HAL
open science

Block Constraints Budgeting in Timing-Driven Hierarchical Flow

Olivier Omedes, Michel Robert, Mohamed Ramdani

To cite this version:

Olivier Omedes, Michel Robert, Mohamed Ramdani. Block Constraints Budgeting in Timing-Driven Hierarchical Flow. DCIS: Design of Circuits and Integrated Systems, Nov 2004, Bordeaux, France. pp.930-935. lirmm-00108941

HAL Id: lirmm-00108941

https://hal-lirmm.ccsd.cnrs.fr/lirmm-00108941v1

Submitted on 21 Jan 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Block constraints budgeting in timing-driven hierarchical flow

Olivier Omedes
Cadence Design Systems, Inc.
Omedes@cadence.com

Michel Robert
LIRMM UMR CNRS Montpellier II
Michel.Robert@lirmm.fr

Mohamed Ramdani
ESEO Angers
Mohamed.Ramdani@eseo.fr

***Abstract*— In this paper, we introduce a new block budgeting algorithm that speeds up timing closure in timing driven hierarchical flows. After a brief description of the addressed flow, block budgeting challenges are detailed. Then, we explain why existing budgeting approaches are not adapted to fulfil these challenges. A new block budgeting algorithm is proposed. In order to derive relevant block constraints, this algorithm analyzes the design flexibility. This Flexibility Aware Budgeting (FAB) approach is then compared to some previous ones. Experiments based on commercial EDA tools and real designs show up to 55 % reduction in hierarchical flow run time and lead to a good flow timing closure.**

I. Block budgeting challenges

Any physical synthesis solution has limitations on the size of circuits that can be handled in a single run. "Divide and Conquer" approaches have been introduced to overcome these limitations. In this kind of approach, large designs are sub-divided into smaller synthesizable sub-blocks.

Fig. 1 depicts a typical timing-driven hierarchical flow. This flow starts with RTL synthesis and technology mapping. Resulting netlist is assumed to be too big or too complex to meet the specified performance. Thus, the design is partitioned into sub-blocks. These blocks are floorplanned upon the chip die. In fact, the floorplanning step includes design physical partitioning, blocks placement and inter-block net global routing. Then, blocks are optimized. Let's notice that partitioning technique not only permits to implement large designs, but also allows optimizing blocks concurrently, that can be precious to decrease time to market – this represents one more reason to use the hierarchical flow. Optimized blocks are reassembled at the top-level and top optimization is run. If chip constraints are not met, the whole process can be repeated.

To implement sub-blocks, EDA tools need constraints. Blocks constraints are computed by the block budgeting step. The budgeting step derives blocks IO constraints from the chip constraints. In order to speed up the hierarchical flow timing closure, the budgeting process has, first to assign feasible blocks constraints, and, second, to ensure that if blocks implementation succeeded, all chip constraints will be met. These two conditions qualify budgets quality. Another key point of block budgeting step stays in its low resources consumption (budgeting that runs slower than flat optimization is not interesting).

If various delay budgeting approaches have been proposed to drive cell placement or net routing [1] [2] [3], there is only a few academic or industrial works dealing with IO constraints computing. [4] proposes an interresting RTL delay budgeting approach but reported that it needs an accurate area-time projection for each block; that is too much time consuming for our flow. [5] also presents an IO budgeting method for the timing-driven hierarchical flow. Unfortunately, the proposed budgeting is only applied evolving from a previous design implementation. Thus, it is usable for a second flow iteration but not to compute IO budgets at a so early stage of the design process. In fact, there are two main issues to resolve when computing IO budgets at this design stage. First, at this stage, design is not optimized yet (just area mapped upon library standard cells). Consequently, current logic structure is likely to largely evolve; this logic flexibility has to be taken into account. Second, IO budgets are not only timing constraints. Indeed, in the same way a cell delay depends on the driven load, an output required time assertion, for example, should depend on the connected load. Thus, block budgeting has to compute both timing and electrical constraints.

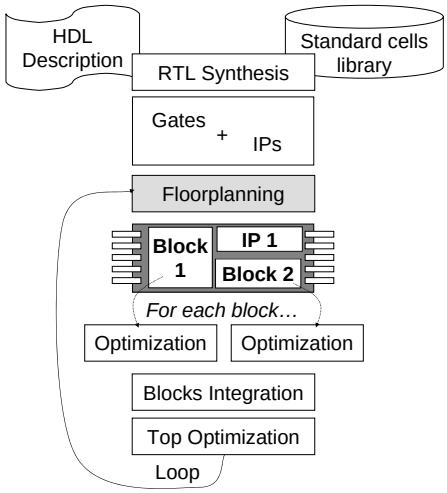

Fig. 1. Typical hierarchical flow

In this paper, we propose a block budgeting method that addresses these two issues. Logic flexibility awareness is obtained using a logical effort cell modeling [6] and

 ISBN 2-9522971-0-X

some simple restructuring or rebalancing algorithms inspired from [7] [8]. The logical effort theory is also used to better correlate timing constraints with block load specifications. The following section is devoted to this Flexibility Aware Budgeting (FAB) description.

II. Flexibility Aware Budgeting

A. Principles

In timing-driven hierarchical flow, at budgeting stage, design is likely to be only area mapped (or not homogeneously optimized). Cells are just implementing RTL functions and are surely not adapted to the load they will have to drive. Moreover, most of time, their logical structure is balanced and it is not taking into account that, for example, one input is more timing critical than another one. Consequently, at budgeting stage, current structure delays fall far short of the design achievable performance; this initial design implementation is highly flexible. If we run delay budgeting using these initial delays and loads, we are quite sure to produce irrelevant IO constraints. To avoid that, the first FAB step aims at computing more relevant delay and load values for all design components. “more relevant” stands for delay and load values that take into account the optimization possibilities: buffering, restructuring, ... Then, timing analysis is performed. This timing analysis gives information about the amount of slack available for each timing paths – slack is defined as the difference between required and arrival times. To ensure timing closure, these slacks (positive or negative) must be distributed between timing paths components. This slack distribution is also called delay budgeting since it consists to increase or decrease components delays – also called budgets. From an optimization point of view, it’s obvious that some components are harder to optimize and should, then, receive more budget. One way to favor these components is to use a weighted slack allocation algorithm. In FAB approach, we implement an IMP like algorithm [9] – weights are detailed in the following. As we said, in our approach, budgets are not only delays but also load specifications.
The very last FAB step computes the block IO constraints from the budgeted design.

B. Logical Effort Theory

In fact, the very first FAB step consists in standard cells characterization using the logical effort theory. The concept of logical effort has been proposed to model the load-independent gate delay:

$$d_{abs} = \tau.d = \tau(p+f) = \tau(p+g.h) \qquad (1)$$

This model divides the gate delay into two parts. First part is the load independent parasitic delay p (its major contribution is capacitance of transistors source / drain). Second part ($f = g.h$) is the effort delay. The logical effort g depends only on the gate topology and on its ability to produce output current. The electrical effort h (also called gain) is defined to be:

$$h = C_{out}/C_{in}$$

Where C_{out} is the connected load value and C_{in}, the gate input capacitance. Finally, τ is a technology dependent factor. All these values are easy to characterize for simple gates (inverter, nand, nor, xnor...). The logical effort theory also defines the branching effort b as the ratio of the total output capacitance to the capacitance which is on the path being analyzed. Given a string of N_g monofanout (b = 1) gates, the path delay is:

$$D = \sum_{N_g} p_i + g_i h_i = P + \sum_{N_g} g_i h_i$$

This string, or path, is also characterized by a path logical effort G and a path electrical effort H :

$$G = \prod g_i \qquad H = \prod h_i = \frac{C_{in-path}}{C_{out-path}}$$

The GH product is also called path effort and noted F. [6] explains that D is minimized when F is equally spread between stages :

$$\hat{f} = \sqrt[N_g]{F} \qquad \hat{D} = N_g \hat{f} + P \qquad (2)$$

Unfortunately, the number of stages N is very likely to evolve during optimization, and thus optimal effort $\hat{f}$ computation can be fairly ineffective. Indeed, on hard to optimize paths, optimization process will surely add some buffering components (inverters or buffers). Since, buffering components logical effort is defined as 1, we can write:

$$\begin{aligned} N &= N_g + N_{buf} \qquad \hat{f} = \sqrt[N]{F} \\ \hat{D} &= N.\hat{f} + P + (N - N_g) * p_{buf} \end{aligned}$$

N_{buf} and p_{buf} are respectively the number of added buffering components, and their unit parasitic delay. Differentiating with respect of N and setting the result to zero gives the following displaymath for $\hat{f}$ to minimize path delay:

$$\hat{f}(1 - ln(\hat{f})\ ) + p_{buf} = 0 \qquad (3)$$

(3) can be solved using numerical method (Newton-Raphson for example) and obtained $\hat{f}$ value permits to define an optimal electrical effort for each type of gate:

$$\hat{h} = \hat{f}/g$$

This result is fundamental for our approach since $\hat{h}$ will be used for delays and drive strengths initialization. Following table shows some experimental results for a 90 nm technology ($\tau = 7.2\ ps$).

Cell	g	$\hat{h}$	p	$\hat{d}$ (ps)
Inverter	1.01	3.71	1.16	4.90
Nand 2	1.41	2.62	2.41	6.10
Xnor 2	1.62	2.31	12.59	16.33

C. FAB Initialization process

FAB walks on an acyclic directed timing graph $G = (V, E)$ which represents the circuit timing structure. V denotes the set of vertices or nodes – nodes are circuit and cells IO. E denotes the set of edges or arcs which represent timing relations between nodes. Each edge e is annotated with a delay

value $d(e)$ which corresponds to its delay budget. Each node v is associated with a budgeted load $C_b(v)$ which corresponds to the maximum drivable load of this node. $\pi^-(v)$ and $\pi^+(v)$ respectively denote the set of v predecessors and successors.
In the timing graph, we distinguish some interesting structures that are:

- Cones: a cone is a set of mono-fanout logic and is completely identified by a set of cone inputs and one cone output. Cone $\mathcal{C}$ implements a boolean function $\mathcal{F}_\mathcal{C}$.
- Buffer trees: a buffer tree is a set of interconnections and buffering components. In following algorithms, a simple mono-fanout net is considered like a particular buffer tree.

FAB initialization process depends on a standard load value C_{ref} that can be either computed during the library characterization or user specified. In practice, its value is similar to the 1X inverter input capacitance. FAB initialization starts with the less flexible design instances which are IPs (Intellectual Property Blocs or small custom-designed blocks) and registers (sequential elements). IPs and registers timing arc delays are set to their propagation delay values when outputs are driving C_{ref}. Budgeted load C_b of IPs and registers outputs is set to C_{ref}. Budgeted load of IPs and registers inputs is set to their input capacitance value and is marked as fixed. Then, the flexibility aware initialization (FAI) step can start.

FAI is based upon two design transforms that are quite easy to anticipate. The first one consists in gates and cones fan-in ordering [7] [8].
Fig. 2 illustrates such a fan-in ordering on a small cone. Ordering starts when budgeted arrival times (BAT) and loads at cone inputs have been computed. It begins by remapping $\mathcal{F}_\mathcal{C}$ using only inverter, NAND and XNOR gates (cf. Fig. 2-b) – NOR gate is voluntary not used because of its low driving capability. Then, the structure is unbalanced in order to get a minimal BAT at the cone output (cf. Fig. 2-c).

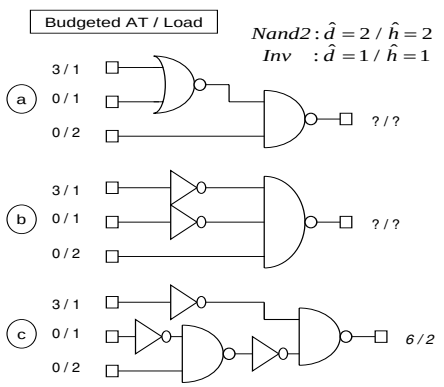

Fig. 2. $\mathcal{C}$ Restructuring

Cone arcs delays and output budgeted load are computed as explained in algorithm 1.

Algorithm 1 Quick $\mathcal{C}$ Restructuring Evaluation

Require: $\pi^-(SinkOfCone(\mathcal{C}))$ already treated.
Ensure: $sink \leftarrow SinkOfCone(\mathcal{C})$ treated
Remap $\mathcal{F}_\mathcal{C}$ (Fig. 2-b) and rebalance it (Fig. 2-c).
$d(v \rightarrow sink) \leftarrow \sum_{v \rightarrow sink} \hat{d}$
$C_b(sink) \leftarrow \min_{v \in \pi^-(\mathcal{C})} \left(C_b(v) * \prod_{v \rightarrow sink} \hat{h} \right)$

The second transform is about buffering. It consists in 2 phases. The first phase computes a balance buffer tree to drive the whole $\mathcal{B}$ buffer tree fanout (number of leaves). Budgets are computed for all buffer tree branches in the same time.
On the contrary, in the second phase, leaves are treated one after the other. This second phase aims at adding some budget to branches that are driving large fixed loads (such as IPs input capacitance). Some budgets are also added in order to do block interconnections buffering (interconnection capacitances are estimated from the initial floorplan).
Algorithm 2 and fig. 3 illustrate this second transform.

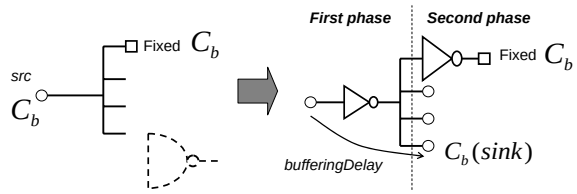

Fig. 3. $\mathcal{B}$ Buffering

These two transforms are combined and evaluated by traversing the graph in topological order (BFS) from primary inputs to primary outputs (this ensures that when treating node v, all $\pi^-(v)$ nodes have already been treated).
It's important to remark that these transform are not really applied (netlist is not edited) but just evaluated to get some budget values. This allows a significant computing gain. For this reason, these transforms are qualified of "virtual". FAB initialization produces a graph which is annotated with timing and load initial budgets.

D. FAB allocation

Timing analysis on this budgeted graph shows paths with both negative and positive slacks. Negative slack identifies hard to optimize paths whereas positive one characterizes more flexible paths where budgets (both d and C_b) can be increased. As explained in FAB principles section, path slack S_π is distributed among path arcs using the IMP algorithm. IMP is an iterative algorithm which adds a timing budget $\Delta d(e)$ at each iteration. $\Delta d(e)$ is computed as follow :

$$\Delta d(e) = w_e \frac{\min_{\pi \in \Pi_e} S_\pi}{\max_{\pi' \in \Pi_e} W_{\pi'}}$$

Algorithm 2 Quick $\mathcal{B}$ Buffering Evaluation

Require: $src \leftarrow Source(\mathcal{B})$ already treated.
Ensure: $\pi^+(src)$ treated.
  if $Fanout(\mathcal{B}) * C_{ref} \leq C_b(src)$ **then**
    $bufferingDelay \leftarrow 0$
    $tempC_b \leftarrow C_b(src)/Fanout(\mathcal{B})$
  else
    $effort \leftarrow Fanout(\mathcal{B}) * C_{ref}/C_b(src)$
    $N_b \leftarrow log_{\hat{h}_{buf}}(effort)$
    $bufferingDelay \leftarrow N_b(g_{buf} * \sqrt[N_b]{effort} + p_{buf})$ (cf. Eq. 2)
    $tempC_b \leftarrow C_{ref}$
  end if /* End of first phase */

  for each $v \in \pi^+(src)$ **do**
    $d(src \rightarrow v) \leftarrow bufferingDelay$
    if $isFixed(C_b(v))$ **then**
      $effort \leftarrow C_b(v)/tempC_b$
      if $effort > 1$ **then**
        $N_b \leftarrow log_{\hat{h}_{buf}}(effort)$
        $d(src \rightarrow v) \overset{+}{\leftarrow} N_b(g_{buf} * \sqrt[N_b]{effort} + p_{buf})$ (cf. Eq. 2)
      end if
    else
      $C_b(v) \leftarrow tempC_b$
    end if
  end for /* End of second phase */

Where w_e is the e arc weight, Π_e is the set of all paths traversing e and, S_π and W_π are respectively π path slack and total weight. A proof of IMP convergence to null slack on final budgeted timing graph is given in [2].
The main difficulty in slack allocation is to find out relevant weights for arcs. In FAB context, weights have to qualify arcs optimization hardness. Given the way initial arcs delays have been computed (using virtual restructuring and buffering), they give a good idea of optimization timing needs. Consequently, we use these initial delays as weights for the IMP allocation. IPs (or fixed blocks) and sequential elements receives a null weight since their flexibility is also nearly null; indeed IPs are impossible to remap and sequential elements are also fixed since we fixed the load they will have to drive.
If this slack allocation is quite typical, FAB not only allocates timing budgets, it also modifies node budgeted capacitance according to the added delay budget. Indeed, if we add delay to an arc, it's obvious to think that the load it can drive is also increased. So, if we differentiate (1) with respect to C_{out}, we obtain:

$$\frac{\partial d}{\partial C_{out}} = \frac{g}{C_{in}} \Rightarrow \Delta C_{out} = \frac{C_{in}}{g}\Delta d$$

Consequently, FAB allocation is ran as described by algorithm 3. At the end of this allocation, we get a full budgeted timing graph with null slack everywhere and each node is also constrained by a maximal drivable capacitance.

Algorithm 3 FAB allocation

Ensure: $\forall e \in E,\ Slack(e) = 0$
  /* Traversing G in topological order */
  repeat
    for each arc e such that $Slack(e) \neq 0$ **do**
      $\Delta d \leftarrow Slack(e)\frac{Weight(e)}{\max_{\pi \in \Pi_e} W_\pi}$
      $d(e) \overset{+}{\leftarrow} \Delta d$
      if $isNull(\Delta d)$ **then**
        $Weight(e) \leftarrow 0$
        $setFixed(C_b(sink(e)))$
      end if
      if $isNotFixed(C_b(sink(e)))$ **then**
        $C_b(sink(e)) \overset{+}{\leftarrow} \Delta d * C_b(source(e))/g_e$
      end if
    end for
  until $\forall e \in E,\ Slack(e) = 0$

E. Blocks constraints derivation

Using the full budgeted timing graph, deriving relevant blocks constraints becomes easy. BAT stands for Budgeted

Algorithm 4 FAB derivation

Require: $\forall e \in E,\ Slack(e) = 0$
  for each Blocks port p **do**
    if $isInputPort(p)$ **then**
      $set_arrival_time(BAT(p))$
      $set_max_input_capacitance(C_b(p))$
    else
      $set_required_time(BAT(p))$
      $set_output_load(C_b(p))$
    end if
  end for

Arrival Time and is computed by propagating budgeted delays. Algorithm 4 gives optimization constraints to facilitate block integration and top level timing closure. Indeed, for an input port, FAB ensures that if the *max_input_capacitance* is not exceeded, the port arrival time will be at least the one asserted. In the same way, block optimization have to implement block such that logic will be able to drive the *output_load* with respect to the *required_time* assertion.

III. Validation

A. Experimental Setup and Metrics

To qualify block IO budgets quality, we need to remember two important points. First, block optimization is a very chaotic step for which small constraints changes are likely to produce very different block implementations. Second, block constraints set is quite large and constraints are highly linked between them. Hence, if we want to compare budgeting algorithms results just looking at blocks implementations or constraints, we are likely to fail. In fact, the most significant approach to evaluate budgeting methods is to use them in

a typical timing-driven hierarchical flow. Then, metrics like optimization runtime, and slacks of blocks and top level can be used to appreciate budgeting results. Metrics we use are defined on fig. 4. Of course, this flow and these metrics favor rapid to close methods but this is what we are looking for.

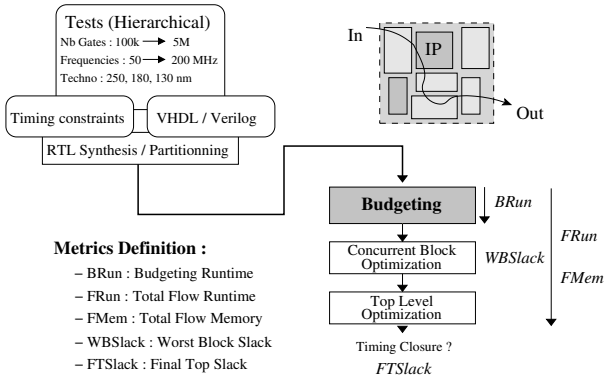

Fig. 4. Validation metrics

B. Some other used approaches

In the results section, we compare FAB approach with two other used block budgeting algorithms. The first one is called critical-path budgeting (CPB). Its principle is to treat block IO ports one after other and, to adjust current port arrival time according to delays of the critical path which crosses it. So, block IO port p budgeted arrival time is computed as follow:

$$BAT(p) = CurrentArrivalTime(p) + Slack(p)\frac{d_{left}}{d_{left} + d_{right}}$$

Where d_{left} and d_{right} are respectively delays of the left and right part of the most critical path traversing p. This budgeted arrival time is then used in algorithm 4 to assert block constraints. Capacitance assertions are computed using current load value for outputs load and driving cell load limit for inputs max capacitance.

Of course, due to completely not optimized design state, current values (arrival times, loads and driving cells) are surely irrelevant and thus, resulting IO budgets quality is doubtful. However, this approach has the advantage to be really fast (runtime wise).

The second approach, called IMP_T, is based upon IMP algorithm but doesn't include a FAI like step. For reasons explained above, it would be ineffective to use current design delays as weights. However, it could be a good idea to set weights using the logical effort theory.

Concerning loads assertions, a common way to avoid asserting huge values is to truncate them (limits can be specified by users). Fig. 5 shows truncation examples. This kind of truncation is part of IMP_T approach.

C. Results

Benchmarks specifications are given in table I. As we can see in the "Initial Slack" column, initial designs slack falls far short of designs constraints. Table I also summarizes flat flow results.

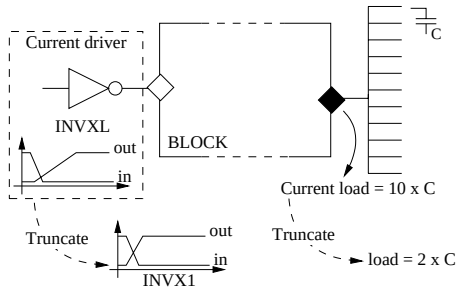

Fig. 5. Truncation of block input port driver and block output port load.

Table II details fig. 4 flow results when using either CPB or IMP_T budgeting algorithms to drive one pass of hierarchical optimization.

Runtime and memory wise, there is no doubt that the "divide and conquer" approach is faster and lighter than the flat one. Even if the timing closure is not obtained at the end of this first pass, the average runtime gain of about 66 % lets us with time to run a second iteration and to achieve this timing closure. The average memory need, which is about 35 % less than the one observed when using the flat flow, will not be increased in following iterations.

Now, let's compare the CPB and IMP_T timing results. The WBSlack column shows the worst block slack measured after blocks optimization. The - 0.94 ns WBSlack obtained when using the CPB algorithm shows well that this approach assigns some irrelevant block constraints. On the contrary, blocks constraints derived by IMP_T budgeting seem more relevant (or at less more feasible) since the average WBSlack is *only* - 0.27 ns – drivers and loads truncation is very helpful for that. Of course, having more feasible blocks constraints is not sufficient to ensure top level timing closure; these constraints have also to allow a feasible top level optimization. However, if we look at FTSlack columns for both CPB and IMP_T algorithms, we can conclude that IMP_T computes better blocks constraints than CPB. Indeed, at the end of this first hierarchical optimization pass, the top level slack is nearer from timing closure when using the IMP_T budgeting algorithm (IMP_T average top slack is about - 0.40 ns whereas CPB one is about - 0.94 ns). Hence, IMP_T budgeting leads to a better implementation to start a second iteration.

Now, let's look at the results obtained when using our new budgeting approach (Table III). Resources wise, the average gain stays similar to previous budgeting algorithms. Concerning the timing closure, load and drive constraints derivation – in parallel with timing ones – as done in FAB seems to be more relevant than simple truncation. Indeed, FAB WBSlacks stand out better than IMP_T ones (average gain is about 0.18 ns) and, as expected, this improves top-level optimization results (average gain is about 0.15 ns).

TABLE I

BENCHMARKS SPECIFICATIONS

Design Name	Number of blocks & IPs	Netlist Size Cell	Netlist Size Net	$\mathcal{G}(\mathcal{V},\mathcal{E})$ $\|\mathcal{E}\|$	$\mathcal{G}(\mathcal{V},\mathcal{E})$ $\|\mathcal{V}\|$	Initial Slack (ns)	Flat flow results CPU (s)	Flat flow results Mem (Mb)	Flat flow results Slack (ns)
TEST1	3 & 3	7k	10k	32k	19k	- 60.65	935	124	- 0.01
TEST2	3 & 16	82k	130k	330k	187k	- 60.35	5400	650	0.01
TEST3	4 & 6	102k	154k	278k	164k	- 135.45	4250	1234	0.00
TEST4	3 & 20	134k	175k	1054k	624k	- 89.59	27700	1180	0.00
TEST5	10 & 16	275k	355k	1656k	1065k	- 22.07	44100	1950	- 0.02

TABLE II

PREVIOUS BLOCK BUDGETING RESULTS – CPB AND IMP_T ALGORITHMS.

Design Name	CPB Hier. Flow Results FRun (s)	CPB FMem (Mb)	CPB WBSlack (ns)	CPB FTSlack (ns)	IMP_T Hier. Flow Results FRun (s)	IMP_T FMem (Mb)	IMP_T WBSlack (ns)	IMP_T FTSlack (ns)
TEST1	720	102	- 0.28	- 0.21	1250	101	- 0.08	- 0.06
TEST2	3800	456	- 1.87	- 0.63	3550	457	- 0.17	- 0.41
TEST3	2850	900	- 0.26	- 1.79	2900	850	- 0.08	- 0.17
TEST4	12000	635	- 1.16	- 1.16	9800	635	- 0.19	- 0.44
TEST5	8600	1123	- 1.14	- 1.13	10500	1280	- 0.87	- 0.91
$\overline{\Delta}$ vs. Flat Flow	- 66 %	- 37 %	- 0.94 ns	- 0.98 ns	- 66 %	- 35 %	- 0.27 ns	- 0.40 ns

TABLE III

NEW BLOCK BUDGETING RESULTS – FAB ALGORITHM.

Design Name	**FAB Hier. Flow Results** FRun (s)	FMem (Mb)	WBSlack (ns)	FTSlack (ns)
TEST1	800	107	0.00	- 0.04
TEST2	3300	456	- 0.08	- 0.33
TEST3	2900	930	- 0.03	- 0.16
TEST4	14000	690	- 0.12	- 0.25
TEST5	11700	1347	- 0.25	- 0.47
$\overline{\Delta}$	- 60 %	- 31 %	- 0.09 ns	- 0.25 ns

When we look closer at why timing closure was not obtained at the end of this first pass, we found out that some drive sizing and some buffering transforms was hardly impossible to anticipate since cells were not placed yet at budgeting step. However, for all testcases, the top-level timing closure has been obtained by just running a quick top-level optimization after the first flow iteration in order to fix these drive / buffering problems – this is not always true with CPB and IMP_T approaches for which deeper optimizations are needed (these results have not been reported in this paper). It's important to notice that, even with this additional top-level optimization, average runtime stays about 30 % less compared to flat flow runtime (memory gain is about 20 %).

IV. CONCLUSIONS

In this paper, we introduced a new budgeting approach that speeds up timing closure of a timing driven hierarchical flow. The initial budget allocation step of this new approach (FAI) allowed us to quickly identify paths that were "really" critical (following optimizations iterate on these paths). FAB IO constraints computing (BRun) is about 10 to 20 times slower than CPB and about the same than IMP_T, but this runtime stays lower than 10 % of FRun (e.g., about 15 minutes for TEST5). In addition, if we look at the results obtained for 5 commercial designs, it is obvious that taking more time to derive relevant IO constraints leads to a better timing performance predictability and to a faster hierarchical flow timing closure.

In the future, by integrating FAB approach to a physical prototyping tool, we should be able to take into account some more physical parameters such as blocks congestion and aspect ratios. This could also be helpful to compute more relevant weights or budgets for blocks interconnections.

REFERENCES

[1] M. Sarrafzadeh, D. A. Knol, and G. E. Téllez, "A delay budgeting algorithm ensuring maximum flexibility in placement," in *IEEE Trans. Computer-Aided Design*, vol. 16, Nov. 1997, pp. 1332–1341.

[2] H. Youssef, R.-B. Lin, and E. Shragowitz, "Bounds on net delays for VLSI circuits," in *IEEE Proc. Of ISCAS*, 1992, pp. 815–824.

[3] J. Frankle, "Iterative and adaptive slack allocation for performance-driven layout," in *IEEE/ACM Proc. of DAC*, June 1992, pp. 536–542.

[4] C.-C. Kuo and A. C.-H. Wu, "Delay budgeting for a timing-closure-driven design method," in *IEEE Proc. of ICCAD*, 2000.

[5] K. Shi and G. Godwin, "Hybrid hierarchical timing closure methodology for a high performance and low power dsp," in *IEEE/ACM Proc. of DAC*, 2005, pp. 850–855.

[6] I. Sutherland, B. Sproull, and D. Harris, *Logical Effort: Designing Fast CMOS Circuits*. MORGAN KAUFMANN PUBLISHERS, 1999.

[7] P. G. Paulin and F. J. Poirot, "Logic decomposition algorithms for the timing optimization of multi-level logic," in *IEEE Proc. of ICCD*, 1989, pp. 329–333.

[8] K. J. Singh, A. R. Wang, R. K. Brayton, and A. Saugiovanni-Vincentelli, "Timing optimisation of combinational logic," in *IEEE Proc. of ICCAD*, 1988, pp. 282–285.

[9] H. Youssef and E. Shragowitz, "Timing constraints for correct performances," in *IEEE Proc. of ICCAD*, 1990, pp. 24–27.