Design Re-Use for Compile Time Reduction in FPGA High-Level Synthesis Flows

Marcel Gort, Jason Anderson
ECE Department, University of Toronto, Toronto, ON, Canada
[gortmarc,janders]@eecg.toronto.edu

***Abstract*—High-level synthesis (HLS) raises the level of abstraction for hardware design through the use of software methodologies. An impediment to productivity in HLS flows, however, is the run-time of the back-end toolflow – synthesis, packing, placement and routing – which can take hours or days for the largest designs. We propose a new back-end flow for HLS that makes use of pre-synthesized and placed "macros" for portions of the design, thereby reducing the amount of work to be done by the back-end tools, lowering run-time. A key aspect of our work is an analytical placement algorithm capable of handling large macros whose internal blocks have fixed relative placements, in conjunction with placing the surrounding individual logic blocks. In an experimental study, we consider the impact on run-time and quality-of-results of using macros: 1) in synthesis alone, and 2) in synthesis, packing and placement. Results show that the proposed approach reduces run-time by $\sim$3$\times$, on average, with a negative performance impact of $\sim$5%.**

I. Introduction

Field-programmable gate arrays (FPGAs) are increasingly being used to implement custom accelerators to raise computational throughput and energy efficiency. For certain applications, the performance advantages (in power and/or speed) of using FPGAs to realize accelerators can be significant and exceed GPUs and multicore processors [4], [5], [1], [16]. There are two impediments, however, to the widespread adoption of FPGAs as computing platforms: 1) they require hardware expertise and the use of hardware description languages (HDLs), which make them difficult to use for software engineers, and 2) the run-time of FPGA tools is too long, taking hours or days for the largest designs, which is mainly due to the tools implementing a design from scratch at the individual bit/wire level. In this paper, we propose an approach to mitigate the second impediment, in the context of recent progress on the first impediment, namely, high-level synthesis (HLS).

HLS raises the level of abstraction for hardware design by allowing software methodologies to be used, thereby improving the "design difficulty" challenge. HLS technologies are of interest to two types of designers: 1) hardware engineers who use HLS to improve productivity, and 2) software engineers who possess no hardware skills yet wish to gain some of the performance benefits of hardware. Software engineers outnumber hardware engineers by 10$\times$ [15] and consequently, the potential user base of FPGAs would be greatly expanded if HLS flows were to gain widespread acceptance in the FPGA community. Software engineers, however, are accustomed to compile times of seconds or minutes.

In this paper, we propose to reduce tool run-time through the use of pre-synthesized and placed "chunks" of the design, which we refer to as *macros*. Our approach is particularly suited to HLS, where few if any HW details are specified in the source code and the HLS tool is free to choose the HW implementation. In essence, with our methodology, the work required by back-end synthesis, placement and routing is reduced considerably, as a large fraction of work is replaced with "instantiation and stitch" of macros from a library. This is analogous to the use of pre-compiled libraries in software development: a user does not recompile the `stdio` library for every program; rather, they simply link it into their binary.

We explore macros with various granularities, ranging from the size of basic blocks in the program's control-dataflow graph to entire `C` functions. Macros limit the ability of back-end CAD tools to optimize the design globally, the effect of which we evaluate on both synthesis and placement. With the use of macros, new placement challenges arise, specifically, the need to handle large macros with a fixed placement footprint alongside individual logic blocks – the so-called "macros and dust" placement problem [13]. We evaluate a number of solutions to this problem using a recently developed analytical placer [7]. In an experimental study of benchmarks with multiple macros, we assess the impact of using macros on quality-of-result (QoR) and run-time. Results show that, on average, the run-time of the entire back-end toolflow can be reduced by almost 3$\times$, at a cost of 5% worse performance (due to the restricted ability of the back-end tools to optimize the design across macro boundaries). The proposed flow should interest users who do not need highest-achievable performance, but who are seeking fast turnaround time for design iterations.

II. Related Work

Recent work [11], [10] presents a placer that handles large macro blocks, which when combined with a fast router, offers $30-40\times$ speedup to the entire CAD flow at the cost of $2-3\times$ higher critical path delay. While the speedup is commendable, decreasing circuit performance by $2-4\times$ is likely unacceptable for most FPGA users. Our macros do not include routing information, which limits the whole-flow speed-up, though the circuit performance is only degraded by 5%, on average.

In [14], Tessier presents an approach that breaks the placement problem into several steps, each of which is a type of clustering. The input to the flow is a set of macroblocks, which are pre-placed groups of logic blocks. Each of these macroblocks implements the function of a library element in the RAW benchmark suite [2]. Input circuits are created as networks of these library elements, which become networks of macroblocks. These macroblocks are first clustered together, then placed into physical partitions of the FPGA using a shape-filling algorithm. Tessier compares the results produced by his tool with the results produced with Xilinx PAR2.1 by connecting his tool with the Xilinx router. He achieves a reduction in placement time of 17$\times$, and a total place and route run-time reduction of 2.6$\times$. Additionally, his tool improves circuit speed by 7%. While Tessier presents an approach tailored to networks of macros, without any dust (glue logic), our approach targets general purpose hardware, generated with HLS tools.

A. High-Level Synthesis

We implemented much of our work in the LegUp open-source HLS framework from the University of Toronto [3]. LegUp is implemented within the LLVM compiler framework [12]. With LegUp, a `C` program is translated into LLVM's intermediate representation (IR), comprising machine-independent assembly code. HLS is performed on the optimized IR, scheduling instructions into states, producing a finite state machine (FSM), and ultimately outputting Verilog RTL code. Much of our work is implemented in the Verilog-generation stage of LegUp, where we altered the tool to create "holes" in the design for later macro insertion.

B. HeAP Analytical Placer

HeAP is a recently published analytical placer for FPGAs [7] that is capable of targeting Altera Cyclone II devices. We adapted it for the mixed-size placement problem (to be discussed below). The basic flow of HeAP is as follows: the placement problem is formulated mathematically as a linear system of equations to be solved. Solving the system provides a real-valued placement position for each block in the design, with overaps between blocks. This is referred to as the *solved* placement. HeAP then legalizes the solved placement to produce a *spread* placement. Legalization is done using a recursive bipartitioning approach, attempting to keep blocks close to their solved locations. The spread location for every block is then used to create an "anchor" for the block (which is a pseudo connection from the block to the location). The purpose of the anchors is such that, with repeated iterations of solving and spreading, blocks will tend to get closer to their spread locations – a legal overlap-free placement. The interested reader is referred to [7] for complete details.

III. Macro Detection and Generation

We use the term *macroize* to refer to the use of a pre-synthesized/placed macro to implement a piece of `C` code, as opposed to letting HLS produce its own HW implementation. We determined the sections of code that were to be macroized in two ways: automatically, from LLVM basic blocks, and manually, from user-specified `C` functions.

A. Automatic Macro Detection

Given the vast amount of software that already exists, it is likely that, in the process of creating an application, a programmer may duplicate existing functionality, say for example, code to find the average of integers in an array. Using existing `C`-to-FPGA tool flows, however, requires the code to be re-synthesized, re-placed, and re-routed, no matter how many times similar code has already been compiled. If, however, the commonly occuring code is automatically tagged as reproducing existing functionality, a pre-compiled solution could be used instead. We envision that eventually such pre-compiled solutions could be located in the cloud, likely as a vendor service. CAD tools located either on user machines or in the cloud could search and retrieve pre-compiled solutions.

In LLVM, the averaging code referred to above would be considered a *basic block*. A basic block is a piece of code with a single entry and exit point. Typically, this piece of code ends with a branch or return instruction. LLVM maintains a control-flow graph, where vertices are basic blocks and edges are entry/exit points between those basic blocks. Computations performed by a program are located within basic blocks.

Our automatic macro detection approach considers macroizing each basic block in a given benchmark. For each basic block, we use a pre-synthesized and placed solution that we pre-generate in advance and deposit in a library. We then evaluate the potential benefits and drawbacks of that solution.

B. Explicit Macro Instantiation

While automatic macro detection and instantiation is convenient from a user point of view, this approach may be limited by the complexity of macros that can be automatically detected. If a user has written a function with a substantial amount of code, it is unlikely that the code will be sufficiently similar to an existing implementation to automatically find a functionally equivalent match. If, however, a user explicitly uses a function from a library of pre-compiled macros, the association can easily be made.

To explore the benefits of having a library of pre-compiled macros, we looked through the CHStone [8] benchmarks for functions, that when compiled to HW, resulted in at least 4 Altera Cyclone II logic array blocks (LABs). We generated pre-synthesized and pre-placed macros for these functions. When compiling each benchmark, we selectively use the macros, one-by-one, to determine the impact on QoR.

IV. Tool Flow

We employ a mixed academic/commercial tool flow that uses components from Altera's Quartus II tool (version 11.1), as well as HeAP. Using HeAP, rather than Quartus II's placer, allows for explicit control of how macros are handled within placement, which enables the use of different placement strategies, and makes possible an in-depth analysis of how placement is affected by macro choice. We target the Altera Cyclone II FPGA. The flow is summarized in Fig. 1. It is split into five main steps:

1) **Run modified LegUp**: LegUp was augmented with the ability to generate and instantiate a Verilog module for any basic block found in the LLVM intermediate representation (IR). For such a basic block, two versions of the corresponding Verilog module are created. The first contains the LegUp-generated Verilog code that implements the functionality of that basic block. The second is a stub, and contains no logic, but contains all the I/O of the basic block. The top-level code generated by our modified LegUp instantiates the stubbed version of the macro. When this code is synthesized by Quartus II, the non-stubbed macro remains untouched, since Quartus II only sees the stubbed version. To prevent Quartus II from optimizing away the stub, synthesis directives were placed on its I/Os.
2) **Add macro to library**: Using Quartus II synthesis, each basic block and each function macro was synthesized in isolation of the surrounding code. It was then placed using HeAP and added to the library of macros. Within HeAP, each macro was optimized for Half-Perimeter Wirelength

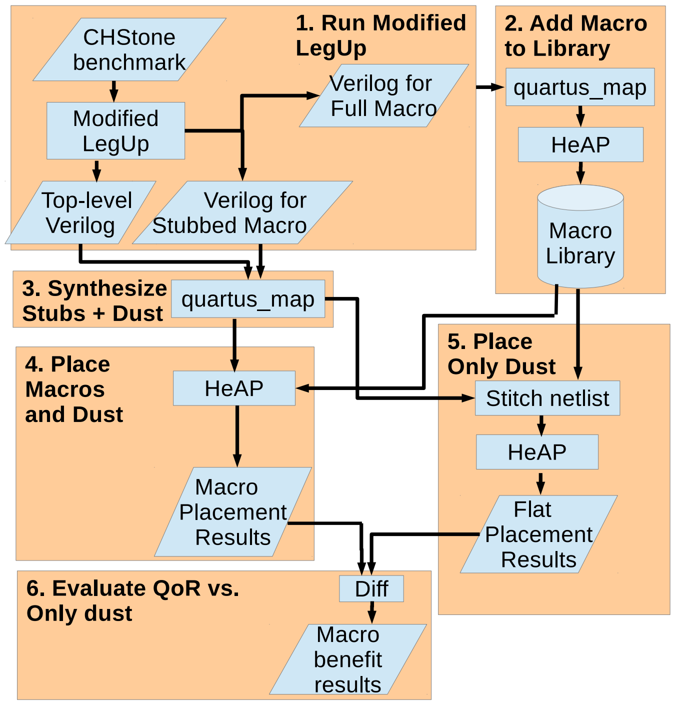

Fig. 1. Macro evaluation experimental flow.

(HPWL) at the same effort level as all other placement runs in our experiments. Nets connected to inputs and outputs of the macro were ignored (not optimized), since, at this point, it is unclear where they should be placed on the periphery. Considering macro I/O placement is an interesting area for future work.

3) **Synthesize stubs and dust**: The top level Verilog file was synthesized using Quartus II, ensuring that the stubbed versions of the macros were used. Next, a Verilog Quartus Mapping (VQM) file was generated from the resultant technology-mapped netlist, which has "holes" in it for the macros. The inputs to holes are wires with no fanout, while the outputs are driverless wires. These are later properly connected to the non-empty macro.
4) **Place macros and dust**: Given a VQM file with holes in it, the list of macros to insert into the holes, and a library of macros, our modified version of HeAP stitches together the VQM netlist and the macros to generate a complete netlist, and a macros-and-dust placement problem. The macros can be moved, but the logic blocks contained within have fixed placement, relative to the macro origin. The details of our macro-and-dust placement approach are provided in Section VI-A. In the end, a legal placement solution is generated, along with a report file that prints the name of each net in the design, the type of net (discussed below), and the HPWL of that net.
5) **Place only dust**: A baseline no-macros placement was generated by stitching the synthesized netlist (with "holes") with the synthesized macro implementations. HeAP then reads this stitched flat netlist and generates a placement.
6) **Evaluate QoR impact of using macros**: To gain a better understanding of how disabling cross-boundary optimization affects the quality-of-results, we split the nets in each benchmark into three categories: *intra-macro* nets, *inter-macro* nets, and *dust-to-dust* nets. The intra-macro nets are those with sources and sinks that are entirely contained within a macro. The inter-macro nets are those with at least one source or sink that is outside of the macro and at least one source or sink that is inside the macro. The dust-to-dust nets are those with no sources or sinks that touch the macro. Intuitively, macroizing a section of code should improve the HPWL of the intra-macro nets since the placer can focus solely on optimizing these nets. Conversely, the HPWL of macro-to-dust nets should worsen, since the placer is only optimizing for these nets after the relative placements of the logic blocks contained within the macro have been finalized. It is not obvious what the impact of macroizing is on dust-to-dust connections.

We also compare the macro flow against three different baseline flows, for a total of four different flows:

1) $Flow_{macros}$: Pre-synthesized and pre-placed solutions are used for each macro.
2) $Flow_{syn\_macros}$: Macros are used in the synthesis stage, but not in the placement stage. After synthesis finishes, HeAP stitches the macro netlists to the main netlist and placement is run on the new flat netlist, which no longer contains macros. Comparing this flow to $Flow_{macros}$ reveals the impact of using macros on placement quality of results, since the flows generate the same input to placement, with the only difference being that $Flow_{macros}$ uses pre-placed solutions to the macros, and $Flow_{syn\_macros}$ does not.
3) $Flow_{no\_macros}$: Macros are not used. By comparing this flow to $Flow_{syn\_macros}$ flow, the impact of using macros in synthesis can be isolated from the impact in placement.
4) $Flow_{Quartus}$: The standard Quartus timing-driven flow (with the Quartus placer instead of HeAP) is included for comparison.

The functional correctness of our macro flow was verified by simulating (with ModelSim) technology-mapped-level netlists generated by our flow. Final placement optimizations (e.g. repacking) were turned off in Quartus when using a placement generated by HeAP, which ensures that Quartus does not modify the placement. Additionally, a timing constraint of 1 GHz was set on all clocks.

V. Single-Macro Evaluation

To develop an understanding of how macro use impacts quality-of-result, we considered designs that contain a *single* macro. For each of the 12 CHStone benchmarks, we explore the use of each basic block and/or each function as a pre-compiled macro. Each of the 174 total macros are considered in isolation of all other macros, in essence, creating 174 different benchmark circuits, each with a single macro surrounded by dust. To measure the impact of using a macro, the associated circuit is compiled with and without the pre-compiled solution for this macro.

1) Effect on Placement QoR: The impact of macroizing a section of code on placement was isolated by comparing $Flow_{macros}$ with $Flow_{syn\_macros}$. In both versions, optimizing across macro boundaries in synthesis is disabled. The macro logic is then added back into the netlist after dust synthesis is complete.

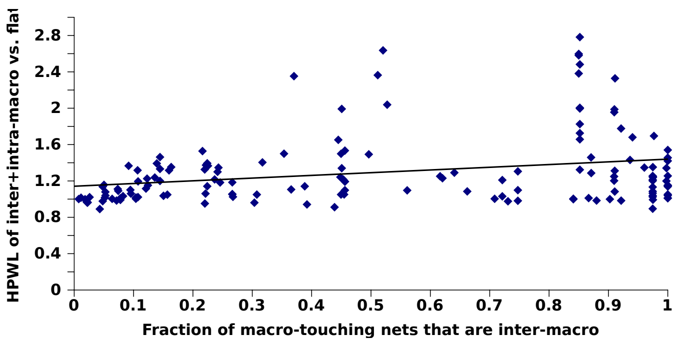

Fig. 2. Sum of the HPWL of all nets touching a macro vs. the fraction of those nets that cross the macro boundary for macros formed from both functions and basic blocks.

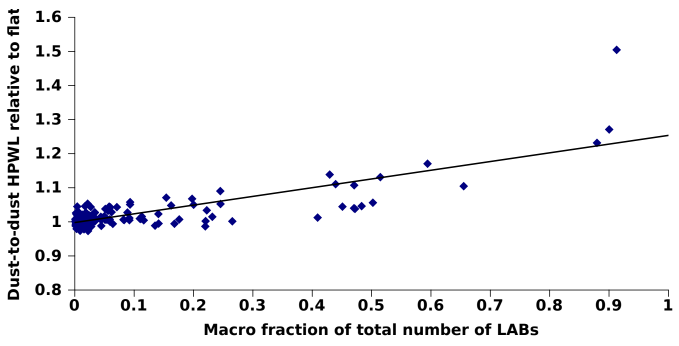

Fig. 3. Sum of the HPWL of all nets which do not touch a macro vs. the size of all macros.

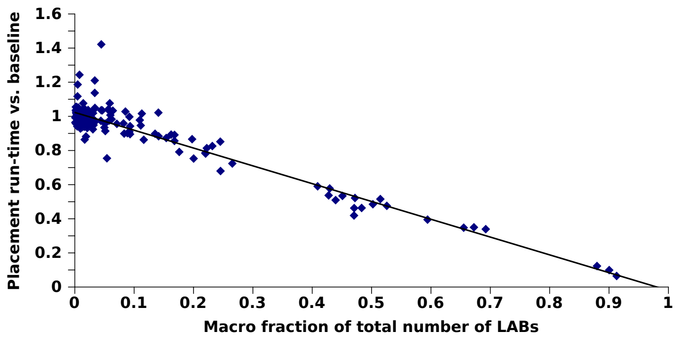

Fig. 4. Placement run-time of benchmark using a macro vs. baseline placing logic blocks across the macro boundary.

Fig. 2 shows a scatter plot, where each point represents a comparison of the two flows for one macro. Also shown is a line representing linear regression analysis of all data points. On the y-axis, the figures show the sum of the HPWL of all nets connected to the macro (i.e. intra- and inter-macro nets) in $Flow_{macros}$ relative to $Flow_{syn\_macros}$. On the x-axis, the figures show the proportion of those nets that cross the macro boundary (i.e. inter/(inter+intra)). The trend in the figure makes intuitive sense: macros with a greater proportion of nets that cross the macro boundary lead to worse HPWL results than macros with a greater proportion of nets that are completely contained within the macro. Since the logic in the macros is optimized independently from the surrounding logic, the nets that cross the boundaries are not well optimized. Conversely, if very few nets cross macro boundaries, the nets connected to the macro can be very well optimized when ignoring logic outside the macro.

Fig. 3 is also a scatter plot, in this case showing the sum of the HPWL of all nets *not* connected to a macro (dust-to-dust) vs. the size of that macro as a fraction of the total number of Cyclone II LABs in the design. The figure is meant to illustrate the impact of having large macro placement blockages on the non-macroized logic. We observe that increasing macro size negatively affects the HPWL of dust-to-dust nets. In other words, big macros push little logic blocks around, and end up affecting the nets connected to those little logic blocks. Althought not shown in the figure, we found that inter-macro nets also tend to get worse as macro size increases. One reason for this could be that with larger macros, it is possible for a LAB to be far from the macro periphery, which means that inter-macro nets needing to connect to that LAB must travel a long way.

On average, across all 174 benchmarks each with a single macro, the HPWL of nets completely contained within a macro was 17% better when keeping the logic blocks in the macro fixed, relative to the macro origin. The HPWL of nets connected to the macro, but not completely contained within it, was 48% worse, and the nets not at all connected to the macro was 2% worse.

Looking to Fig. 4, each data point represents two placement runs: one of which is the time necessary to place a benchmark which uses one of the 174 macros, and the other of which is the time necessary to place the same benchmark using $Flow_{syn\_macros}$. The y-axis shows the $Flow_{macros}$ placement time divided by the $Flow_{syn\_macros}$ placement time. The x-axis shows the fraction of overall area taken up by the particular macro. As expected, the figure clearly shows that placement run-time is reduced to a greater extent when a greater proportion of a benchmark is pre-compiled in a macro.

2) Effect on Synthesis QoR: To consider the impact of cross-boundary optimization on synthesis QoR, we compared the post-placement HPWL of $Flow_{syn\_macros}$ with $Flow_{no\_macros}$. In $Flow_{no\_macros}$, Quartus II was allowed to see the logic inside the macros during synthesis (rather than seeing macro stubs filled with empty logic), and so it optimized across macro boundaries. The resultant netlist was passed to HeAP, which performed placement and reported the resultant HPWL. This was compared to HPWL obtained in $Flow_{syn\_macros}$ to isolate the impact of disabling synthesis optimizations across macro boundaries.

Fig. 5 shows the effect of using macros on synthesis QoR. The y-axis shows the HPWL of a benchmark that uses a macro divided by the HPWL of that same benchmark when not using a macro. The x-axis shows the fraction of total LABs that are in the macro. The figure shows that having a greater portion of a circuit macroized at synthesis time will generally lead to worse QoR, though there is more noise in this data than in the placement data. Fig. 6 shows the synthesis run-time (relative to $Flow_{no\_macros}$) vs. the fraction of total LABs that are in a macro. The figure shows that as a greater portion of a benchmark is macroized, synthesis run-time decreases close to linearly.

VI. Multi-Macro Evaluation

Having looked at designs containing single macros, we now move on to evaluate the case of multiple macros. We use ten benchmarks to evaluate the effect of macroizing multiple

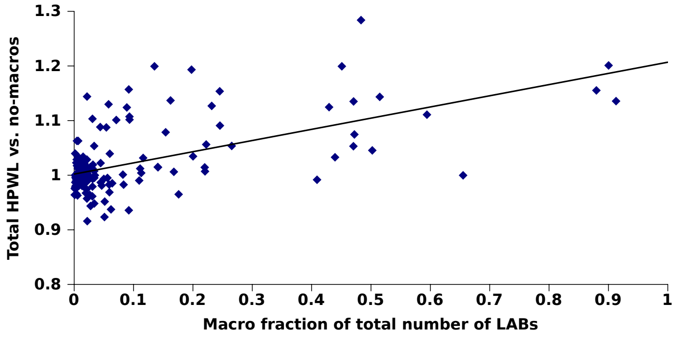

Fig. 5. HPWL impact of using a macro in synthesis vs. the fraction of overall LABs that are in a macro.

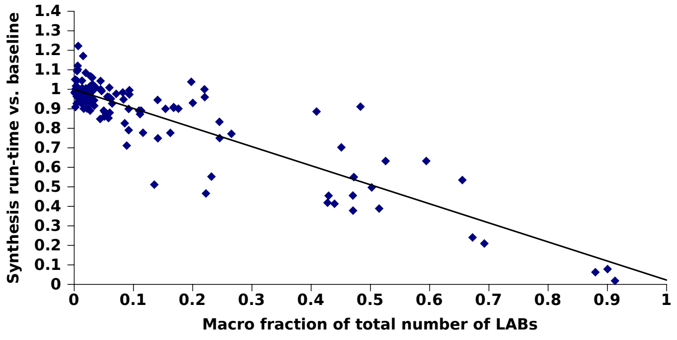

Fig. 6. Synthesis run-time of benchmark using a macro vs. baseline of synthesizing across the macro boundary.

sections of code. Each benchmark uses between three and ten pre-synthesized/placed macros, chosen either from basic blocks or from functions. All macros are composed only of LABs. The benchmarks are as follows, with this first four drawn from the CHStone benchmark suite (and therefore not described in detail):

- **jpeg (49566 logic cells)** : 3 functions macroized.
- **adpcm (26647 logic cells)**: 9 functions macroized.
- **gsm (17964 logic cells)**: 10 basic blocks macroized.
- **dfsin (36340 logic cells)**: 8 basic blocks macroized.
- **Black-Scholes (45021 logic cells)**: Monte-Carlo-based European-style option price estimation in fixed-point. Four functions were macroized, chosen because they were general-purpose enough to be included as part of a fixed-point or financial applications library.
- **mcml (21017 logic cells)**: Monte-Carlo modeling of light transport in multi-layered scattering media (MCML) models how light interacts with multi-layered tissues in fixed-point, similar to Black-Scholes. Five functions were macroized.
- **fft (11147 logic cells)**: A Fast Fourier Transform (FFT) `C` implementation was split into six different calls to a "butterfly" function, each of which was macroized.
- **df (36922 logic cells)**: The double float (df) benchmark was generated by stitching together division, exponential, and addition double-precision CHStone floating point benchmarks.
- **hash (5114 logic cells)**: 4 hashing functions macroized.
- **mandelbrot (21456 logic cells)**: Macroized four instances of mandelbrot, each operating on 1/4 of the data.

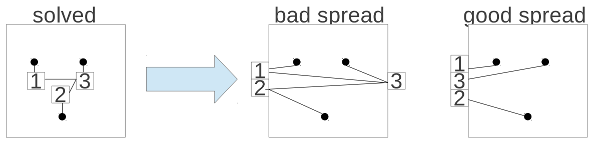

Fig. 7. Example of bad spreading that can occur with macros and dust. Because the spreading step maintains the ordering of blocks by x or y location provided by the solver, blocks 1 and 2 must be spread left of the macro block and block 3 must be spread right of the macro block, leading to a bad spreading. A better spreading is shown to the right, with blocks 1, 2, and 3 kept together.

A. *Macros-and-Dust Placement*

Placing big movable objects together with small movable objects poses unique placement challenges. Simulated annealing (SA)-based placement can be used, but finding legal object swaps becomes much more complicated when considering big blocks, as multi-step swaps that involve many different blocks need be considered. Additionally, swapping big blocks can result in large cost swings and while this may be desirable when the SA schedule is still "hot", it will interfere with the fine-grained tweaking that occurs during the "cold" phase.

Analytical placement (AP) approaches seem to be a better fit for macros-and-dust placement. During the solving step, x and y offsets into macros can be included in the formulation so that non-macro blocks connected to a specific block within a macro are placed close to that particular internal block. The spreading step is already designed to legalize an initial congested solution, so it is natural for it to spread dust away from a big macro. The spreading step is not without issues, however. With many big blocks of various shapes and sizes, spreading may encounter issues more akin to those found in floorplanning, rather than to those typically found in placement. Also, an issue that arises is the possibility of the spreading step perturbing the solved solution in an unusual way. For example, dust that should ideally be placed closely on one side of a macro could get spread to either side of a macro, as shown in Fig. 7. This is because the spreading step attempts to maintain the xy-order of blocks, including macros. To overcome this issue, as well as others, we devised and evaluated four macros-and-dust placement approaches:

1) **Floorplacement**: This approach was presented for the ASIC domain in [13]. It builds upon SimPL [9] to handle "boulders", which are similar to our macros. We augmented HeAP (based on SimPL) in the same way: During the recursive bipartitioning spreading phase, when boulders in a partition constitute a large enough fraction of the overall area of that partition, their locations are locked based on the results of floorplanning the big blocks in that partition. The dust locations, however, are left unfinalized. The boulders are then treated as obstacles for the dust, and spreading continues on the dust.
2) **Fix macros every second solve step**: When placing macros with dust, the locations of the dust can be significantly perturbed by slight changes in the xy-block ordering from the solver, since this can cause the dust to be placed on one side or another of a large macro, as in in Fig. 7. We minimize the perturbations experienced by the dust by only re-solving macro locations every second iteration. For example, in iteration 1, the solver solves for both macros and dust and spreads only the macros. In

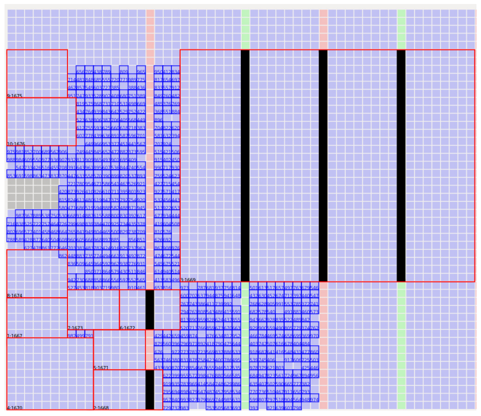

Fig. 8. Floorplacement flow on gsm benchmark where macros are shifted and fixed after 3 iterations. Dust is more centralized than without shifting and fixing of macros.

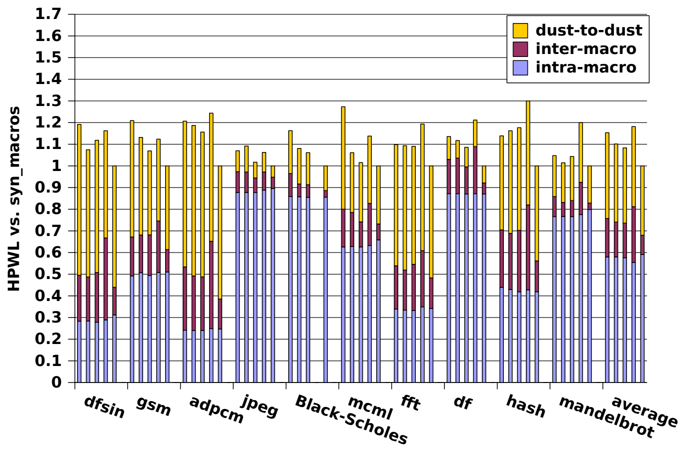

Fig. 9. HPWL of five flows separated into net categories. For each benchmark, a bar is shown for each flow, which are, from left to right: $floorplacement$, $floorplacement_{alternating}$,$floorplacement_{fixed\_macros}$, $partitioned$, and $Flow_{syn\_macros}$.

iteration 2, the macros are fixed to their spread locations and the formulation is adjusted such that, during the solving step, any dust connected to a macro is attracted to the macro's fixed location. Everything is then spread together. We refer to this flow as $floorplacement_{alternating}$.

3) **Fix macros early**: Another approach that we considered is to fix macro locations after a few AP iterations and treat them as obstacles for the remainder of the placement process. The macros are fixed based on the spread locations in the first 5 iterations that resulted in the best HPWL. Additionally, we found that reducing the blockages presented by these macros helped to improve overall HPWL, so the macro locations are shifted in the direction of the closest FPGA edge, ensuring no overlaps with other macros. The macro locations are based on the spread locations, so a macro located in the bottom-left corner, for example, will remain in the bottom left corner. However, by shifting the macros, a large unobstructed region is created in the middle of the FPGA, where dust can be placed. Fig. 8 shows the result of using this flow on the gsm benchmark. The macros are generally forced to the corners, while the dust is generally in large unobstructed regions. We refer to this flow as $floorplacement_{fixed\_macros}$.
4) **Place macros and dust separately**: The placement area is partitioned into macro space and dust space. The macros are first floorplanned into one partition, and the dust is then placed into the other partition. While this improves the QoR of the connections between dust, connections between macros and dust are difficult to minimize. We refer to this flow as $partitioned$.

Further details on the placement approaches could not be included owing to space limitations. The interested reader is referred to [6] for complete details.

We compared the four placement approaches using the set of 10 benchmarks and the resultant HPWL for the three different classes of nets were compared. Additionally, the run-time and HPWL relative to $Flow_{syn\_macros}$ was analyzed.

Fig. 9 shows the $Flow_{macros}$ HPWL for each of the four placer flows described above, in addition to the HPWL for $Flow_{syn\_macros}$. The difference between each of the four flows and the baseline $Flow_{syn\_macros}$ represents the impact of using macros on placement QoR, without considering synthesis QoR, since the technology-mapped netlists that are used as input to placement are the same for all flows. HPWL is shown on the y-axis relative to the baseline $Flow_{syn\_macros}$ HPWL. A value greater than 1 represents a worsening of HPWL vs. the baseline. For each benchmark, five bars are shown, which represent, from left-to-right, the following flows: $floorplacement$, $floorplacement_{alternating}$, $floorplacement_{fixed\_macros}$, $partitioned$, and $Flow_{syn\_macros}$. The right-most set of bars gives the average. Each bar is split into three components: the HPWL of nets not completely contained within one macro (labeled "inter-macro"), the HPWL of nets completely contained within one macro (labeled "intra-macro"), and the nets that do not touch any macros (labeled "dust-to-dust"). Note that for the Black-Scholes benchmark, using the $partitioned$ flow led to an impossible placement, because the area necessary to floorplan the macros did not leave enough room to place the dust in the other partition.

The best macro flow is $floorplacement_{fixed\_macros}$, which includes the methods in $floorplacement_{alternating}$, and results in a 9.9% improvement, on average, for the inter-macro nets and a 12.3% improvement, on average, for the dust-to-dust nets for an overall HPWL improvement of 6.1% vs. $floorplacement$, on average.

Fig. 10 shows the run-time speed-up for each of the four flows (normalized to $Flow_{syn\_macros}$). The figure shows that speed-ups vary greatly based on the benchmark, each of which has a different proportion of overall logic pre-compiled in macros (the jpeg benchmark, for example, has 95.8% of its LABs in macros). Considering the macros and dust together throughout every iteration ($floorplacement$ flow) slows down the placement by approximately 2× vs. the partitioning flow which considers macros and dust separately. Alternating solving for macros and everything else speeds the macros-and-dust placement slightly, since considering only macros in placement is faster than considering everything. Fixing the macros in an early iteration significantly speeds up the placement, since macro placement only has to be performed for a few early iterations. The overall average speed-up compared to $Flow_{syn\_macros}$ is 4.9× for the $floorplacement$

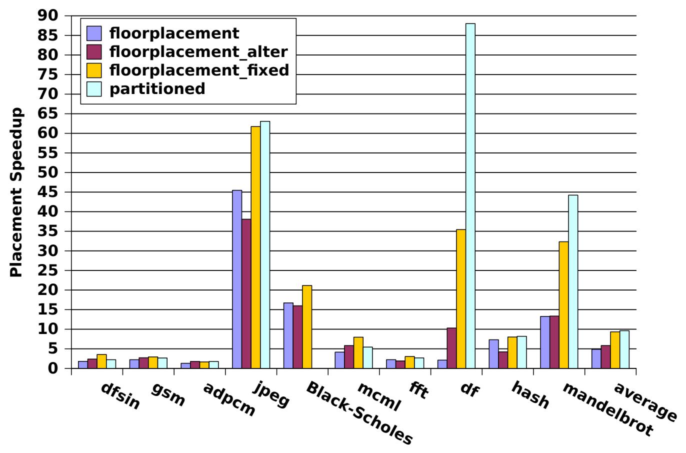

Fig. 10. Runtime speed-up of four flows relative to baseline. For each benchmark, a bar is shown for each flow, which are, from left to right: $floorplacement$, $floorplacement_{alternating}$, $floorplacement_{fixed}$, and $partitioned$.

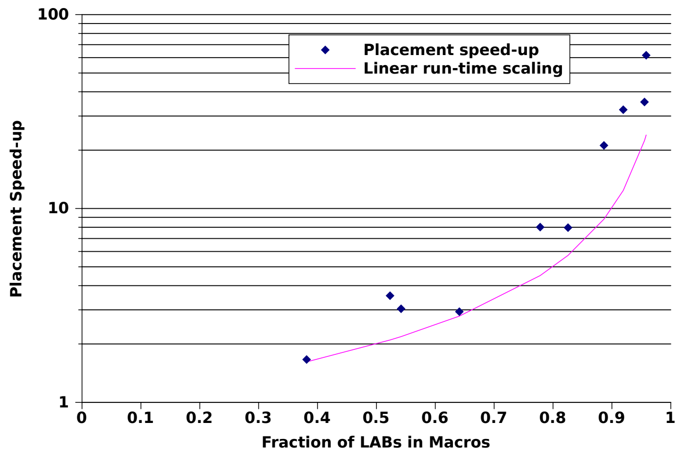

Fig. 11. Placement speed-up for each benchmark vs. the fraction of LABs that are in macros.

flow, $5.8\times$ for the $floorplacement_{alternating}$ flow, $9.3\times$ for the $floorplacement_{fixed\_macros}$ flow, and $9.6\times$ for the $partitioned$ flow.

B. Macro Flow Evaluation

Using the $floorplacement_{fixed\_macros}$ flow described above, we examine the trade-off between QoR and run-time for the 10 benchmarks. Fig. 11 shows the placement speed-up for each of the 10 benchmarks vs. the fraction of LABs that are in macros. Also shown in this figure is a trend line corresponding to a run-time that scales linearly with the number of LABs not in macros. Observe that run-time grows at a greater than linear rate with number of LABs not in macros, which is a symptom of the worse-than-$O(n)$ computational complexity of placement. Fig. 12 shows a similar plot for synthesis speed-up.

Fig. 13 shows the QoR and run-time of $Flow_{macros}$ for all 10 benchmarks relative to $Flow_{no\_macros}$, which uses no macros in either synthesis or placement. The QoR metrics used are routed wirelength (not HPWL) and $Fmax$, which were obtained from Quartus II report files. The run-time represents whole-flow run-time, which means synthesis, placement, and

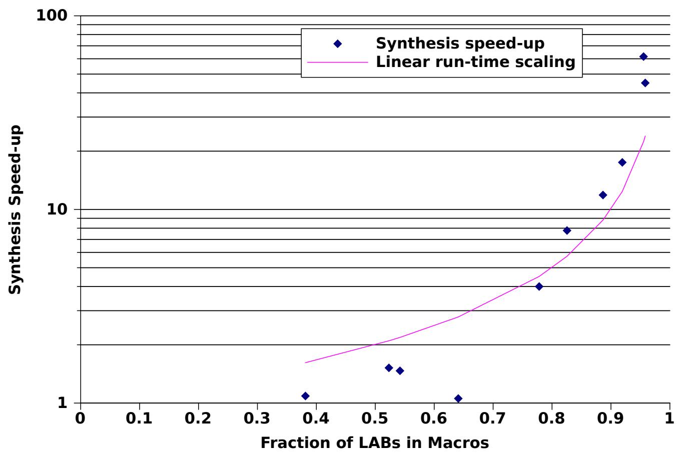

Fig. 12. Synthesis speed-up for each benchmark vs. the fraction of LABs that are in macros.

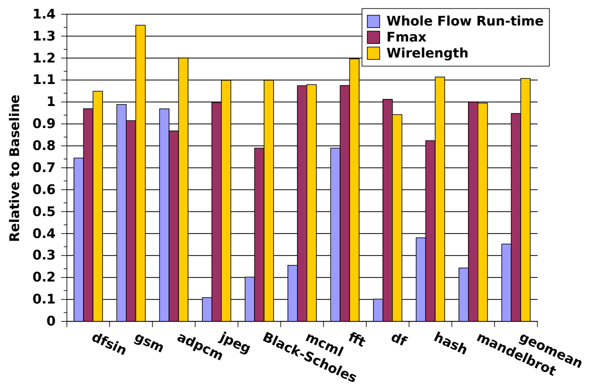

Fig. 13. Whole-flow run-time, post-routing wirelength, and post-routing $Fmax$ for 10 benchmarks, relative to the baseline using no macros. Also shown is the geometric mean across the benchmarks using macros relative to the geometric mean across benchmarks not using macros.

routing. The geometric mean across the benchmarks using macros shows that run-time is reduced to 35% of its original value (2.9$\times$ speed-up) at the cost of 5% worse $Fmax$ and 11% worse wirelength. The best case is the "DF" benchmark, which experiences a 9.9$\times$ whole-flow speed-up with 1.2% better $Fmax$ and 6.1% better wirelength. The worst case is the GSM benchmark, which experiences no speed-up (1.01$\times$) at the cost of 9.3% worse $Fmax$ and 35% worse wirelength. Note that while the macro flow causes a speed-up to both synthesis and placement, it actually causes a slow-down to router run-time, which is taken into account in the speed-ups reported above. The router slow-down occurs because the placements that are generated by our macro flow are generally worse than those generated by the flat flow. Worse placements mean that longer routes must be taken for source-sink connections. This causes the router to explore a larger part of the routing resource graph and increases routing congestion.

Table I shows the complete results for the macros flow vs. three baselines: $Flow_{no\_macros}$, $Flow_{syn\_macros}$, and the regular timing-driven Quartus flow (no HeAP) with default effort options. Note that the "syn run-time" column for the Quartus II flow is also the synthesis run-time for $Flow_{no\_macros}$ and the "syn run-time" column for $Flow_{syn\_macros}$ is also the synthesis run-time for $Flow_{macros}$.

The table shows that of the 5% $Fmax$ and 11% wirelength

TABLE I. MULTI-MACRO QOR AND RUN-TIME RESULTS.

	Quartus II						no_macros					syn_macros						macros				
Benchmark	syn run-time	place run-time	route run-time	total run-time	Fmax	WL	place run-time	route run-time	total run-time	Fmax	WL	syn run-time	place run-time	route run-time	total run-time	Fmax	WL	place run-time	route run-time	total run-time	Fmax	WL
DFSIN	233	33	31	297	18	395	7.5	27	268	15	384	148	7.6	40	195.6	16	379	2.1	43	193	14	403
GSM	341	13	15	369	54	202	3	12	356	47	182	323	2.4	25	350.4	44	239	1	28	352	43	246
ADPCM	398	28	22	448	45	295	5.4	19	422	41	281	364	4	37	405	38	318	3.3	40	407	35	337
JPEG	589	51	56	696	21	466	11.7	43	644	18	475	13	10.2	63	86.2	18	528	0.2	56	69	18	522
Black-Scholes	276	171	53	500	19	341	9.8	42	328	17	369	23	9.1	43	75.1	14	407	0.5	42	66	13	406
MCML	178	17	18	213	37	221	3.5	15	196	36	194	23	2.7	27	52.7	38	212	0.4	27	50	39	209
FFT	68	6	8	82	47	98	1.7	6	76	41	96	47	1.3	13	61.3	44	109	0.6	13	61	44	114
DF	430	42	33	505	42	390	8.2	27	465	37	385	7	7	41	55	36	351	0.2	40	47	38	362
Hash	28	3	3	34	34	41	0.8	3	32	33	44	7	0.5	5	12.5	26	46	0.1	5	12	27	49
Mandelbrot	109	15	17	141	42	212	3.8	15	128	42	199	6	3.2	27	36.2	42	201	0.1	24	30	42	198
geomean	196	21	19	243	34	221	4.3	16	218	30	215	33	3.5	27	82	29	233	0.5	27	76	29	238
vs Quartus							5.0x	1.2x	1.1x	91%	98%	5.9x	6.1x	0.7x	3.0x	87%	105%	46.9x	0.7x	3.2x	86%	108%
vs no_macros												5.9x	1.2x	0.6x	2.6x	96%	108%	9.3x	0.6x	2.9x	95%	111%
vs syn_macros																		7.6x	1.0x	1.1x	99%	103%

degradation that occurs as a result of using macros, very little occurs as a result of degradation in the placement step (1% $Fmax$ and 3% wirelength degradation), which means that the main source of quality degradation is in synthesis. The table also shows that for these benchmarks, synthesis dominates total CAD time, which is an unexpected result, as conventional wisdom is that placement and routing are the most run-time intensive steps. This points to a potential area of future research, which is how circuit structures produced by HLS affect CAD run-time, or how synthesis can be tuned to handle the circuit structures produced by HLS. Finally, when examining placement vs routing run-time, the latter dominates when using macros in placement, which means that to get significantly lower whole-flow run-times, macros must also include pre-routed signals.

Comparing with Quartus II, we observe a whole-flow speed-up of 3.2×, with 8% higher routed wirelength and 14% worse $Fmax$. The results are encouraging in that they illustrate significant run-time speed-ups are possible relative to highly tuned commercial tools, for a modest QoR hit. Moreover, note that the comparison with Quartus is not entirely "apples-to-apples" as the Quartus placer is timing-driven, whereas HeAP is only wirelength driven. Thus, we expect some of the speed-performance loss can be recovered.

Overall, compared to both a completely commercial toolflow, as well as a hybrid commercial/academic flow that incorporates the HeAP placer, the proposed macro flow offers whole-flow run-time speed-ups of roughly 3× with a relatively small performance degradation in the 5-14% range, on average. We expect that our "library-assisted" compilation strategy will be of interest both to improve engineering productivity, and to enable a move to software-like compile times in HLS flows.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we presented an approach to reduce synthesis and placement run-times by making use of a library of pre-compiled macros, accessed in high-level synthesis. Through the use of macros, a significant fraction of the back-end tool work is replaced by the instantiation-and-stitch of pre-compiled solutions. We showed that run-time speed-ups were superlinear with respect to the fraction of logic in macros. The QoR impact of using macros varied significantly – using macros with a large amount of internal connectivity and a small amount of external connectivity led to the best QoR. Overall, comparing the whole-flow run-time of the proposed library-based macro methodology vs. Quartus II, we observed 3.2× speed-up for 14% performance loss, on average. Comparing whole-flow run-time vs. a flat flow that uses the HeAP analytical placer (and Quartus for the remaining flow steps), we see 2.9× speed-up with 5% performance loss, on average.

Future work includes handling heterogeneous macros (comprising blocks other than LABs), support for macros with fixed routing, and enhancing HeAP to be timing-driven.

REFERENCES

[1] 40Gbit AES Encryption using OpenCL and FPGAs. *http://www.nallatech.com/Technical-Library/white-papers.html*.

[2] J. Babb and et al. The RAW benchmark suite: Computation structures for general purpose computing. In *IEEE FCCM*, 1997.

[3] 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 International Symposium on FPGAs*, pages 33–36, 2011.

[4] D. Chen and D. Singh. Using OpenCL to evaluate the efficiency of CPUs, GPUs and FPGAs for information filtering. In *FPL*, pages 5–12, 2012.

[5] D. Chen and D. Singh. Fractal video compression in OpenCL: An evaluation of CPUs, GPUs, and FPGAs as acceleration platforms. In *ACM/IEEE ASP-DAC*, pages 297–304, 2013.

[6] Removed for blind review (PhD thesis).

[7] M. Gort and J. Anderson. Analytical placement for heterogeneous FPGAs. In *FPL*, pages 143–150, 2012.

[8] 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*, 17(0):242 – 254, 2009.

[9] M.-C. Kim, D. Lee, and I. Markov. SimPL: An effective placement algorithm. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems (TCAD)*, 31(1):50–60, 2012.

[10] C. Lavin, B. Nelson, and B. Hutchings. The impact of hard macro size on FPGA clock rate and place/route time. In *FPL*, 2013.

[11] 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*, pages 117–124, 2011.

[12] LLVM. *The LLVM Compiler Infrastructure Project (http://www.llvm.org)*, 2010.

[13] J. A. Roy, S. N. Adya, D. A. Papa, and I. L. Markov. Min-cut floorplacement. *IEEE Trans. on CAD*, 25(7):1313–1326, July 2006.

[14] R. Tessier. Fast placement approaches for FPGAs. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 7(2):284–305, 2002.

[15] United States Bureau of Labor Statistics. *Occupational Outlook Handbook 2010-2011 Edition*, 2010.

[16] S. Weston, J. Spooner, S. Racaniere, and O. Mencer. Rapid computation of value and risk for derivatives portfolios. *Journal of Concurrency and Computation: Practice and Experience*, 24(8):880–894, 2012.