improvements across all timing metrics. Figure 5 provides the macro-by-macro percentages of change for each metric category in the cost function. Only one macro sees degradation (macro H) for the Learning3+ algorithm, while three macros see degradation for Base.

Figure 6 provides a visual representation of the explored design space for the largest macro in the test suite. These plots show total power and route-score vs. total negative slack (all values normalized). The less the power/route-score/negative-slack, the better the quality of result (i.e., the optimization goal is the lower-right corner of the plots). For macro L, we see that Learning3+ provides the best timing and power scenarios, whereas the Base algorithm finds solutions with better route score. The results also show that Learning3+ indeed adapts to the more promising design space, i.e., the *i*=5 dots are more focused on the lower-right region of the spaces than the *i*=3 dots. Overall, if we compare the single best scenario (lowest cost scenario) for macro L based on the balanced cost function (Figure 5), Learning3+ actually outperforms Base in each metric, including route score.

4.2 14nm STS Improvements

In this subsection we describe STS improvements for macros from a 14nm server processor currently being designed. The goal is to not only show the effectiveness of STS at a more advanced technology node, but also highlight the multiple design point options made available by STS.

Table 3. Avg STS improvements for a 14nm processor (150 macros).

Top Scenario Per Metric	Improvement % Per Metric					
Scenario \ Metric	Slack	L2L	TNS	Cong	Power	Area
Overall optimal	19%	49%	40%	25%	2%	4%
Slack optimal	25%	43%	38%	10%	1%	3%
L2L optimal	12%	60%	40%	0%	1%	3%
TNS optimal	19%	50%	48%	17%	2%	4%
Cong optimal	7%	20%	24%	30%	2%	3%
Power optimal	0%	7%	9%	13%	11%	6%
Area optimal	8%	27%	27%	16%	6%	9%

STS ranks scenarios by a single cost value computed from multiple metrics and weights from a user's cost function. But the choice of scenario(s) to continue through the later stages of the design flow, e.g., routing steps, is ultimately in the hands of the designer. Thus, STS not only provides a suggested best scenario by cost, but also a number of other scenarios that may be more attractive with respect to specific metrics while having a higher overall cost. Table 3 shows the average STS improvement percentages for over 150 macros from a 14nm server design in progress. The results were mined from the STS archive that stores data from all STS runs and are thus results from the actual processor design cycle.

The first row of the table shows the average improvement percentages for the top scenario in terms of cost. The second through last rows of Table 3 provide average STS improvements for the top scenarios with respect to a specific metric. Although these scenarios often are not as well rounded as the scenario with the lowest cost, they provide a variety of design point options that may solve the challenges of a specific macro.

5. STS SCHEDULER (STSS)

While the optimization approaches described in previous sections are effective for achieving QoR improvements for a single macro, large design projects often consist of a large number of macros that are concurrently designed by multiple human designers. Furthermore, limited compute resources, even in an industrial setting, require ROI considerations when investing effort into tuning the parameters of macros. For example, during the processor design described in [2] there were many times when the compute cluster was heavily loaded with parameter tuning jobs and other times when the cluster was relatively idle. This unpredictable pull on compute resources inherently arises when designers work independently without global project ROI considerations and/or without tightly-coupled communication of compute needs. These challenges motivate a solution for enhancing the cumulative parameter tuning QoR for an entire design project that consists of multiple macros.

Our novel solution is the *SynTunSys Scheduler*, a.k.a., *STSS*, a system that manages multiple STS runs for multiple macros. This system works at a higher level of abstraction that considers the ROI of STS runs at the project level. Figure 7 shows a diagram of the components and processes of STSS. The general goal of STSS is to take a list of STS-run requests and optimally determine the order in which to submit them to the queue manager, given resource limits. The more general problem STSS addresses is CAD-tool scheduling, which recently is receiving more attention, e.g., for scheduling architectural simulations for a single design [5].

The process begins at Step (1) in Figure 7 where the following inputs are provided to STSS: A) a list of STS run requests and a reference set of synthesis QoR stats for the multiple macros (note that if the reference QoR stats are not available, STSS will first schedule one synthesis run on all macros in the list to generate them), B) a priority-ranking policy and global cost function, which will be described below, and C) compute resource limits. Given these input data, STSS creates a priority ordered list of the STS requests. There are multiple policies that can be employed for the priority ranking, as we describe later. Next, at Step (2), the queue interfacing component of STSS submits one or more STS runs to the queue manager (existing software, e.g., Platform LSF), which ultimately starts the synthesis jobs on the compute cluster. After submission, an STSS monitor process starts to interact with the queue manager to monitor progress of the STS jobs and to sample the compute cluster load. Step (3) in Figure 7 is triggered whenever an STS run completes and involves

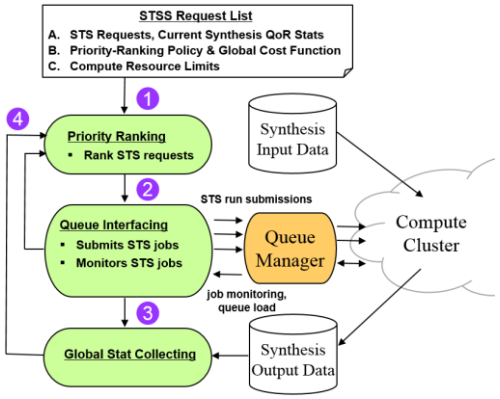

Figure 7. Architecture of the STS Scheduler (STSS).

a result collection process that adds the new STS tuning results to the list of the current best synthesis results for all macros. The feedback process is then initiated as Step (4) where the updated list of synthesis results for all macros, compute cluster load information, and compute resource limits are passed to the priority ranking algorithm. After updating the priority ranking, the queue interface determines if more STS runs should be submitted.

5.1 STSS Priority Ranking

The key component of STSS is the priority ranking that predicts which macros will provide the highest ROI from tuning. There are multiple policies for this prediction. Due to space limitations we describe only two possible ranking policies.

Policy 1: This policy ranks macros based on a cost analysis of existing synthesis run stats for each macro. The cost analysis can use the same "normalized weighted sum" cost function and metrics from Equation (1). However, in this case the goal is to compare a single set of synthesis metrics for multiple macros, rather than compare multiple scenarios from a single macro. This policy effectively works on the worst macros based on the current QoR. One possible shortcoming is that it assumes that tuning macros with worse QoR will provide strong tuning improvements.

Policy 2: This policy first performs a sensitivity test on all macros and then ranks macros based on QoR improvements from the sensitivity test. The sensitivity could in fact be the first iteration of a STS run or a simplified test that is used to only rank macros. The advantage of this policy is that it samples the actual tuning QoR potential before investing in a complete STS run. The downside is that the up-front effort invested in the sensitivity test could be used to directly tune macros.

5.2 2nd Pass STS Runs

Another tradeoff to consider is that multiple STS runs can be applied to a macro to further explore the design space. These "2nd pass" runs build off the results of the first STS run by keeping top performing primitives and removing poor performing primitives from the search space. New unexplored primitives are then added to the 2nd pass STS run. Thus, choosing whether to perform a 2nd pass tuning run on a previously tuned macro or tune a new macro is a decision for the priority ranking algorithm. Although these multi-pass (2nd pass and beyond) STS runs can lead to diminishing returns, the next sub-section will show the advantages.

5.3 STSS Results

To demonstrate the effectiveness of STSS we apply the system to the 12 macros from Section 4 and use the Learning3+ algorithm for all STS runs. For this example we use *Policy 1* and the same balanced cost used for the tuning results from Table 2 and Figure 5. Also for brevity, we assume that tuning runs are executed sequentially, i.e., one at a time. First, we consider tuning the 12 macros without allowing 2nd pass tuning runs. Figure 8a shows results comparing cumulative improvement of three key metrics using the STSS *Policy 1* ordering vs. a random macro ordering. We compare against random macro ordering because during an actual design project, without a centralized higher level system like

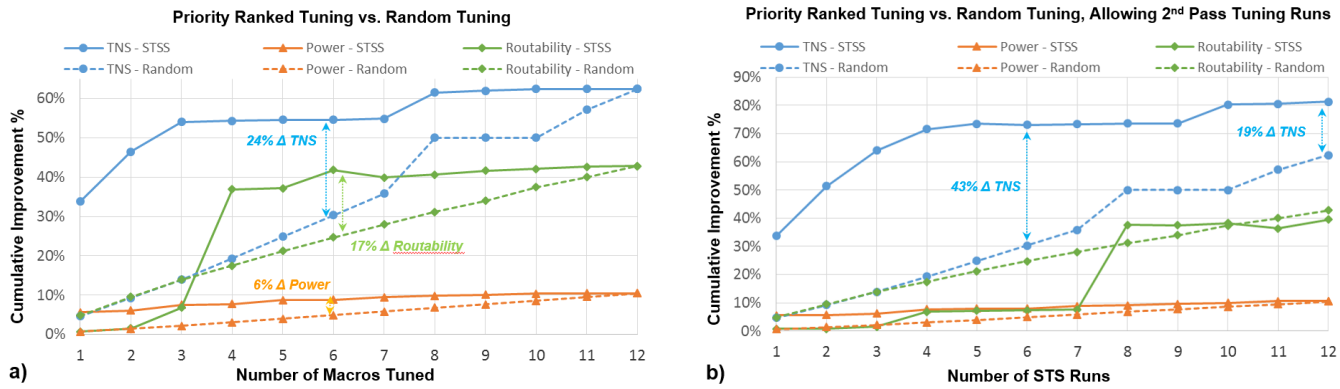

Figure 8. Results of tuning 12 macros based on STSS priority ordering vs. random ordering.

STSS, designers effectively submit STS runs at will without considering macro priority, thus a random ordering reflects a realistic industrial setting. To simulate this, we generate 1000 random orderings of the 12 macros and then average the QoR improvements across them. The results show the distinct advantage of STSS in terms of total negative slack (TNS), power, and routability. We highlight one sample point in the plot where half the macros are tuned. At this point STSS has a 24% TNS advantage and a 6% power advantage over random tuning. Furthermore after tuning only half the macros, STSS has achieved 58% out of the total 62% TNS savings and 9% out of 10% power savings available from tuning all 12 macros. Thus, the STSS priority ordering allows us to evaluate ROI options and determine whether continuing to tune macros is beneficial or has diminishing returns.

Next, we consider the same case-study of the 12 macros but allow the option of 2nd pass STS runs. Figure 8b shows the results of this experiment. We perform 12 total tuning runs, where the random ordering results are the same as in Figure 8a. For the STSS case we perform 12 total tuning runs where with the 2nd pass STS option not all macros are necessarily tuned. In this experiment, the priority ranking algorithm chooses to tune 6 macros twice, one macro once, and does not tune 5 macros. The ability to choose 2nd pass STS runs leads to a 19% TNS improvement compared to random ordering and a 43% TNS advantage after only 6 STS runs.

6. RELATED WORK

The synthesis parameter tuning problem we address can be classified as a black-box optimization problem, i.e., we treat the synthesis program as black-box software by supplying input conditions (input data and parameter settings) and measuring the output response in terms of synthesis quality of results (QoR). Black-box problems are often approached using techniques from the field of simulation optimization [6], which is an umbrella term for optimization techniques that operate in the absence of an algebraic model of the system. Since each macro exhibits a unique input-output response to the synthesis parameter settings and digital logic can take on an intractable number of functionalities, the synthesis tool-flow of our focus is far too complex to be modeled algebraically. Black-box optimization techniques can also be employed for DSE purposes. However, unlike conventional DSE, the goal of black-box optimization is often to find one or more optimal or near-optimal design points without necessarily requiring a complete exploration of the design space to determine the whole Pareto frontier of tradeoff points.

Black-box optimization is a common problem seen across a number of fields, e.g., compiler tuning [7] and software engineering [8,9]. With respect to VLSI design, DSE is becoming a more attractive solution for complex problems across various levels of abstraction. At the architectural level, many DSE studies based on models or simulators have been used to explore multi-objective design spaces, e.g., [10]. Architectural-level studies, however, typically do not result in implemented designs. DSE approaches have been used for high-level synthesis by leveraging machine-learning methods [11] and for FPGA synthesis by tuning parameters with genetic algorithms [12] or Bayesian optimization [13].

STS differs from all these approaches because it operates on a general-purpose synthesis tool-flow targeting VLSI chip design. Also, STS has been proven in an industrial setting, being used for multiple high-performance processors, currently in production in advanced technology nodes.

In comparison to the two recent works on FPGA parameter tuning, STS does not require the human-designer hints that have been used for genetic algorithms in [12] and our in-house learning algorithm has advantages over the Bayes approach in [13]. In particular, our Learning algorithm from Section 3.2 performs *cost ranking* after each iteration, as opposed to the *classification* that standard machine-learning algorithms, such as Bayes or SVM, would perform. The ranking allows predicting the top-k most promising scenarios for the next iteration, as opposed to random sampling followed by classification. Thus, we believe the Learning algorithm may converge faster than the classification approaches and is more appropriate for low-latency optimization. In fact, we often see high-quality scenarios emerge after only two iterations.

7. CONCLUSION

We presented enhancements to the *STS* system across various fronts, including a novel parameter tuning algorithm employing adaptive learning that improves quality of results over our original algorithm. Results using these enhancements are presented based on 22nm and 14nm high-performance industrial server components. Furthermore, we presented STSS, a novel higher-level system that manages parameter tuning of multiple designs. This addresses the next key challenge of automating design tasks performed by a larger design team, leading to improved timing and power.

Acknowledgments. This work is partially supported by the NSF (A#: 1527821) and by C-FAR (C#: 2013-MA-2384), one of the six SRC STARnet centers.

8. REFERENCES

[1] L. Trevillyan, et al., "An Integrated Environment for Technology Closure of Deep-Submicron IC Designs," IEEE Design & Test of Computers, vol. 21:1, 2004.

[2] M. M. Ziegler, et al., "A Synthesis-Parameter Tuning System for Autonomous Design-Space Exploration," DATE 2016.

[3] J. D. Warnock, et al., "22nm Next-Generation IBM System z Microprocessor," ISSCC 2015.

[4] S. Zhou, et al., "Utopia: a Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems," John Wiley & Sons, 1993.

[5] G. P. Mariani, et al., "DeSpErate++: An Enhanced Design Space Exploration Framework using Predictive Simulation Scheduling", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. Vol. 34, No. 2, Feb. 2015.

[6] S. Amaran, et al., "Simulation Optimization: A Review of Algorithms and Applications," 4OR - A Quarterly Journal of Operations Research, Dec. 2014.

[7] G. Fursin, et al., "Milepost GCC: Machine Learning Enabled Self-tuning Compiler," International Journal Parallel Programming, 39:296-327, 2011.

[8] A. Arcuri, G. Fraser, "Parameter Tuning or Default Values? An Empirical Investigation in Search-Based Software Engineering," Empirical Software Engineering, June 2013, Volume 18, Issue 3.

[9] H. H. Hoos, "Programming by Optimization," Comm. of the ACM 55(2), Feb. 2012.

[10] O. Azizi, et al., "An Integrated Framework for Joint Design Space Exploration of Microarchitecture and Circuits," In Proc. of DATE, 2010.

[11] H.-Y. Liu and L. P. Carloni, "On Learning-Based Methods for Design-Space Exploration with High-Level Synthesis," DAC 2013.

[12] M. K. Papamichael, P. Milder, J. C. Hoe, "Nautilus: Fast Automated IP Design Space Search Using Guided Genetic Algorithms," In Proc. of DAC, 2015.

[13] N. Kapre, et al., "Driving Timing Convergence of FPGA Designs through Machine Learning and Cloud Computing," FCCM 2015.