ANALYTICAL PLACEMENT FOR HETEROGENEOUS FPGAS

Marcel Gort and Jason H. Anderson

Dept. of Electrical and Computer Engineering
University of Toronto
email: {gortmarc, janders}@eecg.toronto.edu

ABSTRACT

We present HeAP, an analytical placement algorithm for heterogeneous FPGAs comprised of LUT-based logic blocks, multiplier/DSP blocks and block RAMs. Specifically, we adapt a state-of-the-art ASIC-based analytical placer to target FPGAs with heterogeneous blocks located at discrete locations throughout the fabric. Our placer also handles macros of LUT-based blocks with specific layout requirements, such as carry chains. Results show that our placer delivers a $4\times$ speedup, on average, compared to Altera's non-timing driven flow, at the cost of a 5% increase in post-routed wirelength, and an $11\times$ speedup compared to Altera's timing-driven flow, at the cost of a 4% increase in post-routed wirelength and a 9% reduction in maximum operating frequency. We also compare with an academic simulated annealing-based placer and demonstrate a $7.4\times$ run-time advantage with 6% better placement quality.

1. INTRODUCTION

Since the early-to-mid 2000s, the rate of increase of uniprocessor performance has not kept pace with the growth rate of integrated circuit size. A consequence of this trend is that the computational time required to design the largest ICs has continued to lengthen – a notion termed the *design productivity gap*. Placement is one of the most time-consuming steps of the FPGA CAD flow and is the stage of the flow addressed in this work. Naturally, the main motivation for reducing CAD tool run-time is to raise engineering productivity and thereby lower cost. However, a secondary motivation lies in the context of broadening the application scope of FPGAs to include their use as computing devices targeted by software engineers. Software engineers are accustomed to `gcc`-length compile times – not the hours or days associated with today's commercial FPGA tools. While a number of researchers have targeted placement run-time through parallelizing existing algorithms (e.g. [1]), in this work, we focus on run-time from the algorithmic perspective and present a fast placement approach for FPGAs.

The two main commercial FPGA vendors use considerably different placement algorithms. Altera uses a simulated annealing (SA)-based iterative placement strategy [1], whereas Xilinx employs analytical placement (AP)-based techniques [2]. SA iteratively swaps objects and uses a cost function to decide which swaps should be accepted or rejected. Hill climbing is permitted by way of accepting some moves that increase cost (in the hope that such moves may later lead to a better solution overall). AP, on the other hand, represents the entire placement problem as systems of equations, to which standard solvers are applied. It is interesting that in the custom ASIC domain, where placers must handle designs with millions of cells, the SA strategy has largely been abandoned as a viable approach in favor of analytical techniques, owing to SA's run-time. Despite this, SA remains popular for FPGAs and besides Altera's commercial tools, SA is the basis for the widely-used VPR open-source placement and routing framework [3]. This work applies AP to heterogeneous FPGA placement and explores its potential for run-time reduction over SA.

AP techniques use solvers that require the placer objective function be both continuous and differentiable. It is indeed this requirement that makes AP well-suited to ASICs, where cells of any type can be placed anywhere on a "continuous" die. Commercial FPGAs comprise an array of heterogeneous blocks – each placed at discrete locations on the die. For example, Altera and Xilinx FPGAs contain columns of RAM blocks and DSP/multiplier blocks spaced at intermediate points throughout an array of LUT-based logic blocks [4, 5]. The discrete nature of FPGA placement presents challenges for AP – challenges that we specifically address in this work. While a limited number of works have applied AP to FPGAs comprising solely LUT-based blocks (e.g. [6]), to our knowledge, no published work has considered AP techniques for heterogeneous FPGAs.

In this work, we adapt a recently-published AP algorithm [7] for ASICs, called SimPL, to target FPGAs with heterogeneous blocks and groups of LUT-based logic blocks with fixed layout patterns (for example, carry chains). Our heterogeneous analytical placer (HeAP), uses Altera's Quartus University Interface Program (QUIP) [8] to route designs on the Cyclone II 90nm FPGA family. We show that HeAP produces a post-routed quality-of-result (QoR) close to Quartus II, with considerably less run-time. We also compare with our own SA-based placer and demonstrate a $7.4\times$ speedup with 6% better QoR. The remainder of this paper

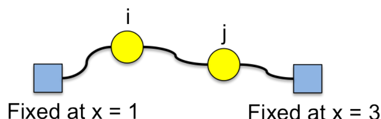

Fig. 1: Toy circuit to illustrate AP formulation.

is organized as follows: Section 2 reviews background and related work. Our AP placement algorithm is described in Section 3. Section 4 presents an experimental study. Conclusions and suggestions for future work are offered in Section 5.

2. RELATED WORK

2.1. FPGA Placement

Modern heterogeneous FPGAs have complex logic blocks, block RAMs, multiplier/DSP blocks, and a variety of I/O blocks. The placement stage takes a netlist of cells of different types and assigns each of them to a physical location on the FPGA, while minimizing an objective function (e.g. wirelength, timing, or routability) and maintaining legality. While, at its simplest, legality checking ensures that no two blocks occupy the same location, it is complicated if groups of blocks must retain fixed relative placements. On Cyclone II, groups of logic blocks that form a carry chain must be placed in adjacent locations in the same column.

2.2. Analytical Placement

The most common objective function for placement is the sum of half-perimeter wirelengths (HPWL) over all nets. Efficient AP techniques approximate this objective function with a function that can be minimized efficiently using a standard solver. First, all multi-pin nets are converted into a set of 2-pin connections. For example, in the *clique* net model, a 2-pin connection is created between *every* block on the net. Most prior AP approaches then minimize the weighted sum of the squared lengths of these 2-pin connections:

$$\Phi(\vec{x}, \vec{y}) = \sum_{i,j} w_{i,j}[(x_i - x_j)^2 + (y_i - y_j)^2] \qquad (1)$$

where $w_{i,j}$ is the weight of the connection between blocks i and j. The objective function can be separated into x and y components and cast in matrix form. For the x dimension, a matrix, Q_x, represents connections between movable objects (i.e. objects being placed), and a vector, $\vec{c}_x$, represents connections between movable and fixed objects:

$$\Phi(\vec{x}) = \frac{1}{2}\vec{x}^T Q_x \vec{x} + \vec{c}_x^T \vec{x} + const. \qquad (2)$$

Minimizing (2), which is a degree-2 polynomial, involves taking the partial derivative with respect to each variable and setting the resulting system of linear equations to zero. The problem, $Q_x\vec{x} = -\vec{c}_x$, can be handled by a standard off-the-shelf linear equation solver. Once solved, the $\vec{x}$ and $\vec{y}$ vectors hold x and y locations for the movable placement objects. To illustrate the AP formulation, consider the example in Fig. 1, with two fixed blocks (squares) and two moveable blocks (circles) i and j. In the x dimension, assuming unit connection weights, the objective function is:

$$\Phi_x = (x_i - 1)^2 + (x_i - x_j)^2 + (x_j - 3)^2 \qquad (3)$$

which can be minimized by taking:

$$\frac{\delta\Phi_x}{\delta x_i} = 2(x_i - 1) + 2(x_i - x_j) = 0 \qquad (4)$$

and

$$\frac{\delta\Phi_x}{\delta x_j} = -2(x_i - x_j) + 2(x_j - 3) = 0 \qquad (5)$$

where the linear system defined by (4) and (5) can be divided by 2 and expressed in matrix form as:

$$\begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix} \cdot \begin{bmatrix} x_i \\ x_j \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix} \qquad (6)$$

which is a linear system in the form $Q_x\vec{x} = -\vec{c}_x$.

Observe that (1) does not take placement constraints into consideration and consequently, the generated solution is not a legal placement – rather, it generally has many blocks overlapping with one another. A number of different approaches have been proposed that use the results of minimizing (2) to produce a legal placement solution – accomplished through iterated minimization and modification of (2). A popular approach to overlap removal (spreading) in recent years has been FastPlace [9], however, the state-of-the-art approach in the ASIC domain is called SimPL [7].

SimPL [7] uses a *Bound2bound* net model, first proposed in [10]. Each multi-pin net is again modeled as a set of 2-pin connections, however, unlike the clique model, it does not generate all 2-pin connections between a net's blocks. Rather, in the Bound2bound net model, the blocks with the minimum and maximum locations on a net (so-called *bound blocks*) are connected to each other and to each internal block on the net. In other words, for a p-terminal net, each internal block has two connections, one to each bound block, and each bound block has $p - 1$ connections, one to every block other than itself[1]. The connection weights are set such that the objective function targets HPWL, rather than quadratic wirelength. For example, a connection between block i and j is weighted by $\frac{1}{(p-1)\cdot|x_i - x_j|}$ (the $|x_i - x_j|$ term in the denominator "linearizes" the quadratic term $(x_i - x_j)^2$ in (1)). The Bound2bound net model leads to higher quality solutions because it directly models HPWL.

[1]The bound blocks in the x dimension may not be the same as the bound blocks in the y dimension. Hence, the internal blocks may be different in each dimension.

Observe that in the Bound2bound model, finding the bound blocks for a net and computing the connection weights requires having an already-computed placement solution as input (i.e. knowledge of the x_i's). To deal with this, an iterated solving process is used, whereby the system is formulated and solved, and the results (x_i's) are fed into the next iteration.

After solving to minimize HPWL, SimPL *legalizes* the placement by spreading the blocks across the die in a manner consistent with the minimum-HPWL solution (described in more detail in the next section). Then, an artificial pseudo connection is created between each block and its target location in the legalized overlap-free placement. When the mathematical system is again formulated and solved, the pseudo connections pull blocks towards their target locations, which tends to reduce overlaps in the placement. The process of formulating the system, solving, and legalizing continues until overlaps have been sufficiently removed. We adapt SimPL to target heterogeneous FPGAs, by extending the concept of a target location in SimPL to blocks of different types.

The few previous published works that use analytical techniques for FPGA placement have targeted a homogeneous fabric. In [6], the authors present an improvement to FastPlace and target FPGAs. They show that compared to VPR, which is based on SA, AP has the potential to offer significant run-time improvements. In this work, we address FPGA-specific challenges in analytical placement and develop a fast approach to generate high-quality placements for real FPGAs.

2.3. Reducing Placer Run-time

Placement can be accelerated through algorithmic changes, or through parallelization. Parallelization of SA-based placement has been previously addressed in a commercial flow [1], resulting in $2.2\times$ speedup using 4 cores with no QoR penalty, and in an academic flow [11], achieving ~$3\times$ speedup with 4 cores at the cost of 5-10% worse HPWL and critical path delay. Parallelization of AP-based flows has been explored in the ASIC domain in [7], where a ~$2\times$ speedup was reported using 4 cores, with no QoR penalty. Algorithmic approaches to reducing run-time have ranged from modifying SA to make directed moves, as in [12], which can improve run-time of SA-based placement by ~$5\times$ while retaining high QoR, to a hierarchical SA approach [13], which offers ~$3\times$ run-time improvement with no QoR penalty. A recent work [14] presented a placer that locates large macro blocks, which when combined with a fast router, offers $10-50\times$ speedup to the entire CAD flow at the cost of $2-4\times$ higher critical path delay. While the speedup is commendable, decreasing circuit performance by $2-4\times$ is likely unacceptable for most FPGA users.

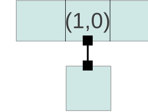

Fig. 2: Cell connecting to macro with with an offset.

3. ANALYTICAL PLACEMENT FOR FPGAS

In this section, we describe how we have adapted SimPL for heterogeneous FPGAs.

3.1. Formulating the System of Equations

Connections between blocks are based on the Bound2bound net model described in Section 2.2. Macros, consisting of multiple unit-sized blocks having a fixed placement with respect to one another (e.g. carry chains), are given a single x, y location in the formulation. In order to properly model connections to macro blocks, HeAP takes into account the position of the unit-sized block within the macro to which the connection is actually made. Fig. 2 shows a connection between a unit-sized block and a macro block. The connection is actually made to the middle block within the macro, which has a x offset of 1, relative to the macro origin (its left block). Such offsets can be modeled by incorporating them into $\vec{c_x}$ in (2) (i.e. the vector representing the connections between moveable and fixed blocks), allowing the actual connection's length to be minimized, rather than a connection to the origin of the macro.

3.2. Legalization

Our legalization phase is similar to the legalization phase in SimPL, though it is complicated by having to place blocks in discrete FPGA locations. Essentially, a recursive partitioning-style placement approach is used to transform the solution produced by the solver into a legal overlap-free placement. The first step is to find an area of the FPGA that is overutilized (i.e. illegal) for which the blocks contained within must be spread to a larger area. To obtain this overutilized area, adjacent locations on the FPGA that are occupied by more than one block are repeatedly clustered together, until all clusters are bordered on all sides by non-overutilized locations. Next, the area is expanded in both the x and y dimensions until it is large enough to accommodate all blocks contained within. Specifically, the area is expanded until its occupancy (O_A) divided by its capacity (C_A) is less than a maximum utilization factor (β) so that $\frac{O_A}{C_A} < \beta$, where $\beta <= 1$ (we use $\beta = 0.9$ in our experiments).

In the second step of legalization, two cuts are generated: a source cut and a target cut. The source cut pertains to the blocks being placed; the target cut pertains to the area into which the blocks are placed. The source cut splits the blocks into two partitions, while the target cut splits the area into two sub-areas, into which the blocks in each partition

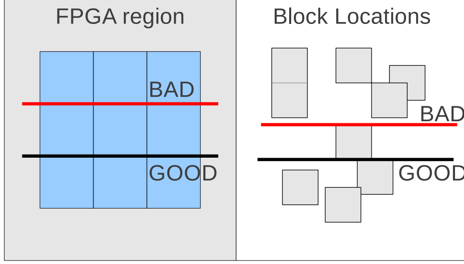

Fig. 3: Discrete cut generation problem.

are spread. Two objectives are minimized during this process: the imbalance between the number of blocks in each partition, and the difference in the utilization of each sub-area. The latter is defined as the occupancy divided by the capacity of the sub-area ($U_{sub-area} = \frac{O_{sub-area}}{C_{sub-area}}$). To generate the source cut, the cells are first sorted by their x or y location, depending on the orientation of the desired cut. Once the cells are sorted, source cut generation is akin to choosing a pivot in a sorted list, where all blocks to the left of the pivot are assigned to the left/bottom partition and all blocks to the right of the pivot are assigned to the right/top partition.

The target cut is an x or y cut of the area such that all blocks in each partition fit in their respective sub-areas, and such that $|U_{sub-area_1} - U_{sub-area_2}|$ is minimized. The discrete nature of FPGA placement locations makes this difficult. For example, in Fig. 3, a region of the FPGA is drawn on the left, while blocks to be placed in this region are shown on the right. The source cut labeled "BAD" on the right splits the blocks between the partitions as evenly as possible. Unfortunately, there is no way to make a corresponding target cut so that $U_{sub-area} <= 1$. It is also possible to make bad cuts if choosing the target cut first, as shown by the cut labeled "BAD" on the left, which does not leave enough room in the top partition for the tall block on the right. We mitigate this issue by allowing the target cut generation phase to perturb the source cut by moving blocks between partitions. In Fig. 3, the source cut labeled "BAD" on the right would be chosen first because it minimizes imbalance between partitions. Next, the target cut labeled "GOOD" on the left would be chosen because it minimizes $|U_{sub-area_1} - U_{sub-area_2}|(|4/3-5/6| = 3/6)$. Finally, the source cut would be perturbed by moving a block from the overutilized partition, which is closest to the cut line, to the underutilized partition, leading to a legal source/target cut combination.

Next, the cells in sub-areas are spread to distribute them evenly. We use the spreading method proposed in SimPL, which splits the sub-area into 10 equal-sized target bins, and splits the cells into 10 equal-capacity source bins. Linear interpolation is used to map cells from their original locations in their source bins to new spread locations in their target bins.

The cut generation and spreading process is recursively repeated for each of the left and right sub-areas, alternating x and y cut directions, until a single block remains in each sub-area. If, at any point, a legal cut is not found, the recursion jumps up one level and switches to a largest-first greedy packing algorithm: we place blocks in non-increasing order into positions as close as possible to those suggested by the solver results. Further details on the packing are omitted for space reasons, however, we found that our packer was invoked in relatively few cases. In the remainder of this section, the *solved placement* refers to the placement solution produced by the linear system solver (which contains overlaps); the *legalized placement* refers to the overlap-free placement solution generated by transforming the solved placement using the procedure described above. We use the term *iteration* to mean one pass of solving the linear system and then legalizing the placement. Once legalized locations are found for all movable blocks, pseudo connections are added to the AP formulation between each movable block and its legalized location. As with SimPL, the weight of a pseudo connection is $\alpha \times i$, where i is the iteration count and α is a scaling factor. We explore the impact of this scaling factor further in Section 3.4.2.

3.3. Implementation

3.3.1. Stopping criteria

The solving/legalization process continues until one of two conditions are met: the legalized placement's HPWL has stalled for $stall_{max}$ iterations or the quality of the solved solution is close enough to the quality of the legalized solution such that further improvements are unlikely. In other words, the solving/legalization process continues until $s > stall_{max}$, where s is the number of stalled iterations, or $HPWL(solved) > converge_T \times HPWL(legalized)$. We have experimentally determined that values of $stall_{max} = 15$ and $converge_T = 0.7$ work well, though higher values of either give HeAP more time to find a good solution at the cost of increased run-time. After each iteration, the legalized (spread) solution is saved if it is the best one encountered thus far.

3.3.2. Iterative Refinement

We invoke a greedy iterative refinement pass after AP terminates. It works as follows: a random block is chosen as swap source, which can be of any block type (including macros). Next, a random swap location is chosen within a fixed sized window, which we set as 3 blocks in any direction of the swap source. The swap is greedily accepted if it improves the sum of HPWL of the affected nets. Macros can be swapped with unit-sized blocks, or can be relocated to vacant locations, however, we do not permit macros to swap locations with other macros owing to the difficulty in handling partial overlaps.

3.3.3. *Solving*

Because Q is positive symmetric definite, we use a fast iterative conjugate gradient solver to solve the systems of linear equations (TAUCS v2.2[15] with GotoBLAS2 [16]).

3.4. Algorithm Tuning

3.4.1. *Handling Heterogeneity*

There is a complex interaction between the legalization step and the solving step which arises due to heterogeneity. HeAP legalizes (spreads) each type of block separately, which means that blocks of different types may be spread to opposite sides of the FPGA, even if they have solved x and y locations in close proximity – this occurs as some types of blocks can only be accommodated in a limited number of locations on the FPGA. It can therefore be useful to solve for the placement of only one block type in isolation then legalize/spread just that type of block. This allows the generation of an optimized placement for blocks of a given type in the context of already-placed blocks of other types. HeAP supports solving for the placement of various block types separately by treating blocks that are not of the type being solved as non-movable (i.e. fixed) objects. For example, it is possible to solve/legalize solely the multiplier blocks in a design, while considering LUT-based logic blocks and block RAMs as fixed.

The choice of when to solve and spread blocks of different types can impact HPWL. For instance, treating all blocks as movable at every iteration, thereby solving for all block types, then spreading each type separately may lead the legalized placement to have lengthened connections between blocks of different types (owing to the mathematical formulation being unaware of the discrete set of locations available for each type of block). Conversely, solving for each block type separately then spreading only that type will reduce this effect, but the lack of a global solving step (with all blocks being movable) may be detrimental to the solved solution quality. To investigate these issues, we tested three different solve/legalize orders on our largest benchmark, `jpeg`, which has a large number of all types of blocks – LUT-based logic blocks, block RAMs and multipliers.

Fig. 4 shows the HPWL of the solved and legalized placement solutions vs. time for three solve/legalize orders. Each order has a corresponding line in the figure for both the solved and legalized placement solutions. The HPWL of the solved solutions gets worse as the algorithm progresses because the solutions are closer to legal; conversely, the HPWL of the legalized solutions gets better as the algorithm progresses because the solved placement is not perturbed as much to get a legal solution. Hence, they can be seen to converge. For the first solve order, labeled “all”, all blocks are solved/legalized at every iteration. For the second solve order, labeled “rotate”, we solve/legalize block types separately in a round-robin style. The third approach, labeled

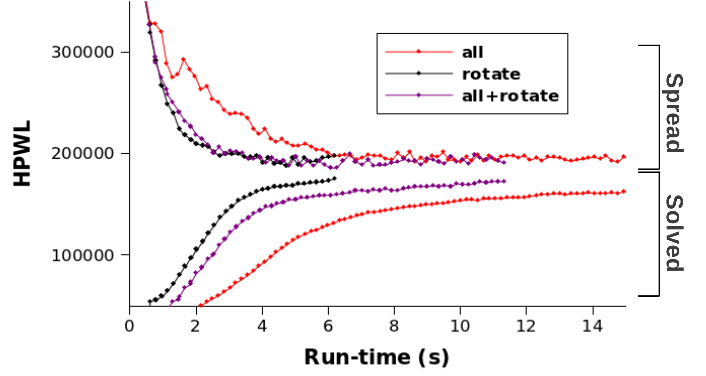

Fig. 4: Convergence rates for different solving orders for the jpeg benchmark.

“all + rotate”, is a hybrid, where we add solving/legalizing for all blocks to the round-robin approach. The results show that solving/legalizing for all blocks at every iteration leads to long run-times and poor QoR. The two other solve orders have comparable QoR, though upon further testing with all benchmarks, the “all+rotate” flow delivers more consistent results, likely because it offers a global view of the problem. Note also that solving for block types in isolation can be very quick; for example, there are relatively few multiplier blocks in comparison with LUT-based logic blocks and hence, when solving for multiplier blocks, there are few unknowns in the linear system. For the remainder of this work, we use the “all + rotate” flow.

3.4.2. *Convergence Rate*

At every iteration, $\alpha \times i$ increases, which leads the HPWL of the solved and legalized solutions to converge. Slow convergence will ensure that a bad spreading will not have a large negative effect on the overall quality of the solved solution. Conversely, fast convergence will improve run-time, which is the main focus of this work. We swept the parameter α (which is the amount by which the iteration count i is multiplied to compute pseudo connection weights), from 0.1 to 0.5 in increments of 0.1, and ran HeAP on the largest benchmark, `jpeg`. Fig. 5 shows the HPWL of the solved and legalized solutions vs. time for each value of α. Each value of α has a corresponding line in the figure for the solved and legalized placement solutions. Surprisingly, using a value of $\alpha = 0.1$ leads to worse QoR than a using a value of $\alpha = 0.2$, which can be seen in the figure, where the line corresponding to $\alpha = 0.1$ in the spread category is higher than the line corresponding to $\alpha = 0.2$ in the spread category. Unsurprisingly, using a very high value of α ($\alpha = 0.5$) leads to fast convergence, but gives the spreading too much influence over the solving, which hurts QoR. The value that offers the best QoR appears to be $\alpha = 0.3$, though values between 0.2 and 0.4 are also reasonable.

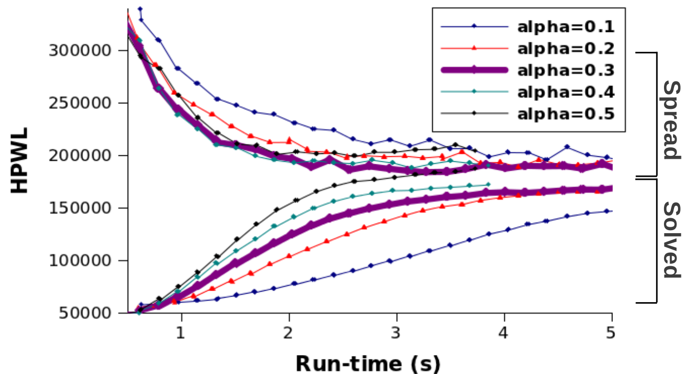

Fig. 5: Convergence rates for different values of α.

3.5. Parallelization

While parallelization was not a main focus of this work, we took advantage of the obvious parallelism present in our algorithm so that we could compare it against the multi-threaded Quartus II without forcing the latter to use only 1 core. We parallelize HeAP in two ways: by calling the solver for the x and y dimensions from different threads, and by parallelizing the "move suggestion" part of our iterative refinement phase. Solving for the x and y dimensions concurrently results in close to 2× improvement in solving time (thread overhead and cache effects prevent perfect parallelism). For iterative refinement using n processors, n swaps are generated in parallel and the best one is committed. We observe a 1.34× speedup using 2 cores and a 1.48× speedup using 4 cores in the refinement stage, for the same QoR. Overall, we were able to speedup HeAP by 1.3× via parallelization, though significant unexplored parallelism remains.

4. EXPERIMENTAL STUDY

In order to interface with Quartus II, HeAP makes use of the Quartus University Interface Program (QUIP) [8]. Quartus II has the ability to annotate files with a technology mapped netlist, a packing of logic blocks, I/O and logic block placement, and routing information. In order to perform an apples-to-apples comparison of HeAP with Altera's placer, we use the same technology-mapped netlist, logic block packing, and I/O placement in both HeAP and Altera back-end tools, then compare the post-routing QoR from HeAP's placement vs. Altera's placement.

For each benchmark circuit, we first pass it through the complete Altera flow and record the resultant QoR. Next, we extract the netlist, packing and I/O placement, and pass this information to HeAP. HeAP then places the design and saves its placement results, which is passed into Quartus II for routing, after which the resultant QoR is recorded. Fig. 6 illustrates the tool flow that we used to test our placer. All results were generated using Quartus II 11.0. As discussed earlier, the following parameters were used for HeAP:

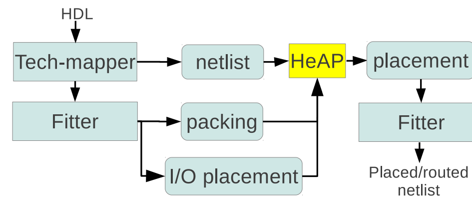

Fig. 6: Tool flow.

Table 1: Circuit characteristics.

benchmark	LUTs	FFs	M4Ks	DSPs	LABs	Carry chains	device
adpcm	15233	9856	14	26	1122	98	C35
aes	15777	9240	21	0	1169	136	C35
bf	8806	6104	0	0	693	49	C20
dfdiv	10142	9359	3	19	1010	74	C20
dfsin	22178	15054	3	35	1925	112	C35
gsm	10525	5541	0	10	897	64	C20
jpeg	37655	17034	32	27	2751	19	C50
oc_des	11695	5906	4	0	1552	0	C35
oc_video	10698	3670	21	0	856	189	C20
sha	11366	7431	16	2	1095	112	C20

$\beta = 0.9$, $converge_T = 0.7$, $stall_{max} = 15$, and $\alpha = 0.3$.

We compare HeAP against two flows. The first flow is our own implementation of the SA-based placement found in VPR, using a bounding-box objective function and the same SA schedule found in the VPR 5.0 source code[2]. We compare against this flow using HPWL and placement run-time metrics. The second tool we compare against is the Quartus II fitter, using post-routed wirelength, maximum circuit frequency (Fmax) and run-time. All experiments were run on an Intel Core i5-750 based system with 4 cores. We ran our tool in single-threaded mode when comparing against our own SA implementation, since the latter is not parallelized, and ran our tool in multi-threaded mode when comparing against Quartus II, which uses all 4 cores. Our tool supports the Cyclone II FPGA family; Cyclone II devices contain multiplier/DSP blocks (DSPs), block RAMs (M4Ks), and logic clusters (LABs). The benchmark circuits are the largest available in the QUIP benchmark suite, and the largest from the CHStone [17] suite. The CHStone circuits were synthesized to hardware using LegUp [18] – an open-source high-level synthesis tool. Circuit details along with the device to which each circuit was targeted are shown in Table 1. Each benchmark circuit is targeted to the smallest Cyclone II FPGA device able to accommodate it.

4.1. Results Versus VPR-like SA

Figs. 7 and 8 show the circuit-by-circuit HPWL and run-time results for both HeAP and our SA implementation. HeAP offers a geomean HPWL across all circuits that is 6% better than SA. Note that we had to make modifications to SA to enable swapping macros. Since multiple blocks must be swapped with each macro, swapping that macro has a large impact on the overall placement – a large placement pertur-

[2] We implemented the VPR "fast" annealing schedule, and also implemented incremental bounding box updates in our SA-based placer.

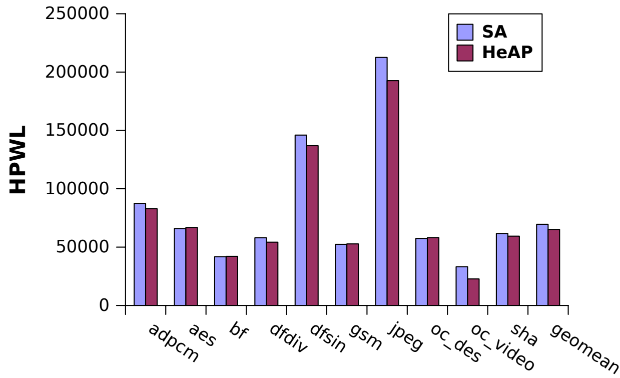

Fig. 7: HPWL of HeAP vs. SA-based placement.

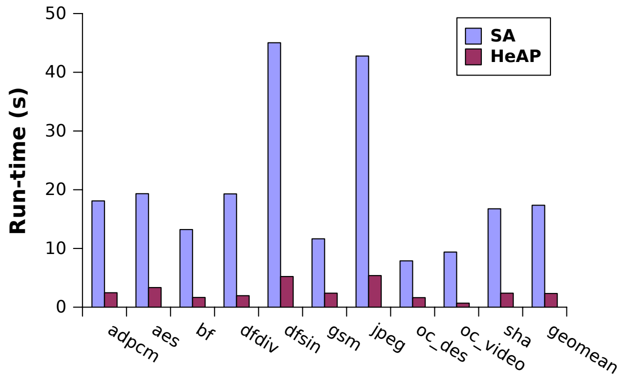

Fig. 8: Placement run-time of HeAP vs. SA-based placement.

bation. SA may perform better when placement perturbations are very small. The run-time results show that HeAP offers a geomean speedup of 7.4×.

4.2. Results Versus Quartus II

In order to thoroughly explore the trade-off between QoR and run-time, we compare HeAP with Quartus II run in both the timing-driven and non-timing-driven modes, with several fitter and placer effort levels for each of these modes. We included a very high effort Quartus run, a very low effort Quartus run, and some in between. Specifically, we ran fitter effort level "FAST" with placer effort levels 0.1, 0.5, 1.0, and 2.0, and fitter effort level "STANDARD" with placer effort level 2.0. The "FAST" fitter effort with a placer effort of 0.1 is the low effort Quartus run, while the "STANDARD" fitter effort with a placer effort of 2.0 is the high effort Quartus run. Together, the results from these runs form a Quartus II QoR/run-time landscape. Figs. 9 and 10 show the QoR/run-time trade-offs offered by both HeAP and by Quartus II, indicating the geomean post-routed wirelength (reported by Quartus II) and geomean maximum frequency, respectively, of circuits vs. the geomean *combined* placement and routing run-times. The data points labeled "Quartus II - TD" and "Quartus II - NTD" represent the geomean of all circuits run through placement and routing in Quartus II in timing-driven or non-timing driven mode, respectively, with

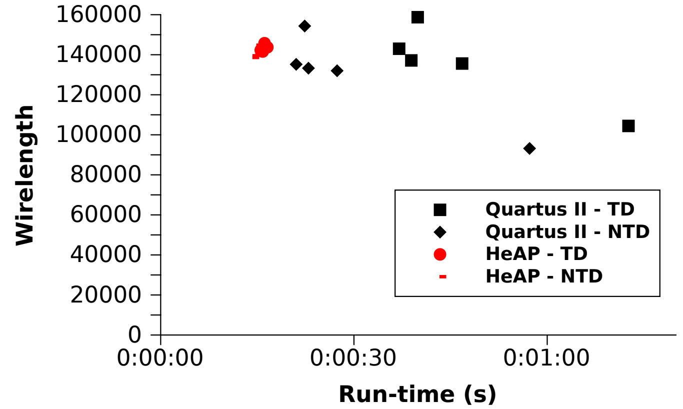

Fig. 9: Geomean wirelength vs. P&R run-time varying Quartus II effort levels.

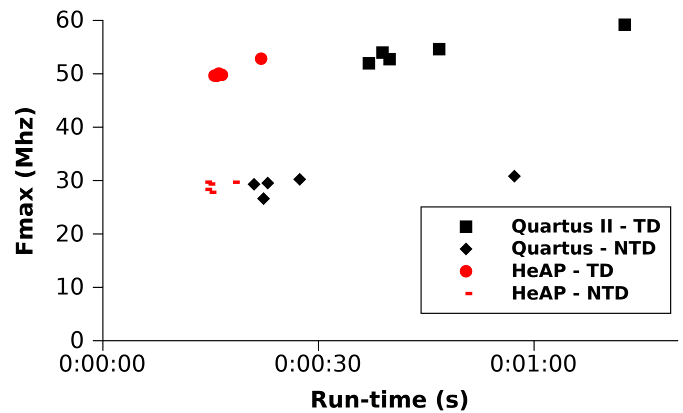

Fig. 10: Geomean Fmax vs. P&R run-time varying Quartus II effort levels.

a range of different effort levels. The data points labeled "HeAP - TD" and "HeAP - NTD" were generated by measuring the run-time of placing the designs with HeAP and routing those HeAP-placed designs with Quartus II, again with a range of effort levels (i.e. effort levels in Quartus II).

Comparing the squares (timing-driven Quartus II) to the circles (timing-driven HeAP) in Fig. 9, we observe qualitatively that the geomean wirelengths provided by HeAP are not far different than those produced by Quartus II – the two sets of datapoints are vertically aligned, with the exception of the high effort Quartus II run, for which considerably lower wirelength was achieved (see the rightmost square in Fig. 9). However, all of the HeAP datapoints (circles) lie to the left of the Quartus II datapoints (squares), indicating a significant run-time advantage for HeAP. The same trends are seen for the non-timing-driven runs (compare the diamond to the dash datapoints). Fig. 10 gives the analogous results for circuit speed, and exhibits the same trends as those seen with the wirelength results – HeAP provides reasonably good Fmax results at much less run-time cost.

Tables 2 and 3 show detailed circuit-by-circuit results comparing HeAP with Quartus II's non-timing-driven and

Table 2: Results for the non-timing driven flow.

	Quartus II			HeAP			Ratios		
	WL	MHz	RT	WL	MHz	RT	WL	MHz	RT
adpcm	145	24.8	21.0	180	23.3	1.8	0.80	1.06	11.7x
aes	133	24.6	8.0	138	22.9	2.5	0.96	1.07	3.2x
bf	96	34.7	3.0	99	36.2	1.2	0.96	0.96	2.5x
dfdiv	109	36.1	7.0	121	37.3	1.6	0.90	0.97	4.3x
dfsin	242	11.9	18.0	271	11.8	3.6	0.89	1.01	5.0x
gsm	110	14.5	4.0	110	15.9	1.9	1.00	0.92	2.1x
jpeg	321	9.1	25.0	366	8.4	3.4	0.88	1.08	7.4x
oc_des	111	110.0	4.0	110	114.6	1.4	1.01	0.96	2.9x
oc_video	85	71.3	3.0	75	73.2	0.7	1.14	0.97	4.4x
sha	110	53.4	6.0	114	49.7	2.0	0.96	1.08	3.1x
geomean	133	29.5	18.1	141	29.4	1.8	0.95	1.01	4.1x

Table 3: Results for the timing-driven flow.

	Quartus II			HeAP			Ratios		
	WL	MHz	RT	WL	MHz	RT	WL	MHz	RT
adpcm	154	41.1	60	183	36.9	1.8	0.85	1.11	33.6x
aes	132	57.7	19	136	50.6	2.2	0.96	1.14	8.6x
bf	97	68.0	7	98	63.1	1.1	0.99	1.08	6.3x
dfdiv	116	67.6	19	120	58.3	1.9	0.96	1.16	9.9x
dfsin	240	28.0	40	273	25.0	3.6	0.88	1.12	11.2x
gsm	116	28.4	11	112	27.0	1.3	1.04	1.05	8.2x
jpeg	324	20.0	55	351	18.4	3.5	0.92	1.09	15.8x
oc_des	113	152.1	11	112	147.7	1.3	1.01	1.03	8.7x
oc_video	88	90.6	10	82	90.2	0.7	1.07	1.00	13.8x
sha	116	87.8	13	118	80.4	1.9	0.98	1.09	7.0x
geomean	137	54.0	1.4	142	49.7	1.7	0.96	1.09	10.8x

timing-driven flows, respectively. These results were generated using the "FAST" fitter effort with the default placer effort of 1.0. In these tables, the "WL" columns report post-routed wirelength (divided by 1000 for easier readability); the "MHz" columns give the circuit speed in MHz; and, the "RT" columns report *placement* run-time in seconds. In both tables, the last three columns give the ratios of Quartus II to HeAP for wirelength and Fmax, and the speedup achieved by HeAP over Quartus II. The last row of the tables gives the geometric mean across all circuits. From Table 2, we see that HeAP offers a 4.1$\times$ speedup vs. Quartus II's non-timing-driven flow, at a 5% cost to wirelength and roughly flat Fmax. For timing-driven (Table 3), HeAP provides a 10.8$\times$ speedup, at the cost of 4% wirelength and 9% worse critical path delay.

It is interesting to note that when run with very high effort, The Quartus II placer can produce very high quality results - significantly better than those produced by HeAP or Quartus II run with low/medium effort levels. Specifically, the right-most diamond in Fig. 9 shows a 26% lower wirelength vs. the equivalent HeAP run, albeit at the cost of an 18$\times$ placer slowdown. This indicates that an SA-based approach may be capable of delivering higher-quality results than an AP-based approach. However, Tte low effort Quartus II run (the left-most diamond in Fig. 9) produces wirelength that is 7% worse than the equivalent HeAP run and has 2$\times$ higher placer run-time, which indicates that AP may be a better choice than SA when tool run-time is important. Overall, we find the results for HeAP to be quite encouraging, considering that Quartus II is a highly-optimized commercial tool. We believe that HeAP will be of value in raising engineering productivity by shortening the design cycle, at a modest cost to layout quality.

5. CONCLUSIONS

The scalability of FPGA CAD tools is a crucial issue for engineering design productivity and for making FPGAs viable as computing devices, targetable by software engineers who expect software-like compile times. In this paper, we presented HeAP, a run-time-optimized AP-based placer that supports FPGAs with heterogeneous block types. On the largest benchmark considered, `jpeg`, HeAP generates a routable placement in less than 4 seconds, with 8% higher wirelength and 9% worse Fmax than Altera's Quartus II placer in timing-driven mode, which uses 55 seconds. On average, HeAP offers a 4.1-10.8$\times$ run-time advantage, at 4-5% wirelength cost and a 1-9% speed performance reduction. HeAP's value proposition lies in reducing the time needed for design iterations, and in applications for which the highest-possible circuit speed is not required. Future work involves further parallelization of our approach, improving the iterative refinement phase (for example, using directed moves [12]), as well as enhancing HeAP to be fully timing-driven.

6. REFERENCES

[1] A. Ludwin and V. Betz, "Efficient and deterministic parallel placement for FPGAs," *ACM TODAES*, vol. 16, no. 3, 2011.

[2] S. Gupta, J. Anderson, L. Farragher, and Q. Wang, "CAD techniques for power optimization in Virtex-5 FPGAs," in *IEEE CICC*, 2007, pp. 85–88.

[3] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *FPL*, 1997, pp. 213–222.

[4] *Cyclone-II FPGA family datasheet*, Altera, Corp., San Jose, CA, USA, 2012.

[5] *Virtex-5 FPGA Data Sheet*, Xilinx, Inc., San Jose, CA, USA, 2012.

[6] H. Bian, A. Ling, A. Choong, and J. Zhu, "Towards scalable placement for FPGAs," in *ACM/SIGA FPGA*, 2010, pp. 147–156.

[7] M.-C. Kim, D. Lee, and I. Markov, "SimPL: An effective placement algorithm," *IEEE TCAD*, vol. 31, no. 1, pp. 50–60, 2012.

[8] "Quartus-II university interface program," *http://www.altera.com/education/univ/research/unv-quip.html*, 2009.

[9] N. Viswanathan and C. C.-N. Chu, "FastPlace: Efficient analytical placement using cell shifting, iterative local refinement and a hybrid net model," in *ACM/IEEE ISPD*, 2004, pp. 26–33.

[10] P. Spindler, U. Schlichtman, and F. Johannes, "Kraftwerk2 - a fast force-directed quadratic placement approach using an accurate net model," *IEEE TCAD*, vol. 27, no. 8, pp. 1398–1411, 2008.

[11] C. C. Wang and G. G. Lemieux, "Scalable and deterministic timing-driven parallel placement for FPGAs," in *ACM/SIGDA FPGA*, 2011, pp. 153–162.

[12] K. Vorwerk, M. Raman, J. Dunoyer, Y.-C. Hsu, A. Kundu, and A. Kennings, "A technique for minimizing power during FPGA placement," in *IEEE FPL*, 2008, pp. 233–238.

[13] Y. Sankar and J. Rose, "Trading quality for compile time: ultra-fast placement for FPGAs," in *ACM/SIGDA FPGA*, 1999, pp. 157–166.

[14] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings, "HMFlow: Accelerating FPGA compilation with hard macros for rapid prototyping," in *IEEE FCCM*, 2011, pp. 117–124.

[15] "TAUCS, a library of sparse linear solvers," *http://www.tau.ac.il/ stoledo/taucs*, 2003.

[16] "Gotoblas2," *http://www.tacc.utexas.edu/tacc-projects/gotoblas2*, 2010.

[17] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis," *Journal of Information Processing*, vol. 17, no. 0, pp. 242 – 254, 2009.

[18] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. Anderson, S. Brown, and T. Czajkowski, "LegUp: high-level synthesis for FPGA-based processor/accelerator systems," in *ACM/SIGDA FPGA*, 2011, pp. 33–36.