REPLACE: AN INCREMENTAL PLACEMENT ALGORITHM FOR FIELD PROGRAMMABLE GATE ARRAYS

David Leong, Guy G.F. Lemieux†

Department of Electrical and Computer Engineering
University Of British Columbia
Vancouver, BC, Canada
dvleong@shaw.ca, lemieux@ece.ubc.ca

ABSTRACT

Recompiling a large circuit after making a few logic changes is a time-consuming process. We present an incremental placement algorithm for FPGAs that is focused on extremely fast runtime for changes which can be *localized*. It is capable of handling *multiple changes* across large regions of an FPGA. This is especially useful when used with a floorplan where a modified subcircuit is instantiated several times in the design hierarchy or where several subcircuits are modified. The algorithm is simpler and faster than past approaches because its insertion and legalization steps are based on CPU-efficient shifting steps which do not continuously evaluate the impact of each move on costs. Instead, any lost quality is recovered by a fast, low-temperature anneal at the end. When 35,000 out of 50,000 LUTs are modified, the incremental placement (including fast anneal) is 7 times faster than VPR's "fast placement" from scratch with only 2% quality degradation. The key concepts utilized in the incremental placement algorithm include uses of floor-planning constraints, CPU-efficient CLB shifting, super placement grid and a tuned annealing refinement process.

1. INTRODUCTION

As FPGA capacity increases, the runtime required to compile and fit a design also increases. For today's largest FPGAs, a full recompile often requires several hours to execute the entire FPGA CAD flow. This is time-consuming and in many cases is unnecessary when changes can be localized, even if they are made in modules that are replicated across the design or if several updates are done in parallel to different modules. When only localized changes of a circuit are made, incremental algorithms can speed up the compilation by operating on only the changed portion of the design. Ideally, any incremental recompilation should be as high-quality as compilation from scratch, even after several generations of incremental updates.

This paper presents an incremental placement algorithm, named RePlace, designed to be part of an incremental FPGA CAD flow. RePlace consists of four key ides: **floor-planned** placement of modified CLBs, CPU efficient **CLB shifting**, "thinking outside of the box" **super placement grid**, and a finely tuned **fast annealing** process. The algorithm works with the VPR place-and-route toolset [2]. RePlace is intended to be used when significant but localized design changes are made. As a result, we do not intend RePlace to be used in conjunction with an incremental router because of significant design alterations. Instead, we expect RePlace to be used after a floorplanning tool. RePlace accepts as input a list of FPGA CLB instances to place. It optionally accepts a list of floorplan rectangles, which can be overlapping, as well as a list of CLBs for each floorplan rectangle. If a specified floorplan rectangle is too small, then RePlace dynamically resizes it during placement to make room for the CLBs that belong inside during the incremental placement process. The floorplanning information can be assigned *a priori* by designers, e.g. as part of a very large chip floorplan, or it can be computed on the fly from heuristics using back-annotation from a previous placement.

The RePlace algorithm starts with an initial high-quality placement of the "before" circuit state produced by VPR. Then, RePlace finds a new placement for the "after" circuit state. The algorithm is kept very simple to maximize the runtime improvement; while it can likely be "enhanced" in several ways, we present the most basic algorithm and demonstrate excellent quality results. The algorithm is based on shifting and annealing at the CLB level. New or modified CLBs are placed in the *floorplanned* regions; the floorplan constraint forces changes to remain localized. If no space remains in the region, CPU-efficient shifting expands the floorplanned rectangle, moving other CLBs beyond the legal array bounds into zones termed the super-grid. This is followed by compaction, which shifts remaining whitespace within the legal zone to make space for these moved CLBs so they can be returned to the legal zone. RePlace ends with a fast, low-temperature anneal to improve the final result quality.

There are no standardized benchmarks available to evaluate or compare incremental algorithms for FPGAs. To facilitate this, we developed two sets of benchmarks available for download along with our source code.[1] Each individual benchmark circuit has a "before" and "after" state to mimic an incremental change. The first benchmark set, called Single Region or SR, represents design changes made to one localized portion of an MCNC circuit. The incremental changes in these circuits are created using the Perturber tool [13] which makes synthetic changes to a small, localized region.

† This research has been enabled by the use of WestGrid computing resources, which are funded in part by the Canada Foundation for Innovation, Alberta Innovation and Science, BC Advanced Education, and the participating research institutions. WestGrid equipment is provided by IBM, Hewlett Packard and SGI. This work was also funded by Altera and NSERC.

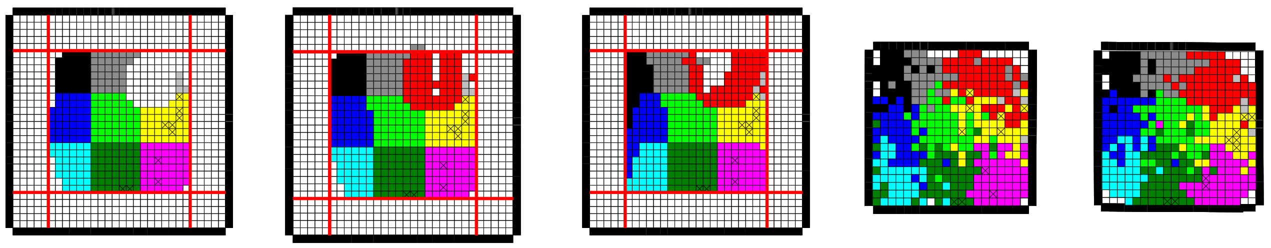
(a) initial placement with cut-out region (white, top right) targeted for replacement **(b) expansion to super-grid, inserting new CLBs (red)** **(c) whitespace compaction to restore legal placement** **(d) final result after low-temperature anneal** **(e) full placement from scratch with locked I/O locations**

Fig. 1. Incremental placement process. CLBs marked with an “X” are connected by the same net.

This benchmark set has the same depth profile information in the before and after circuits – in this way, we isolate the ability of the CAD tool to recreate a good placement with a very similar but perturbed circuit and avoid including the possibly unknown effects of altering circuit depth. While a depth change is more “realistic”, keeping it constant helps better assess “stability” of the tools.

The second benchmark set, called Multi-Region or MR, represents widespread design changes to several localized regions of very large synthetic circuits. The incremental changes in these circuits are created using the Un/DoPack CAD flow [12]. Un/DoPack repeatedly performs a complete place-and-route to reduce interconnect congestion, each time spreading the clustering solution of several (10s to 100s) localized regions to eliminate congestion. The benchmark circuits are taken from a single iteration of this CAD flow using different levels of effort for congestion reduction.

The rest of the paper is organized as follows. Section 2 gives background and related work in the area of incremental placement for FPGA and ASIC CAD. Section 3 describes the RePlace algorithm. Section 4 describes the benchmark generation process and the experimental methodology. Section 5 compares RePlace to VPR. Finally, Section 6 presents conclusions and future work..

2. RELATED WORK

Cong and Sarrafzadeh [4] give a high level overview of the problems associated with incremental CAD, including placement. They note two separate needs for incremental placement: to optimize an existing good placement for a new metric, such as power, or for handling the addition and removal of logic or nets. RePlace is designed for the latter situation. Incremental placement algorithms also fall into both FPGA and standard cell problem domains, as described below.

FPGA-based algorithms focus on small-scale changes: they operate on individual LUTs, and evaluate each move based upon timing [10], congestion [11], or combined [9] cost functions. These LUT-at-a-time operations are costly in runtime. For example, ICP is ~8 times faster than VPR for small changes, but slower than VPR for large changes [9]. In contrast, RePlace is ~7 times faster than VPR in “fast” mode even when changes encompass up to 2/3 of the device.

[1] http://www.ece.ubc.ca/~lemieux/downloads

Altera’s Quartus II and Xilinx’s ISE tools advertise full-flow speedups of 2-3 times in incremental mode. However, algorithmic details are proprietary.

Standard cell incremental placement algorithms are often used as legalization steps to resolve cell overlap resulting from global placement or changes like inserting buffers or decoupling capacitors. Approaches [3] and [6] are meant for very small netlist changes as they compute optimal locations and shift patterns for each cell individually. Floorplan shifting is used in [5], but it presumes significant available whitespace between floorplan regions. Diffusion [14], ripple moves [15], and network flow [16] algorithms are all effective approaches, but the algorithms are more complex and likely slower than RePlace. For example, [15] takes ~1hr on problems with only 12,000 cells compared to a runtime of ~1min for RePlace on 50,000 BLEs. Also, [16] is ~6 times faster than full ASIC placement but only ~1% of cells are moved, compared to RePlace being ~7 times faster than full FPGA placement when moving up to 2/3 of 50,000 BLEs. Diffusion [14] is fast, but it computes costs in the inner loop and may sacrifice quality compared to replacing from scratch or performing a final global anneal as done in RePlace.

The key difference between RePlace and other incremental placers is scalability to widespread changes. Most incremental placers evaluate cost functions for every individual move while inserting modified logic and shifting to make space, and this makes them non-scalable. RePlace only considers cost at the end while annealing. This oblivious nature allows it to *quickly* re-place 2/3 of a 50,000 LUT circuit with the *same speedup* as a small change to the circuit (i.e, RePlace is ~7 times faster than fast VPR placement, regardless of the size of the region of change). RePlace works because: (i) floorplan constraints localize changes and prevent global movement of entire “modules”; (ii) FPGA architectures with fixed-length wires are often tolerant to mild placement shifts (although there is a step function in delay when you cross the reach of a routing wire, within the reach of a routing wire delay is very flat and wirelength is unchanged); and (iii) carefully-tuned annealing recovers most of the lost quality from the cost-oblivious shifting steps.

The focus of RePlace is to use CPU-efficient shifting and moves that require **no** complex movement cost computation. In terms of algorithmic time complexity analysis of RePlace, we note that runtime scales with the number of cells like VPR (which was shown to be roughly O($N^{4/3}$)) and independent of the number of moved cells. However, we believe that final algorithm runtime (which includes constant factors) is far more important than theoretical scalability.

The previous approaches described here tend to focus on solving the problem of overlaps: they start with an initial best but illegal placement, then iteratively resolve the illegal locations using different schemes until a valid placement is produced. In comparison, RePlace approaches the problem differently by disallowing CLB overlaps. Instead, CPU-efficient shifting is used to shift partial rows or columns of CLBs out of the way to immediately create more whitespace for insertion. The shifting creates an expanded placement grid that is then compacted and annealed. Differences in reporting styles and lack of standardized benchmarks make further comparison between incremental methods difficult. By releasing our benchmarks and source code for the VPR tool flow, we hope to encourage further improvements to incremental placement efforts.

The RePlace algorithm was previously named iPlace in [1].

3. ALGORITHM

The RePlace algorithm was designed with the fastest possible runtime in mind. The purpose is to speed up the iterative place-and-route process used after back annotation of physical information to improve overall quality of results in flows such as Un/DoPack [12]. The algorithm initially preserves *spatial locality* in the original placement by keeping previously-placed *unmodified* CLBs close to their same *relative order* when possible. The placer also employs *simplicity* by avoiding the use of heavy computation or cost function evaluations in the first three steps. This simplicity is the key to fast runtime. A carefully-tuned, low-temperature anneal in the final step recovers all lost quality from the first 3 steps. Despite this simplicity and the use of annealing, the quality of results and total runtime is remarkably good. The four steps to RePlace are as follows:

1. Initial Placement, Floorplanning, and Re-clustering
2. Super-grid Expansion Placement
3. Compaction (Re-legalization)
4. Refinement by Low-Temperature Annealing

The first step re-clusters new LUTs into *modified* CLBs and pre-places the other *unmodified* CLBs. The second step pre-places *modified* CLBs, sometimes shifting partial rows/columns of preplaced CLBs by one position to create whitespace (empty CLBs) where needed; this preserves relative ordering. The third step re-legalizes by shifting remaining whitespace to zones where placement has expanded outside of the legal array bounds of the FPGA. For fast runtime, all shifts are deliberately done without checking potential impact to cost functions. While these steps can be made more resource-aware to potentially result in higher-quality results, they are deliberately left "dumb and simple" here to demonstrate that the final annealing step can recover all lost quality.

The RePlace algorithm is implemented in the VPR framework [2]. Three inputs are required for the incremental placement process: i) an initial placement from the "before" circuit state; ii) a *floorplan* or rectangular region identifying approximately where to place the changed elements – this can be provided or automatically computed from the bounding box; and iii) the modified or "after" circuit state. RePlace identifies which CLBs are *modified* and which are *unmodified* by comparing the first and third input data. It can automatically compute floorplan constraints from bounding boxes in the "before" state. For multiple changed regions, the input data for steps (i) and (iii) must subdivide the CLBs into per-region changes.

3.1. Initial Placement, Floorplanning and Re-clustering

The first step of RePlace provides initial placement for all *unmodified* CLBs by examining the placement solution of the "before" circuit state. This step is pictorially shown in Figure 1(a). The colored CLBs represent unmodified CLBs; these are initially placed in their previous placement locations to maintain spatial locality. The CLBs are colored according to their original location within a coarse 3x3 grid in the FPGA array. In this example, one change region has been selected as a circular region in the top-right. Here, the old CLBs have been removed, leaving behind whitespace within that region to be filled in later by *modified* CLBs.

The bounding box of any holes left behind by the removed CLBs is identified by RePlace as a single rectangular *floorplan region*. To support multiple change regions, the "before" and "after" circuits must be subdivided into per-region changes as part of the input specification; RePlace cannot determine these automatically. However, RePlace will compute a corresponding bounding box for each change region. While the methods used to subdivide these change regions are beyond the scope of this paper, they can be identified through bounding boxes calculated using the design hierarchy or a design floorplan, for example. For the multiple region benchmarks in this paper, each change region is identified and subdivided ahead of time by the Un/DoPack flow.

Last, *modified* CLBs are produced by an incremental re-clustering step with a modified version of iRAC [8]. An incremental design change can result in the deletion of old LUTs and the creation of new LUTs. Any "before" CLB containing deleted LUTs is fully broken down (unclustered) into its constituent LUTs and added to the list of new LUTs. Other CLBs (i.e., without deleted LUTs) are left intact and considered *unmodified* CLBs; these CLBs are used as a partial starting solution for iRAC. iRAC greedily extends this partial solution into a complete, timing-driven clustering solution by re-clustering all new and unclustered LUTs into *modified* CLBs. This re-clustering step is performed separately for each region of change.

3.2. Super-grid Expansion Placement

The second step of RePlace is placement of modified CLBs. This step randomly places each modified CLB in the free whitespace of its corresponding floorplan region. If some CLBs exist that do not belong to any floorplan region, they are randomly placed in the remaining space. Pseudocode for this step is shown in Figure 2, and Figure 1(b) shows a real example.

```
initial_placement()
shift = 0
   for each change region r with floorplan f {
      for each modified CLB c of region r {

         if no remaining free space within f {
            shift%4 == 0 ? shift right by 1
            shift%4 == 1 ? shift up by 1
            shift%4 == 2 ? shift left by 1
            shift%4 == 3 ? shift down by 1
            shift++
         }
         randomly place c within free space of f
   }
}
randomly_place_any_remaining_clbs()
```

Fig. 1. Super-grid expansion pseudocode.

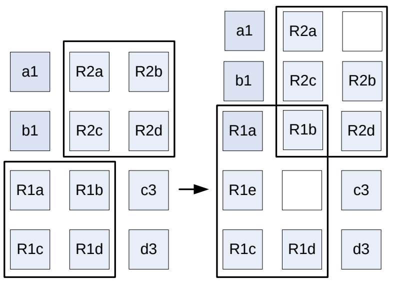

Fig. 1. Multi-region floorplan handling.

If the floorplan region runs out of whitespace when a modified CLB is being inserted, an expansion of that floorplan region is triggered. Expansion is done in a round-robin/spiral fashion, one complete side/direction at a time. When a side is expanded, the floorplan rectangle increases in size by 1 CLB unit in that direction. To make space, partial rows or columns of previously-placed CLBs along that same side are shifted over 1 unit starting from the centerline of the region; shifting in this manner preserves relative placement of the CLBs along the column or row. When shifting, any other affected floorplan rectangles are also extended by 1 unit in the same direction as the shift. For example, the floorplan region *R1* in Figure 3 must be expanded to create room for CLB *R1e*. The two columns on the top side are shifted up, starting at the centerline of the floorplan region, creating 2 spaces and increasing the floorplan to a 2 × 3 region. The nearby floorplan region *R2* is affected by the upward shift of *R1a* and *R1c*, so it is also extended upwards to become a 2 × 3 region. Afterwards, regions *R1* and *R2* overlap, which is permitted. As a side effect of extending region *R1*, region *R2* now has some extra whitespace. In general, the amount of shifting required is quite modest. For example, to expand a 5 × 5 CLB region by 20%, only one shift on one side is required to make it 5 × 6. The limited shifting helps maintain locality.

This shifting process may place CLBs outside the device boundaries into zones we call the *super-grid*. The super-grid expands as needed to hold the relative order of CLBs. This allows the placement algorithm to avoid calculations that would be required to re-shuffle whitespace more carefully. Note that I/O locations at the super-grid periphery just shift outwards but are not reordered or increased in number. The super-grid represents an illegal placement, so the next step involves re-legalization in the form of compaction.

Note regarding hard macro blocks and carry chains
Real-world FPGA placement constraints such as carry chains, large blocks of memory, or stripes of memories/multipliers interfere with shifting. For small obstacles, one can imagine shifting through them (to the other side). For larger obstacles, they can be considered illegal areas, like the super-grid, which allow CLBs to be shifted outside and resolved later by compaction. This approach is further discussed in [1] but omitted here due to space limitations. Lack of support for carry chains and similar obstacles in VPR means these constraints are not implemented or tested in the RePlace code.

3.3. Compaction (Re-legalization)

The third step of RePlace is to re-legalize CLBs placed outside the valid placement area defined by the FPGA array size. One method to re-legalize all CLBs is to use annealing, but we found this approach too slow. Instead, we created a simple and fast solution called compaction. The pseudocode for compaction is shown in Figure 4. Figure 1(c) shows a real example.

Compaction divides the entire super-grid into 9 zones like a "#" symbol with the legal placement zone at the centre. This is visually depicted in Figure 5 where the center green

```
for each illegal zone s {
   if s is corner
      shift all free space to corner
      randomly move illegal CLBs to free space
   else if s is side
      shift all free space to side s
      find average location of illegal CLBs
      shift all free space to average location
      randomly move illegal CLBs into free space
   end if
}
```

Fig. 2. Compaction pseudocode.

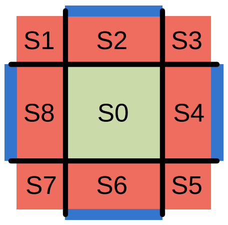

Fig. 3. Compaction areas.

zone S0 is the legal placement area. The red zones S1-S8 are illegal locations to be fixed. The peripheral blue areas depict the I/O locations. Compaction legalizes each of the four corners and four sides, one zone at a time in random order[2], by shifting *all available* whitespace in the center zone to a legal location nearby the illegal CLBs and then moving them into these legal whitespace locations. The precise destination is random.

For example, the top-center zone S2 is legalized by first shifting all whitespace vertically to the top rows as if all legal CLBs "fall" to the bottom of S0 due to gravity. Then, the average horizontal location of the illegal CLBs in the S2 zone is used as a dividing line. All whitespace in S0 is shifted towards this line as follows: all legal CLBs on the left of this line "fall" to the far left and all right CLBs "fall" to the far right. This places all whitespace just below the majority of illegal CLBs, allowing them to be randomly moved to S0.

3.4. Refinement by Low-Temperature Annealing

The last step of RePlace is to improve quality with a carefully-tuned, low-temperature anneal. After compaction, we found that the average bounding box and critical path delays were not ideal. In many cases, the bounding box cost was 20% higher than placement from scratch. To avoid significant placement alterations, we re-tuned various parameters within the simulated annealing algorithm of VPR. To limit hill climbing, the *initial temperature* was lowered so that fewer "bad" swaps would be accepted. To maintain spatial locality, the initial *range window* was lowered to focus the swaps within a more localized area. To reduce and control runtime, the number of swaps per temperature parameter, *inner_num*, and the temperature reduction factor, *alpha*, were also tuned. A series of experiments were conducted to determine the following parameter values:

- Initial temp. of 44% acceptance rate from prev. placement
- Initial window range (*rlim*) of 12.5% of the FPGA width Temperature reduction factor alpha of 0.7
- Number of swaps per temp. range, *inner_num* of 1 to 3

Full details of this tuning are described in [1]. This produces a good, high-quality result that is comparable to a full placement. Runtime is controllable via the *inner_num* parameter.

4. EXPERIMENTAL METHODOLOGY

This section describes the experimental framework and the circuits used to benchmark the incremental placement algorithm.

4.1. Single-Region Benchmarks (SR)

The first SR benchmark set is designed to test the performance of the incremental placer with a single region of localized, modified logic. These are simple test cases that any incremental placer should handle.

The SR benchmark characteristics are given in Table 1 for five MCNC circuits. Similar results for the other 15 traditional circuits are reported in [1]. Columns in the table give the number of BLEs (#LE) and number of CLBs after clustering (#CLB) for the original and modified versions. Also given is the number of synthetically generated BLEs representing a design change in each modified circuit (#syn LE).

These circuits were generated as follows. An MCNC circuit is used as a starting point for the "before" circuit state. It is then modified using the Perturber tool [13] to produce five "after" circuit states. One random rectangular floorplan region, consisting of 2.5%, 5%, and 10% of the total CLBs, is selected for each circuit. Each region is replaced with synthetically modified logic, either identical- or double-sized, to generate 5 "after" states (the 10% change region is not doubled). The identical-size cases are simple tests, while the double-size cases test expansion. The circuit depth of the synthetically generated sub-circuit is unchanged to provide a stable comparison between the baseline and modified circuits. This was done to simulate incremental changes such as a small error correction.

4.2. Multi-Region Benchmarks (MR)

The second MR benchmark set is designed to test performance of the incremental placer with multiple regions of localized, modified logic. This represents cases where significant changes to multiple parts of a circuit are made to more fully stress the incremental recompilation.

The MR benchmark characteristics are given in Table 2. For each benchmark, the total CLB count (#CLB), number of modified CLBs (□CLB) and the number of changed regions are shown.

This set of benchmarks is generated using the Un/DoPack flow [12]. Each circuit is **over 50,000** 4-input LUTs in size, initially clustered into CLBs of 16 LUTs per CLB. This flow iteratively runs place-and-route many times to reduce interconnect congestion. Each region that exceeds a target channel width is re-clustered to use more CLBs (by inserting empty LUTs to spread out the region), and then re-placed and re-routed. If congestion persists, the flow iterates again. Rapid incremental placement is required for this flow to execute quickly. The "before" MR circuits are 3 large, synthetically generated circuits from [12]. Five "after" versions of each circuit with multiple regions of change are created by targeting a 10%, 20%, 30%, 40% or 50% reduction in routing channel width in the first pass of Un/DoPack. *The first congested region* is created by choosing the most-congested CLB and selecting all CLBs within radius 5 that have not been previously selected. A floorplan for this region is formed by the bounding box of selected CLBs. The region is marked for re-clustering from 16 down to 13 LUTs per CLB, representing 23% more CLBs. *Additional congestion regions* are created by iterating over the remaining CLBs until all CLBs with a channel width above the target width have been selected. Each CLB belongs to just one local congestion region, but the floorplanned regions can overlap.

4.3. Experimental Process

The incremental placement CAD flow is implemented as part of VPR with the following parameters and settings:

- Initial clustering uses iRAC, initial placement uses VPR 4.30
- 4-input LUTs, cluster size N=10/16 for SR/MR, buffered L4 wires
- VPR flags: `–verify_binary_search –pres_fac_mult 1.3 –max_ router_iterations 100`, and for the final route with 20% more tracks: `–pres_fac_mult 1.1`
- Runtimes include placement only; initialization time is excluded
- Runtime measurements use an Intel P4, 3GHz, 512MB RAM
- Low-temperature annealing parameters from Section 3.4
- Annealing results are an arith. average of 5 different starting seeds

[2] In retrospect, this random zone order will slosh the whitespace across the die many times (once for each zone). Each slosh shears the locations of many preplaced CLBs. It may be better to visit the 8 zones in a specific order to reduce sloshing.

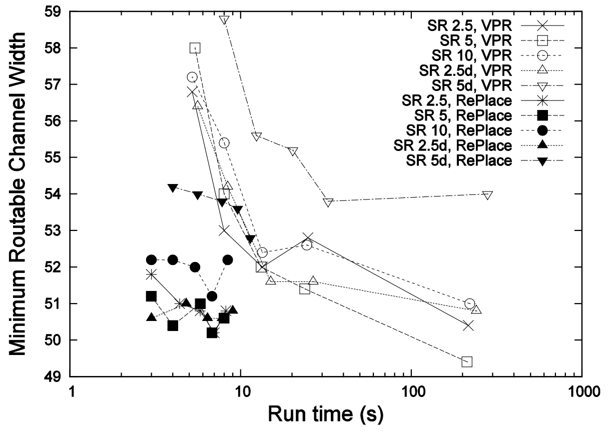

Fig. 1. Routing/Runtime, CLMA(SR).

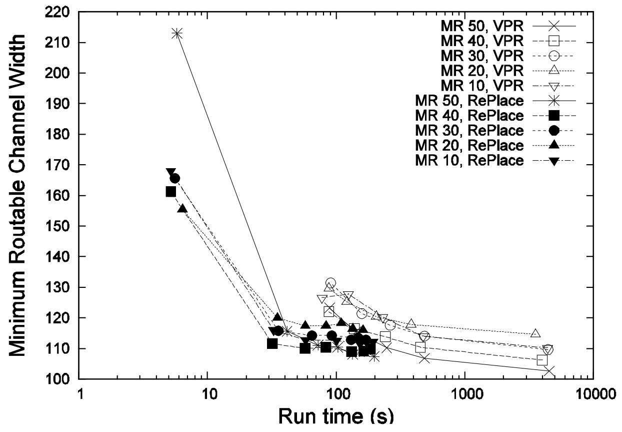

Fig. 2. Routing/Runtime, Stdev010(MR).

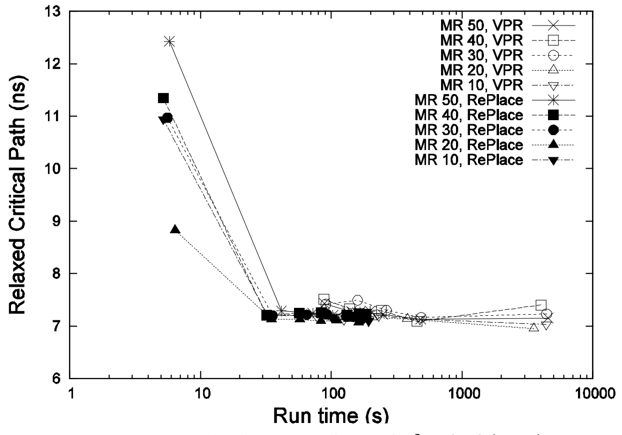

Fig. 3. Timing/Runtime, Stdev010(MR)

The placement speed of RePlace was varied by setting the *inner_num* annealing parameter to 3, 2.5, 2, 1.5, and 1. Reducing this value reduces the number of swaps that are performed at each temperature. Lower values result in faster runtimes, but this does not significantly affect quality. RePlace uses a default value of *inner_num=1*.

The placement speed of VPR was varied by setting its *inner_num* parameter to 10, 1, 0.5, 0.25 and 0.125. An *inner_num* value of 10 is the "default" value for VPR. An *inner_num* value of 1 is set when VPR 4.30 is invoked with the "fast" placement option. This produces slightly lower-quality placements but improves runtime nearly 10-fold. We created a new "superfast" VPR placement option that sets *inner_num* to 0.125. Various other VPR parameters such as initial temperature, range limit etc. were also studied to determine the reduction of runtime versus placement quality trade-off. It was found that reducing *inner_num* provides the most graceful degradation of placement quality versus runtime improvement.

5. EXPERIMENTAL RESULTS

Figures 6 and 7 show routability versus runtime trade-offs for CLMA from the SR set and Stdev010 from the MR set, respectively. Figure 8 shows critical path versus runtime trade-off curves for Stdev010 from MR. The vertical axis is routing quality, either the minimum channel width needed to route or critical path. The horizontal axis is runtime on a $\log_{10}$ scale. The left-most (fastest) data markers for RePlace in Figures 7 and 8 are the results when RePlace skips the fast anneal. VPR results are drawn using open-box markers and RePlace results are drawn using solid-box markers. RePlace significantly outperforms default VPR by nearly two orders of magnitude in runtime, yet achieves comparable quality. VPR quality degrades 10–20% across the performance range, but RePlace degrades only about 5% (except when annealing is skipped). Results were similar for the other benchmarks (not shown).

An ultra-rapid incremental placement based entirely on shifting CLBs and skipping the anneal is not recommended due to *significant quality loss*. However, these results show that the quality is not lost forever; it can be restored with a rapid anneal. We include one more data point in Figures 7 and 8 for RePlace *inner_num=0.5* showing that an even faster anneal *still recovers all lost quality*. We also tried adding a 2nd annealing step between expansion and compaction, but this only increased runtime and did not improve quality at all. It isn't clear if this is because most of the quality is lost due to compaction, or if a single anneal is sufficient to recover all lost quality. Either way, since the quality is recoverable by one anneal that is sufficiently fast, it is likely unnecessary to spend additional runtime in the expansion or compaction phase to make "carefully evaluated" shifting decisions. This is encouraging because it shows that extremely robust incremental placement can be done with very simple heuristics.

Tables 3 and 4 give more precise results for VPR and RePlace. Speedup columns are normalized runtimes. Due to measurement precision, runtimes <200ms are reported as 0s and excluded from speedups. Columns CW, CP, BB, WL are channel width, critical path delay, placement bounding box, and routed wirelength. Columns ending in Q or Quality are normalized to RePlace, so values > 1.0 indicate RePlace is better. RePlace is within 4% of VPR full placement quality, but with ~60-fold speedup. It is within 2% of VPR's "fast" placement quality with ~7-fold speedup. VPR in "superfast" mode degrades quality 11-14% on average and does not achieve the same speed as RePlace; this shows that rapid annealing alone is not sufficient for incremental placement.

It is worth noting that the quality of multi-region incremental placement does not degrade even when a substantial portion of the circuit is modified. In particular, the MR-50 set of circuits have 1/3 to 2/3 of the CLBs modified in a 50,000 LUT circuit, but RePlace is still able to produce quality results with similar 60-fold/7-fold speedups. This ability to tolerate widespread but localized changes is what makes RePlace an ideal "fast placement" tool for Un/DoPack.

Speedups and high quality are obtained with RePlace because of its ability to preserve the original placement information. In particular, the floorplanning and shifting ensures that modified CLBs are initially placed nearby the original CLBs they are replacing, without disturbing relative placement locality across the device. If the initial placement of modified CLBs was completely unconstrained, the annealing step would need many more swaps to achieve the same effect via the global migration of many more CLBs. Instead, the fast anneal at the end with a narrow range limit can focus its moves on quality improvement via localized CLB movement rather than global CLB movement.

6. CONCLUSIONS

RePlace is a fast incremental placement algorithm. The ideas contributing to its speed include the use of *floorplanning*, a placement *super-grid*, CPU-efficient *CLB shifting* which performs no detailed cost calculations, and *rapid annealing* to restore lost quality. It was shown that simply shifting alone, or just speeding up annealing alone, are insufficient by themselves to achieve the same quality and runtime benefits. Adding better cost-aware shifting steps could be imposed, but it is not clear they are needed from a quality or runtime perspective. While such changes may further improve the results, the minor improvements they offer to intermediate results may also simply be lost when annealing. Instead, it is important to note that the "dumb and simple" algorithm presented here is sufficiently robust without this added complexity.

RePlace achieves speedups roughly 70-fold compared to default VPR for single region changes encompassing up to 10% of small designs with no lost quality. Compared against VPR's "fast" mode, RePlace is still about 8 times faster. Compared to a new "superfast" mode created for VPR, RePlace is almost 2 times faster and much better in quality. Even on large designs of 50,000 LUTs where up to two-thirds of a circuit is modified, RePlace maintains similar speedups and performs only 2-4% worse in quality. This shows that RePlace is capable of scaling to situations where significant circuit modifications are made.

Incremental placement algorithms that can quickly incorporate logic design changes is even more important for FPGAs than ASICs. Future work for the RePlace algorithm includes extension to handle macro blocks and carry chains. Additional benchmarking can be done with real world examples, including small logic changes and large scale changes such as updating a large subcomponent or updating a small subcomponent replicated many times throughout the design. A detailed comparison of quality and runtime results against other incremental placement algorithms should also be done; we are facilitating this by releasing our code and benchmarks online.

7. REFERENCES

[1] D. Leong, "Incremental Placement for Field-Programmable Gate Arrays," M.A.Sc. Thesis, Dept of ECE, University of British Columbia, Nov., 2006.

[2] V. Betz, J. Rose, S. Marquardt Architecture and CAD for Deep-Submicron FPGAs, Kluwer, Feb., 1999.

[3] C. Choy, T. Cheung, K. Wong, "Incremental Layout Placement Modification Algorithms," Trans. CAD, pp. 437-445, Apr 1996.

[4] J. Cong, M. Sarrafzadeh, "Incremental Physical Design," International Symposium on Physical Design, pp. 84-92, 2000.

[5] J. Li, J. Yu, H. Miyashita, "An Incremental Placement Algorithm for Building Block Layout Design Based on the O-Tree Representation," IEICE Trans. Fundamentals, vol. E88-A, no. 12, pp 3398-3404, 2005.

[6] Z. Li, W. Wu, X. Hong, J. Gu, "Incremental Placement Algorithm for Standard Cell Layout," IEEE ISCAS, vol.2 pp. 883-886, 2002.

[7] A. Marquardt, V. Betz, J. Rose, "Using Cluster-Based Logic Blocks and Timing-Driven Packing to Improve FPGA Speed and Density," FPGA, pp. 37-46, 1999.

[8] A. Singh and M. Marek-Sadowska, "Efficient Circuit Clustering for Area and Power Reduction in FPGAs," FPGA, 2002.

[9] D. Singh, S. Brown, "Incremental Placement for Layout Driven Optimizations on FPGAs," ICCAD, pp. 752-759, 2002.

[10] P. Suaris, L. Liu et al, "Incremental Physical Resynthesis for Timing Optimization," FPGA, pp. 99-108, 2004.

[11] N. Togawa, K. Hagi, M. Yanagisawa, "An Incremental Placement and Global Routing Algorithm for Field Programmable Gate Arrays," ASP-DAC, 1998.

[12] M. Tom, D. Leong, G. Lemieux, "Un/DoPack: Re-Clustering of Large System-on-Chip Designs with Interconnect Variation for Low-Cost FPGAs", ICCAD, 2006.

[13] D. Grant, G. Lemieux, "Perturb+Mutate: Semi-Synthetic Circuit Generation Incremental Placement and Routing", ACM TRETS, Sept. 2008.

[14] H. Ren, D. Pan, C. Alpert, P. Villarrubia, "Diffusion-based Placement Migration," DAC, pp. 515-520, 2005.

[15] M. Hrkic, J. Lillis, G. Beraudo, "An Approach to Placement-Coupled Logic Replication," DAC, pp. 711-716, 2004.

[16] S. Dutt, H. Ren, F. Yuan, V. Suthar, "A Network-flow Approach to Timing-driven Incremental Placement for ASICs," ICCAD, 2006.

Table 1. Single-region benchmark characteristics (10 BLEs per CLB).

	Original		Synthetic 2.5			Synthetic 5			Synthetic 10			Synthetic 2.5d			Synthetic 5d		
	#LE	#CLB	#LE	#syn LE	#CLB	#LE	#syn LE	#CLB	#LE	#syn LE	#CLB	#LE	#syn LE	#CLB	#LE	#syn LE	#CLB
CLMA	8383	839	8384	251	840	8383	490	839	8385	992	840	9032	980	904	9613	1966	962
EX1010	4598	460	4598	150	460	4599	251	461	4598	490	460	4752	304	476	4850	502	486
MISEX3	1397	140	1397	30	140	1397	147	140	1397	247	140	1460	93	147	1489	182	150
PDC	4575	458	4576	151	459	4575	490	458	4575	630	458	4829	504	484	4961	736	497
SPLA	3690	369	3691	91	370	3691	251	370	3692	492	370	3787	187	379	3907	417	391

Table 2. Multi-region benchmark characteristics (16 BLEs per CLB).

		Multi Region - 50			Multi Region - 40			MR - 30			MR - 20			MR - 10		
	org. #CLB	New #CLB	ΔCLB	Num. Regions	New #CLB	ΔCLB	Num. Regions	New #CLB	ΔCLB	Num. Regions	New #CLB	ΔCLB	Num. Regions	New #CLB	ΔCLB	Num. Regions
CLONE	3151	3618	2233	135	3310	762	46	3265	560	29	3206	275	12	3288	681	34
STDEV0	3148	3603	2218	114	3595	2208	114	3606	2224	116	3272	617	30	3370	1087	50
STDEV010	3152	3463	1490	85	3278	588	37	3254	490	29	3193	202	9	3237	425	20

Table 3. RePlace post-routing results (SR benchmarks).

Single-Region Circuit	RePlace inner_num=1					VPR (default) norm. to RePlace					VPR (fast) norm. to RePlace					VPR (superfast) norm. to RePlace				
	RT (s)	CW	CP (ns)	Bbox	WL $\times 10^4$	Speedup	CWQ	CPQ	BBQ	WLQ	Speedup	CWQ	CPQ	BBQ	WLQ	Speedup	CWQ	CPQ	BBQ	WLQ
clma p25	3	51.8	27.0	528.8	6.76	**72.00**	0.97	0.95	0.98	0.97	**8.27**	1.02	0.94	1.01	1.00	**1.73**	1.10	0.96	1.11	1.10
clma p5	3	51.2	26.5	529.5	6.78	**70.80**	0.96	0.97	0.99	0.98	**7.93**	1.00	0.97	1.01	1.00	**1.80**	1.13	0.98	1.13	1.12
clma p10	3	52.2	26.9	536.9	6.88	**73.53**	0.98	0.96	0.98	0.98	**8.13**	1.01	0.93	1.01	0.99	**1.73**	1.10	0.96	1.11	1.10
clma p25d	3	50.6	30.9	572.7	7.23	**80.27**	1.00	0.89	0.98	0.97	**8.87**	1.02	0.89	1.00	0.99	**1.87**	1.11	0.90	1.09	1.09
clma p5d	4	54.2	32.0	644.4	8.26	**69.95**	1.00	0.99	0.98	0.97	**8.15**	0.99	0.96	0.99	0.98	**2.00**	1.08	0.95	1.08	1.08
ex1010 p25	1	47	17.8	277.8	3.66	**77.60**	0.99	0.91	0.99	0.99	**9.20**	1.04	0.95	1.01	1.01	**2.20**	1.06	0.95	1.05	1.04
ex1010 p5	1	46.6	16.9	276.3	3.63	**75.00**	0.99	0.95	0.99	1.00	**8.60**	1.03	0.95	1.00	1.01	**2.00**	1.12	1.01	1.08	1.09
ex1010 p10	1	46.4	16.8	277.3	3.64	**77.00**	1.03	0.97	0.99	0.99	**8.80**	1.03	0.98	1.00	1.00	**2.00**	1.06	0.97	1.05	1.05
ex1010 p25d	1.2	46.2	18.0	299.5	3.95	**69.00**	1.00	1.04	0.98	0.97	**8.17**	1.03	1.05	1.00	0.99	**2.00**	1.09	0.96	1.07	1.06
ex1010 p5d	1	47	17.1	279.1	3.68	**76.20**	1.01	0.99	0.99	0.99	**8.80**	1.01	0.95	1.01	1.00	**2.20**	1.08	0.95	1.06	1.06
misex3 p25	0	37.4	11.4	71.1	0.94	**-**	1.01	1.17	0.99	0.98	**-**	1.02	1.02	1.00	1.00	**-**	1.04	1.18	1.04	1.05
misex3 p5	0	37.6	13.7	71.3	0.94	**-**	0.99	0.83	0.99	0.99	**-**	1.01	0.91	1.00	1.00	**-**	1.05	0.95	1.04	1.06
misex3 p10	0	37.6	11.7	71.2	0.94	**-**	0.99	1.09	0.99	0.99	**-**	1.00	0.98	1.00	1.00	**-**	1.06	1.09	1.04	1.05
misex3 p25d	0	38.6	13.3	72.1	0.96	**-**	0.97	0.85	0.99	0.98	**-**	0.98	1.03	0.99	1.00	**-**	1.03	1.02	1.03	1.03
misex3 p5d	0	37.6	13.7	81.0	1.08	**-**	0.99	1.44	0.98	0.99	**-**	0.99	0.97	0.99	1.00	**-**	1.05	1.00	1.04	1.05
pdc p25	1	61.4	19.8	348.6	4.67	**80.60**	1.00	1.08	0.99	0.97	**9.80**	0.99	0.96	1.00	1.00	**2.20**	1.07	1.06	1.06	1.06
pdc p5	1.2	62	18.9	348.4	4.64	**64.00**	0.98	1.27	0.99	0.98	**7.00**	0.97	1.35	1.00	0.99	**1.83**	1.07	1.35	1.07	1.07
pdc p10	1.2	60.6	21.0	347.2	4.65	**68.67**	1.01	0.96	0.99	0.97	**7.00**	1.03	0.95	1.01	1.00	**2.00**	1.09	1.00	1.08	1.06
pdc p25d	1	61.2	25.2	367.6	4.93	**84.40**	0.98	0.78	0.98	0.98	**10.20**	0.99	0.76	0.99	0.98	**2.60**	1.05	0.81	1.05	1.05
pdc p5d	1.4	61.6	23.0	402.8	5.34	**68.14**	1.01	0.97	0.99	0.98	**7.86**	1.03	1.06	1.01	1.00	**1.86**	1.08	0.96	1.07	1.07
spla p25	0.6	51.6	17.3	230.6	3.11	**75.67**	0.98	0.96	0.99	0.99	**8.67**	1.00	1.23	1.01	1.01	**2.33**	1.08	1.11	1.09	1.09
spla p5	0.8	51.8	19.2	230.8	3.13	**55.50**	0.99	0.90	1.00	0.98	**6.75**	0.99	0.91	1.01	1.00	**1.50**	1.07	0.96	1.10	1.09
spla p10	1	51	17.7	230.4	3.13	**44.80**	1.01	0.97	1.00	0.98	**5.00**	1.02	1.07	1.01	1.00	**1.20**	1.09	0.97	1.08	1.08
spla p25d	0.6	49.8	19.6	250.5	3.37	**84.00**	1.03	0.92	1.00	1.00	**9.67**	1.04	0.92	1.01	1.01	**3.00**	1.09	1.02	1.09	1.08
spla p5d	1.2	53.8	20.8	289.1	3.84	**51.17**	0.98	0.96	0.98	0.97	**6.00**	1.01	1.04	1.00	0.99	**1.50**	1.09	0.99	1.07	1.06
Geo. Mean	NA	49.2	19.1	252.6	3.33	**70.08**	0.99	0.98	0.99	0.98	**8.04**	1.01	0.98	1.00	1.00	**1.94**	1.08	1.00	1.07	1.07

Table 4. RePlace post-routing results (MR benchmarks).

Multi-Region Circuit	RePlace inner_num=1					VPR (default) norm. to RePlace					VPR (fast) norm. to RePlace					VPR (superfast) norm. to RePlace				
	RT (s)	CW	CP (ns)	Bbox $\times 10^3$	WL $\times 10^5$	Speedup	CWQ	CPQ	BBQ	WLQ	Speedup	CWQ	CPQ	BBQ	WLQ	Speedup	CWQ	CPQ	BBQ	WLQ
clone - 50	72.2	111.0	72.4	4.32	5.30	**62.8**	0.92	0.99	0.92	0.92	**6.7**	0.96	0.98	0.96	0.96	**1.2**	1.11	1.03	1.11	1.09
clone - 40	57.2	110.0	72.5	3.90	4.87	**70.0**	0.97	1.02	0.96	0.96	**7.9**	1.00	0.98	0.99	0.99	**1.5**	1.11	1.04	1.11	1.10
clone - 30	64.8	114.2	72.1	3.87	4.85	**68.3**	0.96	1.00	0.96	0.96	**7.5**	1.00	0.99	0.99	0.98	**1.4**	1.15	1.03	1.14	1.12
clone - 20	57.6	117.4	71.2	3.79	4.77	**61.5**	0.98	0.98	0.96	0.96	**6.6**	1.00	1.00	0.99	0.99	**1.5**	1.10	1.03	1.14	1.11
clone - 10	58.0	112.8	71.9	3.90	4.88	**75.9**	0.98	0.98	0.95	0.95	**8.2**	1.01	0.99	0.99	0.98	**1.3**	1.12	1.00	1.11	1.09
stdev0 - 50	60.8	92.6	74.4	4.27	5.22	**67.3**	0.92	0.95	0.94	0.95	**7.6**	1.00	0.98	0.99	0.98	**1.5**	1.14	0.98	1.10	1.08
stdev0 - 40	63.0	90.6	72.4	4.20	5.15	**66.8**	0.97	0.98	0.97	0.97	**6.9**	1.03	1.00	1.01	1.00	**1.4**	1.21	1.03	1.14	1.11
stdev0 - 30	76.2	92.0	74.2	4.26	5.21	**55.1**	0.96	0.97	0.96	0.96	**5.6**	0.98	0.97	0.98	0.98	**1.1**	1.23	1.01	1.13	1.11
stdev0 - 20	71.0	95.6	74.1	3.91	4.85	**56.0**	0.96	0.95	0.95	0.96	**6.5**	1.00	0.95	0.99	0.99	**1.1**	1.19	0.97	1.13	1.11
stdev0 - 10	59.8	93.6	73.0	4.00	4.96	**66.3**	0.96	0.96	0.95	0.95	**7.1**	1.00	0.96	0.99	0.99	**1.4**	1.17	1.00	1.12	1.10
stdev010 - 50	89.0	140.2	75.8	4.23	5.26	**47.1**	0.97	0.96	0.96	0.96	**4.9**	1.00	0.99	1.00	0.99	**0.9**	1.13	0.98	1.15	1.13
stdev010 - 40	66.0	140.0	74.3	4.04	5.08	**59.7**	0.98	0.97	0.96	0.96	**6.3**	0.99	0.98	0.98	0.98	**1.3**	1.14	1.00	1.14	1.12
stdev010 - 30	63.8	142.0	75.0	4.03	5.07	**68.6**	0.98	0.98	0.96	0.97	**8.0**	1.02	1.00	0.99	0.99	**1.7**	1.10	0.99	1.11	1.10
stdev010 - 20	56.4	150.6	74.0	3.93	4.98	**66.5**	0.99	0.98	0.97	0.96	**7.1**	1.00	0.98	0.99	0.99	**1.4**	1.11	0.99	1.12	1.11
stdev010 - 10	70.4	144.4	74.3	4.01	5.05	**58.5**	0.97	0.97	0.95	0.96	**5.8**	1.00	0.99	0.99	0.98	**1.2**	1.10	0.99	1.11	1.10
Geo. Mean	65.2	114.6	73.4	4.04	5.03	**63.0**	0.96	0.98	0.96	0.96	**6.8**	1.00	0.98	0.99	0.99	**1.3**	1.14	1.00	1.12	1.11