ANUBIS: A New Benchmark for Incremental Synthesis

Rafael T. Possignolo
rafaeltp@soe.ucsc.edu

Nursultan Kabylkas
nkabylka@ucsc.edu

Jose Renau
renau@ucsc.edu

University of California, Santa Cruz
1156 High Street
Santa Cruz, CA, USA
http://masc.cse.ucsc.edu

ABSTRACT

Synthesis, placement, and routing turnaround times are some of the major bottlenecks in digital design productivity. Engineers usually wait several hours to get accurate results to design changes that are often quite small. Incremental synthesis has emerged as an attempt to reduce these long times, but research in incremental synthesis currently lacks a consistent benchmark to enable comparison between different flows and that is reflective of real design changes. In this paper, we propose ANUBIS, a benchmark for incremental synthesis based on real designs and real design changes. ANUBIS comes with a standard score that allows for easily comparing different flows. We evaluate ANUBIS using two incremental commercial flows to give insights on its usage and reporting.

1. INTRODUCTION

Synthesis[1] is a tedious and time consuming process that is repeated multiple times during the design phase of a project [18]. Industry players have recognized this problem and have been trying to reduce synthesis time by taking different approaches [3, 22]. Nevertheless, the current standards are either limited in results or require manual interactions, often increasing designer effort and degrading Quality of Results (QoR).

Incremental synthesis techniques have been shown to improve synthesis time by re-utilizing parts of the resulting circuit. These techniques have been applied in industry [3, 24] and in academia [7, 18], but the lack of standard benchmarks makes it hard to compare different approaches and to understand how the results presented in a paper can be translated into "real-life" expectations.

Moreover, different approaches target different steps of the synthesis process, making it harder to directly compare them, even if the benchmarks are the same. For instance, the current version of Vivado includes incremental placement and routing [24] but does not perform incremental logic synthesis, while existing academic papers focus only on logic synthesis [7, 18].

There are multiple sets of standard benchmarks used by the design automation community, *e.g.*, the ISCAS benchmarks [5, 6] or the ITC benchmarks [9] (among others). However, these benchmarks do not target incremental flows. This prevents them to be applied directly in incremental synthesis due to the lack of standard changes to those circuits. Incremental synthesis benchmarks should be representative of real world designs but they should also include representative changes over which incremental synthesis is evaluated. Moreover, when comparing multiple flows, the same set of changes needs to be used to allow for a fair comparison.

Previous papers dealing with incremental synthesis have used ad-hoc benchmarks, that included a variable number of designs, some of which that may have "real-life," but with rather arbitrary changes. For instance, Chen and Singh [7] used 40 industrial benchmarks with hand made "small" changes; while it is more likely that the designs come from actual industrial applications, it is unclear whether the changes are reflective of real changes. *LiveSynth* [18] used three publicly available designs and based their changes on commented out code and repository history. This approach yields more reasonable changes, but the designs used (except for the MIPSfpga [12]) are not good representatives of industrial designs.

In this paper, we aim to provide the first incremental synthesis benchmark suite, ANUBIS, which includes a collection of open-source designs as well as standard changes. The use of industrial benchmarks would certainly improve the representativeness of the benchmark set. However, we want to stick with a set of designs that is publicly available and can be freely used for academic research without licensing. On top of the designs used, we introduced changes based on repository history and commented out code, when available, with the addition of synthetic changes that aim to exercise the case where RTL file changes do not result in any logic change in the circuit (comments, variable renaming, so forth), as those could be common and should not result in any effort by an "ideal" flow.

We also propose a standard way of scoring and reporting results using ANUBIS. The goal is to have an

[1]"Synthesis" is used here in a broad sense. Throughout this paper we distinguish between synthesis and logic synthesis. Logic synthesis includes RTL elaboration, logic optimization and technology mapping. Synthesis is logic synthesis plus placement plus routing.

equivalent to the popular SPECint benchmark typically used to report performance numbers in CPUs. The unified ANUBIS score provides an easy way to compare different proposals for incremental synthesis, while the standard reporting requirements provides insights on where flows are doing a good or a poor job. The ANUBIS score takes into account incremental synthesis time but also includes QoR results.

To evaluate ANUBIS, we run two commercial incremental synthesis flows over ANUBIS. The results are reported in the proposed standard table, with the final scores and provide some insights on the results found. Our evaluation shows that both the flows considered do a very good job in delivering the same QoR when in incremental mode, with a maximum of 4% of area degradation observed, and no more than 1% increase in delay observed for both flows. However, we also observe that the incremental synthesis time is usually not proportional to the amount of changes. For instance, in cases where no actual change was made, runtime was usually on the order of half of the full synthesis runtime.

The main contributions of this paper are:

- Propose ANUBIS, the first incremental synthesis benchmark set
- Propose a standard score and report table to facilitate the comparison of flows using ANUBIS
- Evaluate ANUBIS using existing incremental synthesis flows

2. RELATED WORK

The related work is split into two main parts: incremental synthesis techniques and other benchmarks. In the first part, we discuss the type of work that could benefit from ANUBIS and the benchmarks used in their evaluation. In the second part, we discuss how other benchmarks (not necessarily for synthesis) work, how they were created and how they are evaluated.

Incremental Synthesis: The first incremental synthesis flow was proposed 30 years ago [13] in order to improve timing closure in digital design. The flow was interactive and kept the design in memory while changes were being made by the designer. The flow needed under 30 minutes to evaluate large (at the time) designs, but could compute the effects in frequency of a small design change in only a few seconds. The main motivation of the flow was timing analysis, with an incremental timer and the designer would manually indicate design changes over the netlist to improve timing.

Incremental synthesis was revisited more recently by other authors. Dehkordi *et al.* [10] propose a flow that partitions the design into independent synthesis regions. After a change is introduced only the affected partition is re-synthesized. Due to the artificial partitioning method, there is a significant hit on QoR depending on the parameters choice. Authors used a set of 22 "industrial benchmarks" with manually added changes. There is no information about the changes added, but it is clear that they were not based on real code changes.

To reduce the impact on QoR, newer approaches include detecting regions impacted by the changes, regardless of an original partitioning of the design. A flow coupled with Altera synthesis flows leverages information of nets whose functionality is not modified during synthesis. When a change is made to the RTL, the flow maps that change to a specific region defined by those "invariant" nets, replaces the synthesized of the affected region by the elaborated netlist of the new code and launch synthesis over the design. Since most of the design is already synthesized and optimized, there is little work that needs to be done, reducing synthesis time [7]. A different approach is to only synthesize the modified logic, which further reduces the synthesis time and has been shown to maintain QoR [18].

Incremental time analyses have recently been pointed out to be a weakness in timing-driven flows [11]. In modern digital design flows, timing analysis is essential to identify critical paths and avoid optimizing non-critical paths [14]. During performance-driven optimization, timing analysis tools are used several times to assess the impact of optimizations in the circuit [14]. Since most of these changes are localized, running full timing analysis is a waste of resources. The recognition of this problem led to an academic competition in 2015[2] for incremental timers. Although incremental timers are more geared towards the optimization process of a static netlist, for instance, during placement, it is also true that they could be used for incremental changes to the RTL, specially during the timing closure loop.

Other Benchmarks: In the Incremental Timing and CPPR contest the evaluation of designs was done by using standard circuit benchmarks with changes generated randomly by a computer program in the netlist level and not from real ECO changes[3]. The changes were described in the form of actions, such as "add/remove connection", "add/remove cell", and so forth. Despite being useful to evaluate and test incremental timing, those changes do not reflect real-world like changes that would be done to a design.

The ISCAS benchmarks [5, 6] are very popular in the synthesis community and have been used by countless research papers to evaluate and compare different proposals. The ISCAS benchmarks consist of a set of netlists from real industrial designs. Another set of benchmarks, the IWLS benchmark [2]–maintained by this community–include RTL description of over 80 industrial designs, with the respective mapped netlists. The IWLS benchmarks serve a more specific purpose for use in logic synthesis, but can also be used to evaluate parsers and elaboration tools. However, both these benchmark sets are static, in the sense that they do not include changes that were made to those designs during their project and therefore are not suitable for incremental synthesis evaluation.

Evaluating a benchmark is not an easy task. The most common problem associated with creating and using a benchmark is to assess how representative it is of the expected space of applications intended. For instance, excessive benchmark redundancy was shown to be an issue due to the added runtime to evaluate the

[2]TAU 2015 Contest: Incremental Timing and Incremental CPPR: `https://sites.google.com/site/taucontest2015/`.

[3]Personal communication with contest organizers.

suite, on the other hand reducing too much the number of entries in a benchmark can yield reduced coverage [17]. Another issue is to define which metrics are the most suited to evaluate a benchmark. For instance, the placement community has been discussing between different metrics (wirelength, routability, so forth) and the decision on which metric to use largely impacts which placer will be considered the best [1]. A good benchmark should be able to tell which flow is the best, however in some cases, conflicting metrics make it hard to intuitively determine which flow is in fact the best.

In the relatively new and largely unexplored field of incremental synthesis methods, there is still need to define what it means to be the best. In this paper, we discuss some of the metrics that we believe will be important and create a scoring system that is used to evaluate existing commercial flows.

3. ANUBIS

ANUBIS, A New Benchmark for Incremental Synthesis, is a benchmark suite that considers incremental changes in digital designs. The main premise of ANUBIS is that most of the time, during the design cycle of digital circuits, small and localized changes are introduced to the code. Still, current benchmarks for synthesis (logical and physical) considers mostly static benchmarks. ANUBIS tries to capture real changes that were introduced into real designs. With ANUBIS researchers working on incremental synthesis, placement, routing, timing, bug finding techniques or others are provided with a standard tool to compare their work more fairly.

ANUBIS consists of a collection of Verilog designs. The benchmarks were chosen based on open-source status, availability of design changes (as explained later) and size/diversity. We tried to maximize the number and type of designs and changes to be representative of a large set of real-life cases. In the next subsections, we describe how the benchmarks were selected, how changes were inserted into the benchmarks, and how to run ANUBIS to compare multiple incremental synthesis flows.

3.1 Benchmark Selection

The main objective of the benchmark selection criteria is to allow for a good number of designs that are as reflective of real world designs as possible and have enough real code changes in them. We note that these criteria are not very well defined or strict.

Closed source designs or code with limited distribution were not used since it would be impossible to distribute them without a license. We also have a particular interest in looking for code changes. This can take two forms: commented out code or repository commits. Given those two requirements, the main source of benchmarks considered are open-source repositories online, such as GitHub[4] and OpenCores[5].

Another important source considered was from academic designs that were made available with changes, for that we contacted some research groups in multiple universities. In particular, we added the designs from Bug Underground[6], a project at University of Michigan that aims to find bugs in RTL code of cores. They provide two processors with a large number of bugs that are inspired into bugs found in commercial CPUs and reported through erratas. Although those are not actual changes that were made to designs, they closely reflect issues found in real commercial CPUs.

Generated code, such as from High-Level Synthesis (HLS), Bluespec, Chisel, and so forth, were not considered. In theory, generated RTL could be used, but this adds an extra layer to the benchmarks and is currently out of the scope of our benchmark suite. In the future, this decision may be reconsidered.

After gathering open-source design candidates, we analyzed how many changes could be found for them. We went over the code to look for commented out code, and went over the commits in the repository. Open-source designs with no design changes, as it is the case of most of the designs in OpenCores, were excluded.

Other variables considered are the ability to fit the design in a large high-end FPGA, to allow for flows to place and route designs for FPGA and that the design should not require specific vendors. For instance, some designs use IPs specific to Altera or Xilinx, which prevents them from being ported to other back-ends. Unfortunately, this leaves a very limited number of useful designs, but more designs will be added to ANUBIS as they are made available. The list of ANUBIS designs is provided in Table 1. In the remaining of the paper, the benchmarks will be referred to using the acronym presented in Table 1. ANUBIS will be provided under the BDS License 2.0, and results of flows using ANUBIS will published on the official ANUBIS repository `http://github.com/masc-ucsc/anubis`. Anyone using ANUBIS can submit results, and a list of top contenders will be available in the official repository.

3.2 Change insertion

In order to emulate design changes, we inserted code changes to the benchmarks. The changes can be activated or deactivated through *define* statements. To approximate real-world changes, changes were taken from repository commits and commented out code. Some synthetic changes were added to exercise some cases that are interesting but did not appear in either of the above, such as replacing a signal by a constant. A change can be single-line, multi-line, or multi-file. Changes include changing conditions in *if* statements, changing logic to generate data, including/removing ports on a module, and others.

The main source of code differences used was commits in public repositories. We specifically looked for commits in nearby dates, since we specifically target small changes in code. Commits that added entire modules or sub-systems were not considered. The idea of using commits from repositories is to try to mimic "real-word" work. Commits of large amounts of code usually reflect the changes over several days or weeks which is not aligned with the incremental synthesis philosophy that we are interested. Commented out code was used when available, following a methodology similar to the one proposed in [8]. In addition, we do not use changes

[4] `http://github.org`.

[5] `http://www.opencores.org`.

[6] `http://bugs.eecs.umich.edu`.

Table 1: ANUBIS consists of a collection of open-source benchmarks and standard changes applied to it. Lines of Code (LoC), area and maximum frequency (Fmax) (for an ASIC 32nm library using a commercial flow) are included as estimates of the design complexity.

Description	Acronym	LoC	FMax (MHz)	Cells
DLX core [4]	DX	743	770	7152
ALPHA core [4]	AL	1086	666	17558
IEEE 754 FPU[7]	FP	4716	2500	58149
mor1k RISC core [15]	MO	15012	2500	62752
OR1200 RISC core [16]	OR	19437	1300	329280

that would cause syntax errors, but do not make any assumptions on functional correctness.

We also add some cases where no behavioral change is inserted. For instance, we add/remove comments, white space, change variable names, so forth. The rationale behind this is to understand how good the system is at detecting those corner cases where no re-synthesis is needed. Ideally, a good implementation should be able to detect that there was no change and return in almost no time. In some cases, this will not happen and at least a part of the flow will be triggered. Those changes were artificially created by us, but were inspired in cases observed in repository diffs.

Changes are divided into three categories: *NoChanges*, *LocalChanges*, *GlobalChanges*. *NoChanges* does not reflect any real change in the behavior of a system, they can be adding whitespace, double invertions, changing the name of a variable, or actual changes to unused parts of the circuit. The *LocalChanges* category includes changes within a module, mostly single line changes, or very localized changes, for instance changing the condition on an *if-else if* chain, changing the constant values, arithmetic operations, so forth. Finally, *GlobalChanges* are changes that either affect multiple modules or a module that is instantiated multiple times in the design. Although we split changes into *LocalChanges* and *GlobalChanges* it is not necessarily the case that the amount of reused cells will be lower for *GlobalChanges*, since this is largely flow-dependent. We also believe that researches could focus more on some types of changes in a given work, since different types of techniques will behave differently in each category.

A summary with the number of changes added to each benchmark is given in Table 2, with breakdown by category and source (actual or synthetic). For DX and AL, all changes are considered actual changes. We tried to keep a distribution of largely *LocalChanges*, since the idea behind ANUBIS is to leverage small incremental steps, although some *GlobalChanges* are expected. The percentage of *LocalChanges* versus *GlobalChanges* approximately reflects our observations from the repository histories that we looked at, although we did not perform any statistical analysis over the histories.

3.3 Setup requirements to report ANUBIS results

For the sake of fairness, we assume that researchers using ANUBIS will abide to ethics when reporting results. Nevertheless, we discuss some of the "minimum" expected setup conditions for a fair reporting on ANUBIS.

Table 2: Summary of changes inserted in the benchmarks with breakdown by category and source: actual code changes including git and commented out code (A) and synthetic (S).

Design	Total	*NoChanges*		*LocalChanges*		*GlobalChanges*	
		A	S	A	S	A	S
DX	27	0	4	23	0	0	0
AL	15	1	5	9	0	0	0
FP	37	0	7	12	14	2	2
MO	34	0	7	20	0	4	3
OR	31	1	5	19	0	4	2

Equality in number of cores and resources used: When running full synthesis, setup and incremental synthesis the same number of cores and physical resources (memory, IO and network bandwidth, so forth) should be available. The workload on the computers running the flow should also be consistent, and if at all possible only the benchmarks should be running. The server configuration should be disclosed as much as possible, but at least the number of cores used and the available memory should be reported. Note that, if, for instance, the setup flow is single threaded, the incremental synthesis could be parallelized, but the results should be reported with a single thread. This measure prevents a flow of scoring artificially high due to higher parallelism, and although we embrace parallel applications, they are not the main target of ANUBIS.

High effort flow: Flow options should be chosen to achieve the highest quality circuit. In general, that means, maximum (or within 5%) achievable frequency. The 5% is to allow for approximation to integer numbers and to avoid pushing the flow to extremes, which could incur in large optimization overheads. However, flow options like "retiming" can be used at the researcher discretion, but those should be consistent between full and incremental flows and should be clearly disclosed.

Multiple runs: To reduce the effects of runtime variability, we require ANUBIS to be run at least 3 times and the average to be used. If too much variability is observed (*i.e.*, the runtime between different runs differs by more than 10%), 5 runs are recommended.

4. HOW TO SCORE ANUBIS

An important part of a benchmark is to have a fair way of comparing different approaches that use the benchmark. The main question to be answered is what does it mean to be the best approach? That can be broken down into finding the metrics of interest for the problem at hand and giving a relative weight to them.

For incremental synthesis, knowing the percentage of

reuse (changed cells, moved cells, wires that needed rerouting) is an initial potential metric of interest, but as it becomes clear in our evaluation, those do not necessarily translate into saved runtime. At the end of the day, designers doing incremental synthesis are interested in reducing runtime and keeping quality of results. Runtime savings and QoR are easily accessible from running the full and incremental flows, although it is hard to place an absolute importance between them. For instance, it is intuitive to think that a designer would not use an incremental flow that saves 90% of runtime but at the cost of doubling the delay. On the other hand, an incremental flow that offers 10% speedup with minimal QoR impact is possibly not appealing either. However, it is not as simple to decide if a designer would use a flow that reduces runtime by half with, say, 5% impact on delay. That may be acceptable in some cases, for instance if a final optimization synthesis can be performed later to catch up on the QoR gap. In those corner cases, it may be harder to decide which of two flows is better.

Given that, we believe that the ultimate metric for incremental synthesis should be related to runtime speedup, that is, how much time the incremental flow under evaluation can save compared to the full flow. However, the flow should be penalized if it degrades QoR by a "too much", of course how much penalty for how much degradation is an important point of discussion.

Runtime comparisons should be as independent of the hardware in which a flow is running as possible, therefore we propose normalizing the runtime by the runtime of running a standard flow, which serves as a baseline for assessing the hardware power. We also note that any incremental flow will require a setup phase, that consists at least of an initial synthesis, but possibly additional processing steps, such as the approach proposed in [7], change the regular synthesis flow to keep track of information needed later and include extra steps beyond synthesis to prepare for the incremental steps. The runtime of this setup phase is also considered, but with is weighted less, since it should not need to be run often.

Our scoring system is such that higher scores indicate better flows. The score system works as follows: first a sub-score is calculated for each change in each benchmark and the baseline benchmark (*i.e.*, no change case). Then a benchmark score is calculated based on the subscores. Finally, a global ANUBIS value is calculated using the benchmark scores. The ANUBIS value takes into account the time to perform synthesis, placement and routing. To provide better insights on the speedup, our standard reporting also includes breakdown for each phase, as it will be discussed later in this manuscript.

4.1 QoR penalty

One important point to consider is QoR degradation. When performing incremental synthesis, it is possible that there will be degradation in QoR. This is not a deal-breaker in the sense that non-incremental synthesis may be used to close the QoR gap. Prior works on incremental synthesis recommend running non-incremental synthesis while no changes are being performed on the design [18]. Therefore, although it is important that a flow can achieve accurate QoR, it is possible to tolerate small losses. However, if the losses are significant, it may be impractical to use the flow. Our scoring system takes that into account.

The answer to the question of how much QoR degradation can be tolerated may vary significantly from case to case and from personal taste. Still, we want to summarize what may be acceptable in general to most designers and in most use-cases. Therefore, we look at this question from multiple angles. First, we note that commercial FPGAs are divided into speedgrade due to process variation. For instance, in Xilinx FPGAs the difference in speed between grades is of about 14%-13% [23], which indicates that 10% is too large of a variation to be tolerated. Another insight is taken from industrial blogs: for instance, setting different clock constraints around the maximum achievable frequency can led to Fmax differences of around 14% [19]. Another industrial post suggests that there is a variation of around $4-5\%$ in performance due to sign-off [20], which may seem to suggest that 5% could be a tolerable error for most designers. As a final argument, we performed 50 full synthesis using Quartus (more details in Section 5) over the same unmodified design to current variation in synthesis flows due to randomness present in the flow[8]. The results show a standard deviation of 3%, the overall range was of $\pm 7\%$ of the average. This result also seems to confirm that variability should be around 5% to be tolerable by designers. In conclusion, we consider that $\approx 5\%$ variation is acceptable, but $\approx 10\%$ is too much. However, since there is not a definitive answer to this question, we decide not to use a step function, but rather have a sharp but continuous increase in penalty as QoR degrades.

The other piece missing to this discussion is how much penalty should be attributed to flows that "break" QoR. The reasoning behind this is much simpler. If an incremental flow is degrading QoR, it is natural to run the full flow to recover the penalty. Therefore, our scoring function should be such that if QoR is within 5% of the full synthesis QoR, the score is inversely proportional to the incremental synthesis time, and if the QoR is degraded to unreasonable levels, the score is inversely proportional to the incremental synthesis plus the full synthesis time, remember that higher scores mean better flows. We call this the corrected runtime for change n of benchmark a: $\tau(a_n)$, where a is one of the ANUBIS benchmarks and $1 \leq n \leq n_a$ is the change id and n_a is the number of changes for benchmark a. Given that, the corrected runtime is given by:

$$\tau(a_n) = t_i(a_n) + (1+\alpha) \times \frac{t_f(a_n)}{\alpha + e^{\beta \times Q_f(a_n)/Q_i(a_n)}}, \quad (1)$$

where α and β are constants, $t_i(a_n)$ is the time it takes to run the incremental flow on change i of the benchmark a, $t_f(a_n)$ is the time it takes to run the non-incremental flow on that change/benchmark, $Q_i(a_n)$ and $Q_i(a_n)$ are the QoR metric of interest (critical path delay, area or power) for the incremental and full syn-

[8]We note that, modern synthesis flow have been moving away from randomness for the sake of repeatability, nevertheless, the results here are a rough estimation of what could be acceptable in terms of QoR fluctuations.

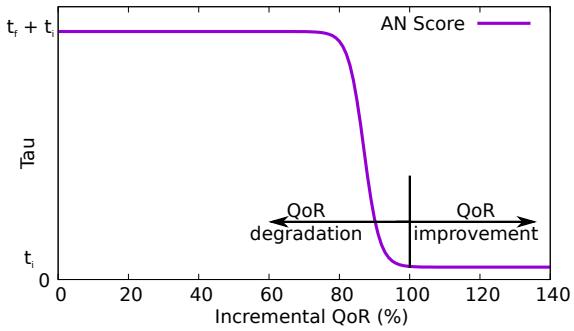

Figure 1: ANUBIS penalizes QoR loses. The penalty is dependent on how much QoR was lost and on the full synthesis time. The rationale is that, if there is too much QoR degradation, the full synthesis will be run to recover it.

thesis flows respectively. We choose $\alpha = 10^8$ and $\beta = 26$ empirically, to match the insights described on the previous paragraphs, *i.e.*, very low penalty for flows that degrade QoR by around 5% or less, and considerable penalty for flows that degrade QoR by more than 10%. We note that there is no benefit for improving QoR. Figure 1 illustrates how this works, the x-axis shows the percentage QoR change of the incremental flow compared to the full synthesis flow (100% is the same QoR, and lower than 100% indicates degradation). The penalty rises sharply after around 5% degradation in QoR up to the time it takes to perform full synthesis. Note that this plots denote the corrected runtime (τ) of the flow for change i of benchmark a, and therefore higher is worse. This number will still be inverted before calculating the final score.

4.2 Score

To make the score machine independent, *i.e.*, to take into account that more powerful machines would artificially improve the runtime, the score for each change in each benchmark is normalized by the runtime of YOSYS (version 0.7+154) with a provided synthesis script for that change in the same machine. YOSYS [21] is an opensource synthesis tool that fully supports Verilog. The correct YOSYS version, the library for techmap and the standard synthesis scripts are provided with ANUBIS, and will be run automatically. Changes to any of those are not allowed. The ANUBIS score $an(a_n)$ for each change is provided by:

$$an(a_n) = \frac{t_Y(a_n)}{\tau(a_n)} \tag{2}$$

where $t_Y(a_n)$ is the YOSYS runtime for change n of benchmark a. This score is calculated for: synthesis, placement and routing independently, but we note that since YOSYS only performs synthesis the baseline is the same for the three. This is only to normalize for computation power. One extra score $an(a_0)$ is added to each benchmark and is calculated considering the setup phase of the algorithm. The idea is that longer setup times will result in a lower overall score.

4.3 ANUBIS Value

For each phase (synthesis, placement and routing) and for each QoR metric (delay, energy and area), the score *an* is calculated as the geometric mean (*gmean*) of all the scores. This yields 9 values that are reported as a table (QoRs vs phase), and 6 sub-scores are calculated as the *gmean* of rows and lines. The final of the flow ANUBIS score is calculated as the *gmean* of the those. A sample standard report table is provided in Table 3. A set of scripts to calculate the scores and generate the table is also provided with the benchmark code. In each cell in the scoring table, a higher number indicates a better flow. Researches focusing on a specific phase can report a subset of the ANUBIS table and/or assume a coupling with incremental approaches for other phases.

The table also include a column with the scores for full synthesis flow. In this case there is no QoR penalty, and thus there is only one column. The average speedus can be obtained by dividing the *gmean* column by the Full column for each task. This already takes into account any penalty in the incremental flow. We also suggest reporting the average runtime for YOSYS, which allows to get absolute average runtime numbers for each task.

5. EVALUATION SETUP

To evaluate our benchmark set, we rely on two commercial incremental synthesis flows (*Flow 1* and *Flow 2*). There is little information publicly available about how these flows are implemented, but the main focus of both flows is to reduce the impact on QoR while leveraging as much as possible from the original design. For this evaluation, both the flows are able to do incremental synthesis automatically, that is, without user intervention such as user-defined partitioning and placement constraints, that are common in FPGA flows.

We run ANUBIS for the two flows independently, in a 32 core Intel Xeon E5-2689 with 64Gb of memory, running ArchLinux-4.9.11-1. Timing measurement was done using the tool provided time to avoid loading overheads. Delay, area and power were also used as provided by the tool, post-routing.

6. EVALUATION

In the first part of our evaluation, we show the standard ANUBIS table for both the flows and discuss the results obtained. We also look into the behavior of each flow in the *NoChanges* category.

6.1 Overall Results

Incremental *Flow 1* consists basically of regular elaboration and synthesis and incremental placement and routing. Whenever a file is changed and saved, the regular frontend flow is run over the design, and the incremental backend flow is ran over the changes. The results for *Flow 1* are shown in Table 4. The best absolute scores are for placement, even though routing is also supposedly incremental the tool still takes considerable time in routing which explains the low scores. The results for synthesis are the worse among the three phases, since it is not incremental. One good way to get insights about the results is to look at the last column

Table 3: Sample report table for ANUBIS.

Phase	Delay	Energy	Area	*gmean*	Full
Synth	$an_{s,d}$	$an_{s,e}$	$an_{s,a}$	*gmean* (an_s)	$full_s$
Place	$an_{p,d}$	$an_{p,e}$	$an_{p,a}$	*gmean* (an_p)	$full_p$
Route	$an_{r,d}$	$an_{r,e}$	$an_{r,a}$	*gmean* (an_r)	$full_r$
gmean	*gmean* (an_d)	*gmean* (an_e)	*gmean* (an_a)	*gmean* (*gmean*)	*gmean* (full)

Table 4: ANUBIS table for incremental *Flow 1*.

Phase	Delay	Energy	Area	*gmean*	Full
Synth	0.105	0.098	0.098	0.100	0.105
Place	2.982	2.704	2.704	2.794	0.175
Route	0.148	0.136	0.136	0.140	0.042
gmean	0.359	0.330	0.330	0.359	0.092

Table 5: ANUBIS table for incremental *Flow 2*.

Phase	Delay	Energy	Area	*gmean*	Full
Synth	0.129	0.129	0.129	0.129	0.075
Place	0.039	0.039	0.039	0.039	0.039
Route	0.070	0.070	0.070	0.070	0.070
gmean	0.065	0.065	0.065	0.065	0.059

of the ANUBIS table that reports the scores for the full synthesis flow. In an ideal case, with no QoR degradation, the speedup of the incremental flow with regards to the full synthesis flow can be obtained by dividing the value in each the *gmean* column by the value in the full column. In this case, we see no speedup for synthesis, ≈ 15 times speedup in placement and ≈ 4 times speedup for routing, on average.

Flow 2 also include a frontend incremental flow, that feeds the incremental backend flow, thus we would expect better scores in the first row of the ANUBIS for *Flow 2*, as compared to *Flow 1* (higher scores is better). The results are shown in Table 5, and confirm this expectation. However, the results for placement and routing are worse when comparing to *Flow 1*. Overall, the much better placement times for *Flow 1* make it have a higher, and thus better, ANUBIS number.

We also observe that all the columns of Table 5 are basically the same. In fact, there was actually some difference after the 5th decimal. This is because *Flow 2* was very good at preserving the QoR, with less than 1% differences between full and incremental flows. In *Flow 1*, it is possible to observe differences mainly in the delay column, which has higher numbers. The variation in delay for the *Flow 1* flow was of up to 1%, but area and power had differences of up to to 4%, which affects the score a bit. Since the median was of $\approx 1\%$, the penalty is still pretty low. From the *Flow 2* table, it seems the only task performed incrementally is synthesis, and there is no change for placement and routing.

One interesting note is that YOSYS took on average 8.83 seconds to complete synthesis in the machine used. Thus it is possible to get average runtime, considering the QoR penalty, for each task, multiplying the *gmean* column by the YOSYS runtime. Although YOSYS is pretty fast for current standards, we believe that incremental synthesis should be able to beat YOSYS, one evidence of that is the current score for placement in *Flow 1*.

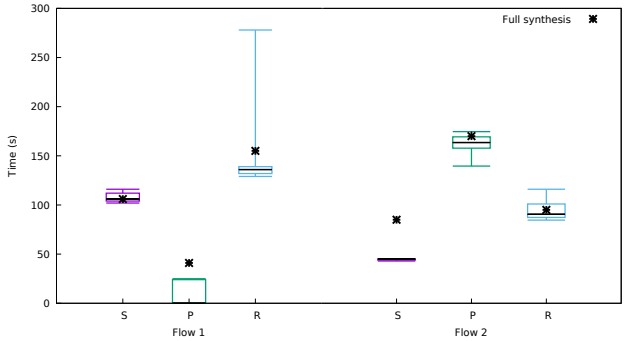

Figure 2: The flows tested cannot detect that no actual change was inserted and run at least partially the incremental flows. *Flow 1* does a very good job in placement presenting a median runtime of zero for placement, but it does a poor job in synthesis and routing. *Flow 2* presents a 2× speedup in synthesis, but placement and routing take a long time.

6.2 No change cases

A good way to gain insights on the flows is to look at *NoChanges*. This is an interesting category to check a flow behavior with no actual change inserted. In these cases, we only added whitespace, changed variable names, added comments, or inserted localized netlist changes that do not alter the circuit functionality.

Due to space restrictions, we are reporting only results for the *NoChanges* category of changes for the FPU design, as other benchmarks followed a similar trend. Figure 2 shows the runtime achieved by both flows for synthesis (s), placement (p) and routing (r) when no actual changes are applied to the design. In *Flow 1*, there is no change in synthesis compared to the full flow. For placement, there is a reduction to zero most of the time. Routing is roughly half of the full flow. In *Flow 2*, the speedup observed for synthesis is basically flat in all the cases, and of around 2x, while placement and routing have more varying runtimes, but in the order of up to 20%.

Although the *NoChanges* scenario is arguably less important, from this data it looks like there is a lot of room for improvement in current incremental commercial flows. In theory, it should be relatively easy to detect, at least after synthesis that no changes are necessary in the physical implementation. This is an unexpected result, given that the reports for both flows show that over 99% of cells and nets were reused from the original to the new implementation. Thus, it looks like there is a lot of time spent in matching which cells and nets can be reused, which eventually reduces the gains from the incremental synthesis.

7. CONCLUSION

We present ANUBIS, a set of RTL designs and code changes, the first benchmark set intended to be used for incremental synthesis, placement and routing. In this initial version, ANUBIS is a small set of designs, but with a rich collection of changes that represent real code changes applied to those designs during time. ANUBIS considers both runtime and QoR to generate a final unified score that allows to easily compare multiple flows.

Incremental synthesis has been the subject of various research in the past and has gained traction in the industry as a path to reduce the synthesis time, which is recognized as one of the main bottlenecks in digital design. Other research areas may also leverage ANUBIS, such as incremental timing analysis tools.

ANUBIS was evaluated using two incremental commercial flows. Although there are not many public available details on how those flows are implemented, they are more focused on keeping QoR, since rather no QoR degradation was observed. This comes with a cost in runtime, which was relatively high when considering the high re-utilization of the designs. Other approaches, such as *LiveSynth* [18], advocate for small QoR degradation for more aggressive runtime reduction. In that case, authors argue for the use of incremental steps while the code is being changed, and full synthesis to recover QoR, when there is no code change being performed.

As new benchmarks with code changes become available, they will be added to future versions of ANUBIS, we are particularly interested in larger designs that could help on the study of the scalability of incremental flows and possibly reflect better industrial designs. We are also interested in improving the scoring system as new discussion and incremental flows emerge.

Acknowledgments

We like to thank the reviewers for their feedback on the paper. Special thanks to Daphne Gorman for suggesting the name ANUBIS. This work was supported in part by the National Science Foundation under grants CNS-1059442-003, CNS-1318943-001, CCF-1337278, and CCF-1514284. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF.

8. REFERENCES

[1] S. N. Adya, M. C. Yildiz, I. L. Markov, P. G. Villarrubia, P. N. Parakh, and P. H. Madden. Benchmarking for large-scale placement and beyond. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(4):472–487, April 2004.

[2] C. Albrecht. IWLS 2015 benchmarks, jun 2005. `http://iwls.org/iwls2005/benchmarks.html`.

[3] Altera Inc. Quartus prime standard edition handbook volume 1: Design and synthesis. `https://www.altera.com/en_US/pdfs/literature/hb/qts/qts-qps-handbook.pdf`, Mar 2016.

[4] V. Bertacco, T. Austin, and I. Wagner. Bug underground. `http://bug.eecs.umich.edu/`.

[5] F. Brglez, D. Bryan, and K. Kozminski. Combinational profiles of sequential benchmark circuits. In *Circuits and Systems, 1989., IEEE International Symposium on*, pages 1929–1934 vol.3, May 1989.

[6] F. Brglez and H. Fujiwara. A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran. In *Proceedings of IEEE Int'l Symposium Circuits and Systems (ISCAS 85)*, pages 677–692. IEEE Press, Piscataway, N.J., 1985.

[7] D. Chen and D. Singh. Line-level incremental resynthesis techniques for fpgas. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 133–142, New York, NY, USA, 2011. ACM.

[8] K. Constantinides, O. Mutlu, and T. Austin. Online design bug detection: Rtl analysis, flexible mechanisms, and evaluation. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 282–293, Washington, DC, USA, 2008. IEEE Computer Society.

[9] S. Davidson. Characteristics of the itc99 benchmark circuits. In *IEEE International Test Synthesis Workshop (ITSW)*, 1999.

[10] M. E. Dehkordi, S. Brown, and T. Borer. Modular partitioning for incremental compilation. In *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pages 1–6, Aug 2006.

[11] T. W. Huang and M. D. F. Wong. Opentimer: A high-performance timing analysis tool. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 895–902, Nov 2015.

[12] Imagination Inc. MIPSfpga microMIPS core, v1.3, 2016. `https://community.imgtec.com/downloads/mipsfpga-getting-started-v1-3/`.

[13] N. P. Jouppi. Timing analysis and performance improvement of mos vlsi designs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 6(4):650–665, 1987.

[14] P. Y. Lee, I. H. R. Jiang, C. R. Li, W. L. Chiu, and Y. M. Yang. iTimerC 2.0: Fast incremental timing and CPPR analysis. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 890–894, Nov 2015.

[15] OpenRISC. mor1kx - an openrisc processor ip core. `https://github.com/openrisc/mor1kx`.

[16] OpenRISC. Or1200 ip core, 2016. `https://github.com/openrisc/or1200`.

[17] A. Phansalkar, A. Joshi, and L. John. Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite. In *Proceedings of the 34th annual International Symposium on Computer architecture (ISCA)*, pages 412–423, Jun 2007.

[18] R. T. Possignolo and J. Renau. LiveSynth: Towards an interactive synthesis flow. In *53rd Design Automation Conference, Proceedings of the*, Jun 2017.

[19] S. Sinha. Using the clock period constraint to your advantage. `http://www.eetimes.com/document.asp?doc_id=1279254`.

[20] Synopsys Inc. Best practices for high-performance, energy-efficient implementations of the arm cortex-a73 processor in 16-nm finfet plus (16ff+) process technology using synopsys galaxy design platform. Synopsys User Group, SNUG.

[21] C. Wolf. Yosys open synthesis suite. "`http://www.clifford.at/yosys/`", 2016.

[22] Xilinx Inc. Vivado synthesis - strategies for reducing run time. `http://www.xilinx.com/support/answers/62215.html`, 2015.

[23] Xilinx Inc. All programmable 7 Series product selection guide, 2016. `https://www.xilinx.com/support/documentation/selection-guides/7-series-product-selection-guide.pdf`.

[24] Xilinx Inc. Vivado design suite user guide. `http://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_1/ug910-vivado-getting-started.pdf`, Apr 2016.