Share with Care: A Quantitative Evaluation of Sharing Approaches in High-level Synthesis

Alex Kondratyev, Luciano Lavagno, Mike Meyer, Yosinori Watanabe
Cadence Design Systems, San Jose, USA

***Abstract*— This paper focuses on the resource sharing problem when performing high-level synthesis. It argues that the conventionally accepted synthesis flow when resource sharing is done after scheduling is sub-optimal because it cannot account for timing penalties from resource merging. The paper describes a competitive approach when resource sharing and scheduling are performed simultaneously. It provides a quantitative evaluation of both approaches and shows that performing sharing during scheduling wins over the conventional approach in terms of quality of results.**

I. INTRODUCTION

One of the main premises of high-level synthesis (HLS) is a superior capability to perform design exploration than the RTL flow. This leads to a higher quality of results and a shorter time to market. Indeed, by providing effective means for making microarchitecture decisions such as pipelining, inlining functions and loop unrolling, the HLS tools explore implementations in which area/power and throughput may vary in wide ranges (see e.g. [1] that reported 2-3x area and up to 20x power variations for different microarchitecures of a JPEG algorithm). Targeting mainly at design space exploration reduced the research interests towards improving the core synthesis routines, since they provide significantly smaller returns in comparison to the huge improvements achieved through choosing the best architecture.

However, we argue that even focusing on relatively well-researched aspects such as the core scheduling routines can provide significant Quality of Results (QoR) improvements in a commercial HLS tool that has been adopted by many semiconductor and system companies. By analyzing the usage of the tool over the last 5 years, we made the following observations:

1. Design exploration is indeed a highly important task in the beginning of the design cycle.
2. It is rare that the microarchitecture changes significantly in the end of the design cycle.
3. Once a microarchitecture is chosen, designers aim to squeeze every single transistor and/or picosecond by iteratively performing detailed optimizations in synthesis runs. Having 5%-10% QoR improvements (using better core algorithms e.g.) provides a competitive edge against manual designs or other HLS tools.

One of the key optimizations used for the QoR improvements in HLS is resource sharing, i.e. the problem of using the same hardware resources to implement multiple operations of the same kind. The operations such as multiplications require resources that are expensive in terms of area, delay, and dynamic power, and if a single resource can be used to implement multiple operations that are scheduled in different clock cycles, it can lead to a significant improvement of the quality of the implementation.

This paper focuses on this problem: resource sharing in HLS. We question the basic assumption broadly employed in HLS, both in the academia and commercial tools, that the task of scheduling of operations is separated from the task of binding resources to implementing the operations, and propose a practical approach that effectively combine the two tasks and provide the experimental evidence of the effectiveness based on a broad set of industrial design examples.

Historically, the task of high-level synthesis is divided into resource allocation, scheduling and resource binding [2]. Allocation determines which resources will be used, then scheduling answers the question of when (at which state) every operation of the specification is executed, while binding specifies a particular resource (from the allocated set) to implement each operation while considering resource sharing.

The approaches known in the literature either solve these problems sequentially or take a naïve formulation of the combined problem, which is too expensive to solve for practical designs. In [3] it was argued that these problems must be solved together, in order to obtain a high-quality implementation, which (1) is competitive with manual design and (2) is guaranteed to be implementable within the given timing constraints. The difficulty is due to the tight relationship between scheduling and binding, because the choice of a resource to implement an operation and its schedule are mutually dependent. For example, if some addition operation is on the critical path, then the scheduler may either choose the fastest implementation (e.g., using carry lookahead) or schedule it in a later clock cycle using a slower but cheaper implementation (e.g., using a ripple-carry adder). Moreover, ignoring false paths during scheduling may lead to an underestimation of both the critical path and of the number of resources that will be needed by binding in order to avoid false paths. Although this observation was already made in [17], most current research and commercial high-level synthesis tools still perform scheduling and binding as two separate phases.

On the other hand, performing binding during scheduling has its own set of issues. Besides being more complex, it suffers from the lack of a global view on the binding problem, because the decisions on which operations to share need to be done based on a partial schedule.

These issues identified in the individual approaches – the post-scheduling binding versus combined approach - are orthogonal, which makes it difficult to provide theoretical arguments to favor one over the other. We performed a

978-3-9815370-0-0/DATE13/©2013 EDAA

quantitative evaluation of the both approaches by implementing them in our tool. We compared the results of synthesis runs on over a very broad set of 100 industrial designs (not just small benchmarks), and show that with our implementation of ***combining binding and scheduling, the results are superior to the post-scheduling binding approach in the majority of cases.***

II. PROBLEM ILLUSTRATION

Consider the SystemC specification shown in Figure 1. Assume that the desired clock period is 2000ps and the throughput for producing a single pixel is 3 clock cycles (which suggests that the latency of a single iteration of the inner while_loop is 3 states). In HLS the input specification is represented by its control flow graph (CFG) and data flow graph (DFG) [4]. Nodes of the CFG either serve to fork/join control flow (conditionals and loops in SystemC) or correspond to "wait()" calls in SystemC. The DFG nodes are operations, and the DFG edges are data dependencies between nodes. Figure 2(a)(b) shows the CFG and DFG for Figure 1. Note that by transforming the "if-statement" of SystemC code into the so called predicate form (roughly corresponding to rewriting `if (aver > th) {aver = aver & scale;}` into `aver = aver > th ? aver & scale : aver;)` one can eliminate the fork-join structure from the CFG corresponding to the loop (see [3] e.g.).

```
void example1::thread() {
    while (true) {
        int aver = 0;
        wait(); // s0
        int filt = mask;
        do {
            delta = mask * chrome;
            aver += delta;
            if (aver > th) {
                aver = aver & scale;
             }
            filt += chrome;
            pixel = aver * filt;
         } while (delta != 0);
}}
```

Figure 1. Example of SystemC specification

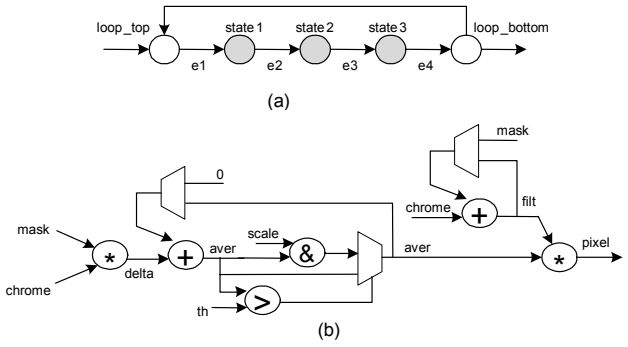

Figure 2. CFG and DFG for the specification example

The set of resources for the DFG is presented in Table 1 where delays are annotated using the artisan_90nm_typical library.

TABLE 1. INITIAL SET OF RESOURCES WITH DELAYS

resource	mul	add	gt	flop	and	mux
delay (ps)	930	350	220	40/70	50	110

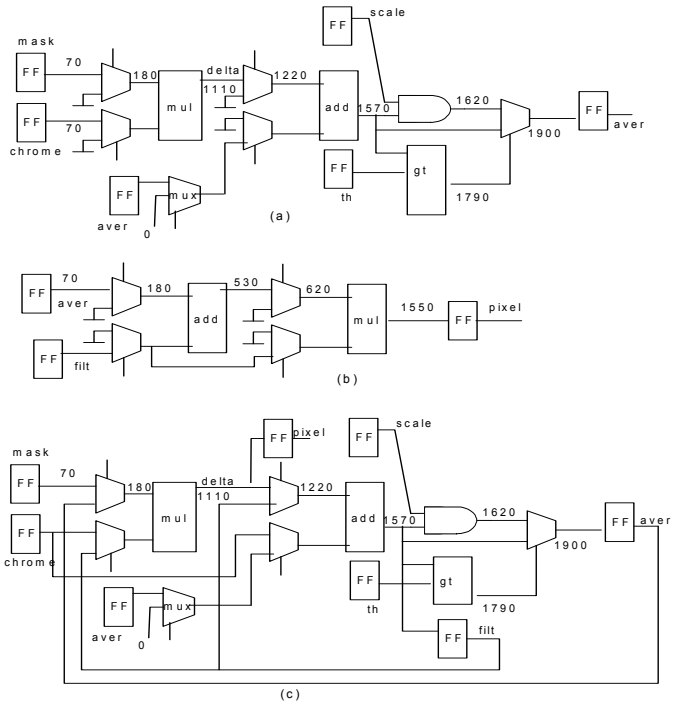

Figure 3. Scheduling netlists

Let us consider scheduling of the datapath from Figure 2(b) with a subsequent binding step. To simplify the illustration we will leave aside mux operations that feed back values for the next iteration (similar to Phi nodes in the Static Single Assignment compiler Intermediate Representation). The scheduling process is represented conceptually as a bin packing problem instance, where bins are CFG edges and items to pack are DFG operations. Bin packing proceeds until either the timing constraint (clock period) or a resource bound (number of available resources of a given type) is violated for a clock cycle. Timing is conveniently checked by building a netlist (called scheduling netlist) corresponding to a set operations that are bound to a connected set of CFG edges without states in between (i.e. a clock cycle). The scheduling netlist for edge *e1* of CFG is shown in Figure 3(a). The validity of timing is confirmed by annotating nets on the critical paths with their arrival times (for simplicity of this illustration (but not in the tool) we account only for resource delays (see Table 1) and omit the delays of interconnects and wireloads). One can see that the netlist of Figure 3(a) has a positive slack under the clock period of 2000ps.

The datapath from Figure 2 has two types of resources that are worth sharing: adders and multipliers. The trivial constraint on the number of these resources would be to allocate a single resource for each type. This constraint is clearly satisfied in the scheduling netlist of Figure 3(a). Note that multipliers and adders are instantiated in the netlist with muxes at their inputs with one mux input grounded. This is done to help with the timing convergence of the binding step when these resources would be shared between states through the use of input muxes. The scheduling netlist for edge *e2* of the CFG is shown in Figure 3(b). It also satisfies timing and resource constraints. As all operations were successfully

assigned to CFG edges, the scheduling step is over and we can proceed with the binding step for resource sharing.

Binding usually starts from the most costly resources, which in our case are multipliers. However an attempt to merge multipliers in the netlists shown in Figure 3(a) and (b) fails the timing check. Indeed the arrival time of the multiplier in Figure 3(a) is 180ps while for the multiplier in Figure 3(b) it is 630ps. Merging these two multipliers will increase the critical path for the netlist Figure 3(a) by 450ps and will result in a negative slack violation. *The violating path is a false path* because it cannot be sensitized under any reachable combinations of mux control values. False paths could be reported to the downstream logic synthesis tool. However, this greatly reduces the room for optimization, because currently logic synthesis must preserve the pins specifying the path, and hence it is generally forbidden by RTL coding rules used by design companies. Instead, we treat all false paths as if they are real. This may require extra resources, but in practice it improves the final post-synthesis quality of results and satisfies established RTL coding rules. The timing violations stemming from the false path force to reject merging of multipliers for the netlists in Figure 3(a) and (b). For the very same reasons merging of adders in Figure 3(a) and (b) is timing infeasible as well. In the end the post-scheduling binding step must produce an implementation that has 2 multipliers and 2 adders.

Let us consider another possible complication during the binding step, ignoring timing penalties from false paths. For this we increase the clock period to 3000ps and proceed with resource sharing. Merging of multipliers for the netlists of Figure 3(a) and (b) would succeed under this clock period. However, proceeding with adders sharing would fail because the resulting network would have a *false combinational cycle* and would be rejected by logic synthesis. Indeed in the first state of the loop the output of the multiplier is fed into the adder while in its second state it is the output of the adder which is fed into the multiplier. Constraining logic synthesis to cut the cycle under different conditions would again lead to sub-optimal results and is rejected by RTL coding rules.

Two observations about the shortcomings of post-scheduling binding could be made from the above example

1. Timing penalties coming from false paths significantly narrow sharing opportunities.
2. Sharing can be prevented by combinational cycles even for designs with large positive slack.

The above problems can be identified and immediately avoided in the design flow when scheduling and binding are combined. In this flow the scheduling network generated for *e1* coincides with the one in Figure 3(a). When scheduling *e2*, the binding of the second add operation to the single adder resource is feasible because it does not increase the length of the critical path. However, binding the second multiply operation in the same state to the single multiplier resource would fail because of the timing violation due to the false path and the creation of the combinational cycle. The scheduling of the multiply operation in *e2* would be rejected. This operation could be successfully scheduled in *e3* and the multiplier resource can be shared between *e1* and *e3* safely. The final netlist after sharing is shown in Figure 3(c) (note that all cycles are broken by flops). It gives an optimal implementation with a single adder and a single multiplier.

On the other side, it is also possible to construct an example in which binding done as a post-scheduler step is better than the combined approach. Let us consider a hypothetical schedule of 4 multiply operations scheduled in 4 different cycles with 2 available multipliers (see Figure 4). Sharing after scheduling benefits from having a complete picture of all competing operations and their arrival times. In this case it is clear that sharing operations *mul1,mul3* and *mul2,mul4* comes for free because their arrival times are the same. This is much harder to see in doing sharing on fly during scheduling. While scheduling *mul2* one needs to decide whether to bind it to the same multiplier as *mul1* and worsen the critical path by 200ps (due to merging) or to bind it into the second multiplier. As the arrival times for *mul3* and *mul4* are unknown yet, making this decision is hard and may result in a non-optimal binding and the increase in the number of resources. Reconsidering this decision when *mul3* and *mul4* are scheduled, albeit possible in principle, would lead to prohibitive scheduling times.

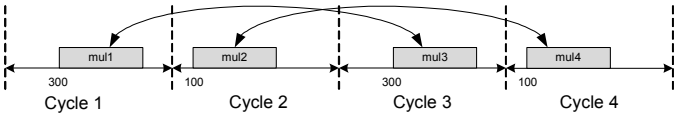

Figure 4. Global vs local aproaches for sharing

In the rest of the paper we will evaluate the efficiency of both sharing approaches and provide a quantitative evaluation of their quality. We focus on **resource sharing** problem leaving **register sharing** aside because register sharing is a simpler issue and performing it after scheduling is legitimate as it does not create false paths and disturb timing.

III. PRIOR WORK

It is almost universally accepted that high-level synthesis systems and tools perform scheduling and binding separately. This methodology is advocated both in the research community (starting from early textbooks [2] to recent reviews [5]) and in commercial tools (see NEC's CyberWorkBench [6], Cynthesizer by Forte [7] and CatapultC from Calypto [8]). It is based on the rationale that timing information is essential for the binding and precise timing gets available only after scheduling. Once a schedule is known the binding problem can be reduced to iterative construction of a weighted bipartite graph containing operations and resources, finding a maximum matching and combining nodes that are matched [9]. Several techniques were proposed for matching, e.g. based on network flow [10] or clique partitioning [9] to name a few. The main deficiency of this classical binding approach is that it neglects false path timing penalties coming from sharing, as was first argued by [17] and as we showed in Section II.

Combining scheduling and binding in a single step is difficult because these problems are mutually dependent. Several approaches to solve the combined problem optimally

were proposed based on simulating annealing [11] or ILP formulations [12]. These techniques are not scalable for practical designs. [13] suggested to interleave scheduling and binding when binding done as a postscheduler step provides soft constraints to the next iteration of scheduling. This is an interesting approach but it does not provide guarantees because soft constraints could be dropped in scheduling. [3] proposed a heuristic approach for binding during scheduling but did not provide direct data to confirm its efficiency. To the best of our knowledge our work is the first one that evaluates the efficiency of both binding approaches and gives a quantitative evaluation of their QoR, as opposed to the small academic benchmarks used by [17].

IV. SCHEDULING IN HIGH LEVEL SYNTHESIS

In the tool considered in this paper, which is a good representative of the industrial state of the art, scheduling is an iterative procedure. At each pass (iteration), a latency-, clock cycle- and resource-constrained scheduling problem is solved, using a limited set of resources. If the scheduling pass fails, an internal expert system chooses an action to relax the constraints, e.g. adding more states, resources, or enabling branch speculation etc., subject to given design requirements, and initiates a new scheduling pass. In the sequel, we first provide an overview of a single scheduling pass to an extent relevant in considering the two different binding approaches: post-schedule binding vs. on-the-fly binding. We then describe the binding procedure for each case.

A. Pass scheduling

Scheduling starts from creating a lower bound on the set of resources required to schedule the given CDFG. For this, we define a mapping that relates every operation to compatible resource types, where a resource type is represented as a combination of the operation type (addition, subtraction, multiplication, etc.) with operand and result widths. For example, A1[7:0] + B1[4:0] and A2[5:0] + B2[6:0] could be implemented by an 8x6 bit adder. We do not merge resources of very different bit widths, to avoid a negative impact, e.g., on power consumption. Then we create a set of intervals through the intersection and union of ASAP/ALAP ranges of operations of compatible types. Finally, we estimate the resource demand for every interval and choose the lower bound to be the maximal among the demands for all intervals. This approach is similar to [14], with some enhancements for accounting timing information during building of intervals and mutual exclusivity of operations coming from the predicate transform (see Section II).

Each scheduler pass uses list scheduling [15], mostly due to its flexibility to introduce dynamic changes of priorities in operation scheduling. The priority function takes into account the mobility of the operations defined by timing-aware ASAP/ALAP intervals, the complexity of operations (more complex ones are scheduled first), the size of the fanout cone of an operation, etc. The description of the pass scheduler is presented in Figure 5.

Note that the high-level description of the pass scheduler is not explicit about which binding approach is taken. The difference

```
SCHEDULE_PASS(CFG C, DFG D, clock period T_clk,
Library L, User Constraints U)
forall edges in CFG {
   Ready ← operations ready to schedule;
   Compute_op_priorities(Ready);
   op_best ← highest priority op;
   Op_res ← resources compatible with op_best;
      forall r in Op_res {
        if (bind(op_best,r) == success) break;
      }
   if(op_best failed and e is last in lifespan){
      Failed_ops ← op_best;
   } else {Update(op_best, Ready);}
  if (Failed_ops != ∅) {return failure;}
}
```

Figure 5. Performing a single scheduling pass

is captured by the way in which the *Op_res* set is built and the *bind()* function is implemented.

1. For the post-schedule binding approach, the scheduler does not bind operations to specific resource instances, but rather chooses a type of resources for a given operation. Hence, *Op_res* captures a set of types of resource instances created by the allocator that are compatible for a given operation. For example if the allocator had created adders with 32, 24 and 16 input bits, then for an add operation *o* with 22-bit inputs, *Op_res* would consist of the 32-bit and 24-bit adder types. The purpose of the *bind()* function is to choose a resource type *r* in *Op_res* for the operation *o* such that (a) it satisfies timing constraints and (b) the number of scheduled ops competing with *o* for this type *r* is less than the resource bound given for this type.
2. For the on-the-fly binding approach, the set *Op_res* consists of all the instances of the allocated resources for a given operation (i.e. if the allocator had created 32 and 24-bit adders and the resource bounds for these adders were 10 and 20 respectively, then *Op_res* would have 10 and 20 instances of 32-and 24-bit adders respectively). For this case, *bind()* assigns a specific resource instance, rather than a resource type, to the operation. It does not need to keep track of resource bounds because the scheduling networks are built using concrete resource instances.

The complexity of the pass scheduler is $P = O(|E| * |O| * |FO\_aver| * |Op\_res|)$, where $|E|$ is the number of CFG edges, $|O|$ is the number of DFG operations, $|FO\_aver|$ is the average operation fanout (relevant for the function *Update*()) and $|Op\_res|$ is the size of the *Op_res* set. It is clear that performing sharing *on fly* is more costly, because for large designs the number of resources is significantly larger than the number of resource types.

When the pass scheduler fails, the set of scheduling constraints must be relaxed. The history of the scheduling pass is recorded in a set of restraints, which are issued every time the binding of an operation to an edge and/or a resource fails. Restraint analysis is done for the DFG fanin cones of the failed operations and of those whose scheduling suggests some room for improvement. Restraints are assigned weights based on their proximity to failed operations and the number of failures they help solve. Each restraint suggests a set of actions that can be applied to improve the scheduling. Timing restraints can be fixed by adding states to the CFG, by adding resources

or by speculating operations. Restraints stemming from combinational cycles forbid the use of a resource for an operation, etc. Every action has an estimated cost, which is combined with the number of restraints solved by this action and the restraint weight. The action with the best estimated gain wins and is used to relax the constraints for the next scheduling pass.

B. *Binding formulation and heuristics.*

On-the-fly binding

Input: Scheduling edge e, operation o, set of available resources for o $(R(o))$, clock period T

Output: mapping $o \rightarrow r \in R(o)$ (if exists) such that the binding is legal (satisfies timing) and has minimal cost.

The cost of the binding in this formulation is a function of:

- Resource delay (faster resources are preferable to get faster implementations)
- Resource area (smaller resources are preferable)
- Arrival times of o and r (closer arrival time introduce less penalty during merging)
- Cost of input muxes (if operation o has common inputs with the already bound resource r muxes get reduced)

Resources in $R(o)$ are sorted by the cost and are bound in the sorting order to heuristically ensure an optimal binding of the scheduled operation. Note that having optimal binding for every op locally does not guarantee the global optimality (as discussed in Section II) but works well in practice (see experimental results).

Post-schedule binding:

Input: Set of shareable operations O, mapping *sched:* $O \rightarrow E$, where E is a set of CFG edges, mapping *compatible:* $O \rightarrow R$, where R is a set of available resources, clock period T

Output: mapping *bind:* $O \rightarrow R$ such that the binding for every operation is legal (no timing violation, no combinational cycles) and is consistent with *compatible*.

Several formulations are known from literature (see Section III) to solve the post-scheduling sharing problem. The problem is reduced to a bipartite graph matching with minimal cost. The cost is formulated in terms of merit parameters such as slack, area, power. Clique-based or network flow methods are commonly used to provide guidance for cost minimization. The binding procedure used in this paper for comparison is shown in Figure 6.

```
PostBind: Operations O, Resource Types R_types,
Resource set R)
Sort R_types by cost (high cost first)
forall rt ∈ R_types {
   Build compatibility graph G for r∈R of type rt
   while G has edges do {
     Find the largest clique C in G
     Find the pair ri,rj in C with the minimal cost
     Merge ri and rj. If success recompute G.
     else delete edge(ri, rj) from G
} }
```

Figure 6. Post-schedule binding

The compatibility relation for resources r_i and r_j of the same type (represented by the compatibility graph) is defined by:

1. **Timing compatibility** guarantees that merging would not result in a timing violation. This information can be statically evaluated based on arrival and required times of r_i and r_j extracted from scheduling networks.
2. **Operation compatibility** ensures that operations bound to r_i and r_j are either scheduled in different CFG states or have mutually exclusive predicates.

For efficiency the clique computation is performed using the algorithm based on the sequential greedy heuristics in [16].

V. EXPERIMENTAL RESULTS

A. *Choosing the best heuristic for post-schedule binding*

For experimenting with different heuristics for the post-schedule binding problem, we chose 20 industrial designs that well represent typical scheduling instances in our experience for the state-of-the-art high level synthesis industry.

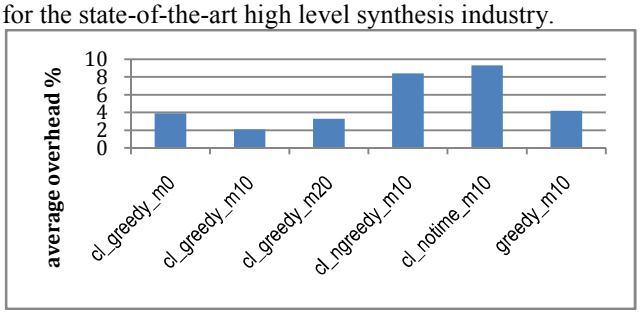

Figure 7. Comparison of binding heuristics

The results from running different heuristics (explained below) are presented in Figure 7. As timing correctness is a necessary part of all sharing conditions, all runs finish with positive slack. This allows us to evaluate the heuristics with the cell area after logic synthesis as the cost function. For each design, we chose the best implementation across all the heuristics and then for each heuristic we report the cost of the implementation relative to the best one in percentage (shown in Y-axis in Figure 7).

The heuristic parameters that we varied are the following:

1. **Timing margin.** Sharing a pair of resources has an associated timing penalty unless the arrival and required times for the resources are the same. Hence sharing is limited for CFG states with low positive slack. To increase sharing opportunities we performed scheduling for a clock period $T1$ which is smaller than the design clock period T. This provides a slack margin $T - T1$ which can be exploited during sharing to tolerate false paths penalties. We performed a parameter sweeping of this margin considering 0, 10% and 20% of the clock cycle (first 3 columns in Figure 7) and observed that the 10% margin provides the best results. The rest of the experiments are run with this margin.
2. **Greedy vs non greedy policy.** In a greedy policy once the best pair of resources r_i and r_j is identified in a clique C, C is exhaustively merged as long as it has compatible resources. The non-greedy policy on the other hand requires that after every merging all cliques are reevaluated, hence the merging process is not history dependent. This approach potentially suffers from creating too many merging seeds in the compatibility graph, and may actually be sub-optimal. Note that the chances of merging resources from different cliques become lower as

merging proceeds. The greedy policy performs best in practice (compare columns *cl_greedy_m10* and *cl_ngreedy_m10*)

3. **Cost of timing penalties during merging.** Using a high timing cost reduces merging of compatible resources but helps to increase slack margins during sharing. This is helpful, as shown by *cl_greedy_m10* vs *cl_notime_m10.*
4. **Cliques versus greedy choice of the best pair to merge.** Clique information guides the choice of the merge seed to a pair of resources that have more future potential for merging with other resources. On the other hand as compatibility relations are changed after the merging (due to timing penalties), the cliques also change, which makes the guidance less precise. However the comparison of *cl_greedy_m10* and *greedy_m10* confirms that cliques provide valuable input on the choice of resources to merge.

Note that the *cl_greedy_m10* heuristic gives best implementations in almost all cases (their average penalty versus the best for each design is about 2%). Hence this heuristic is used to compare against the on-the-fly binding.

B. On-the-fly vs. post-schedule binding

We applied the on-the-fly and the post-schedule binding approaches to a large number of industrial designs, and compared them with 52 designs for which their results differed more than 2% in their quality. The design characteristics are presented in Table 2.

TABLE 2. DESIGN CHARACTERISTICS

# operations			# resources			Frequency Mz		
min	max	avg	min	max	avg	min	max	avg
180	8400	1300	2	49	19	10	2000	200

The synthesis results are given in Table 2.

TABLE 2. COMPARING SHARING APPROACHES

Binding type	# wins	Aver.area penalty	Max penalty
On-the-fly	37	8%	12%
Post-schedule	15	12%	40%

Table 2 shows that on-the-fly binding provides superior quality over the post-schedule approach, confirming the results of [17], which used a much smaller set of academic benchmarks. It wins in 37 designs out of 52 and the average area reduction per winning case is more significant (12% versus 8%). The variance of area penalties is also smaller for this approach (12% versus 40%).

We compared the scheduling runtimes for both approaches using the 3 designs with the longest scheduling run times.

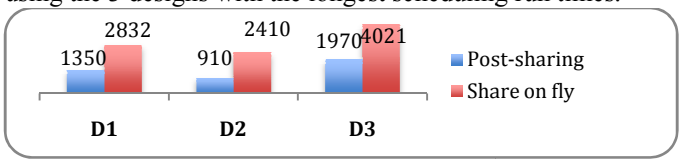

Figure 8. Performance comparison for binding approaches

The performance overhead for the on-the-fly binding is about 2-3x. With the run times well below 1 hour, this is acceptable in practice for the achievable quality gains.

VI. CONCLUSIONS

In this paper we argued and justified quantitatively, that scheduling and binding must be performed together in high-level synthesis, in order to properly account for the timing effects of false paths introduced by sharing. We evaluated on a broad set of over 100 industrial designs several possible heuristics to perform binding decisions after scheduling. Then we compared the best one with our efficient approach to simultaneous scheduling and binding. The results show that our proposed approach wins in 37 designs out of 52 for which the difference is significant.

The post-scheduling approach, however, is still better in 15 out of 52 designs. Hence the best option in general is to try with both strategies and use the winner for subsequent design steps. This is the approach currently employed in our tool.

REFERENCES

1 A. Sgouros, B. Leung, M. Warren, J. Ng. ”Microarchitecture Power Tradeoffs at the Electronic System Level”, CDNlive, 2008

2 G. De Micheli, *Synthesis and Optimization of Digital Circuits,* McGraw-Hill, 1994.

3 A. Kondratyev, M. Meyer, L. Lavagno, Y. Watanabe, “Realistic Performance-constrained Pipelining in High-level Synthesis,” Design, Automation & Test in Conference, pp. 1382-1387, 2011.

4 D. Knapp and M. Winslett. “A Prescriptive Formal Model for Data-Path Hardware.” IEEE Trans. On CAD, Vol. 11, No. 2 (Feb. 1992): 158-184.

5Coussy, P.; Gajski, D.D.; Meredith, M.; Takach, A.; , "An Introduction to High-Level Synthesis," *Design & Test of Computers, IEEE* , vol.26, no.4, pp.8-17, July-Aug. 2009

6 Toi, T.; Nakamura, N.; Kato, Y.; Awashima, T.; Wakabayashi, K.; Li Jing; , "High-Level Synthesis Challenges and Solutions for a Dynamically Reconfigurable Processor," *Computer-Aided Design,* pp.702-708, 2006

7 Sanguinetti, J.; Meredith, M.; Dart, S.; , "Transaction-accurate interface scheduling in high-level synthesis," *Electronic System Level Synthesis Conference (ESLsyn), 2012* , vol., no., pp.31-36, 2012

8 Guo, Y.; McCain, D.; , "Rapid prototyping and VLSI exploration for 3g/4G MIMO wireless systems using integrated catapult-c methodology," *WCNC 2006. IEEE* , vol.2, no., pp.958-963, 2006

9 Rim, M.; Mujumdar, A.; Jain, R.; de Leone, R.; , "Optimal and heuristic algorithms for solving the binding problem," *IEEE Transactions on VLSI*, vol.2, no.2, pp.211-225, 1994

10 Chen, D.; Cong, J.; Fan, Y.; Xu, J.; , "Optimality study of resource binding with multi-Vdds," *DAC, 2006.*, pp.580-585

11 Devadas, S.; Newton, A.R.; , "Algorithms for hardware allocation in data path synthesis," *IEEE Transactions on CAD*, vol.8, no.7, pp.768-781, 1989

12 Wilson, T.C.; Grewal, G.W.; Banerji, D.K.; , "An ILP solution for simultaneous scheduling, allocation, and binding in multiple block synthesis," *ICCD '94. Proceedings.,* pp.581-586, 10-12 Oct 1994

13 J. Cong, B. Liu, J. Xu: Coordinated resource optimization in behavioral synthesis. DATE 2010: 1267-1272

14 A. Sharma and R. Jain. “Estimating Architectural Resources and Performance for High-Level Synthesis Applications.” DAC Proc. 1993, pp 355- 360.

15 H. DeMan, J. Rabaey, P. Six, L. Claesen “Cathedarl II: Silicon Compiler for Digital Signal Processing.”, IEEE Design and Test 3, 1986, p 13-25.

16 E.Tomita, S. Mitsuma, H. Takahashi “Two algorithms for finding a near-maximum clique”, Tech Rep UEC TR, 1988

17 R. Bergamaschi. “The effects of false paths in high-level synthesis.” ICCAD, 1991, pp 80-83.