Zephyr: A Static Timing Analyzer Integrated in a Trans-hierarchical Refinement Design Flow

Christophe Alexandre, Marek Sroka, Hugo Clément, and Christian Masson

Université Paris VI, Laboratoire LIP6, 4 Place Jussieu, 75005 Paris, France

Abstract. The evolution of silicon technologies has fundamentally changed the physical design EDA flow, which now has to go through a progressive refinement process where interconnections evolve seamlessly from logic to final detailed routing. Furthermore the level of integration reached makes mandatory the use of hierarchical enabled design methodologies. In this paper, we present Zephyr: an Elmore Delay Static Timing Analysis engine tightly integrated in the open academic Coriolis EDA physical design platform on which tools act as algorithmic engines operating on an integrated C++ database around which they consistently interact and collaborate. Coriolis provides high level C++ and Python APIs and a unified and consistent hierarchical VLSI data model through all the design steps from logic down to final layout. We discuss here more specifically the integration issues and concepts used to support timing analysis through the progressive refinement of hierarchical designs.

1 Introduction

The evolution towards nanometer silicon technologies has deeply enforced the role of interconnections in the VLSI design flow.

This has introduced fundamental changes in the physical design flow, which now has to go through a progressive refinement process in which CAD tools incrementally update an integrated central database representing the current state of the design. Within this process, interconnections evolve seamlessly from the logic view to the final detailed routing view through intermediate and more or less precise global routing steps.

Therefore it is increasingly important to have at one's disposal a flexible interconnect timing analysis tool, which is able to adapt its analysis to the changing level of precision of the routing description. Moreover, computing wire delay with an acceptable degree of precision requires to take into account the wiring topology and the distribution of resistance and capacity. This leads to the conclusion that the structure of interconnects is an information that must be continuously accessible to the timing analysis tool.

An other major problem comes from the level of integration now reached, which makes mandatory the use of hierarchical enabled design methodologies in order to handle large hierarchical netlists mixing hard IP blocks (pre-designed RAM, ROM, CPU Cores), soft IP blocks and glue control logic. Within the physical design process, this structural hierarchy may be partly lost (completely

J. Vounckx, N. Azemard, and P. Maurine (Eds.): PATMOS 2006, LNCS 4148, pp. 319–328, 2006.

lost in most place and route CAD tools which operate on the flattened netlist). However the capability to handle the physical design hierarchy while maintaining it correlated with the netlist hierarchical description makes the functional equivalence checking (through LVS, simulation or formal proof) as well as the timing and signal integrity analysis results, much easier to understand and manage by the designer.

Zephyr the Static Timing Analysis engine that we describe is designed to deal with these issues:

- It is a central component of an open academic EDA research platform (Coriolis) presented in Section 2.
- It relies on a trans-hierarchical occurrence model, a feature at the heart of Coriolis: the Hurricane database, subject of section 3.
- It is associated with an interconnect analysis engine, discussed in section 4, which estimates RC values on composite interconnects, along the refinement process.

Section 5 surveys major components and characteristics of Zephyr, and section 6 ends with the conclusion.

2 The Coriolis Platform

In the past ten years, the CAD community has seen the emergence of several industrial platforms. They have the common characteristic of managing a centralized design database working as a framework on which different tools can share and refine continuously the design data. By merging the logical and physical aspects of the circuits, these databases avoid inconsistencies and losses of informations due to the continuous changes in level of representation.

Cadence and Synopsys offer respectively OpenAccess and Milkyway. The main difference between them lies in their diffusion policy. While Milkyway is partially opened to Synopsys clients through the MAP-In program, Cadence is at the origin of the OpenAccess Coalition, which provides OpenAccess as a open source EDA database to registered users. Conversely, Magma's platform is entirely built around a proprietary central database whose details are not publicly available.

At the same time, a very active academic research community, mainly addressing specific algorithmic steps of the flow, shares tool implementations and common benchmarks in an open-source approach, as exemplified by the GSRC "*bookshelf*" repository [1]. However, those tools communicate only through interchange formats and the development of an integrated Physical Synthesis environment in academia was, up to now, considered as impossible [2].

Nevertheless, academic projects have recently emerged. On the one hand, the OpenAccessGear [3]: an open source development environment for physical design built on top of the OpenAccess database [4], which includes a user interface, a wrapper to the academic standard-cell placer CAPO, a set of benchmarks and a static timing analyser: OA Gear Timer.

On the other hand, Coriolis [5]: an ongoing project of the LIP6 laboratory, which provides the academic community with an open source platform

(downloads under the GPL license are available through the project homepage at [6]). Coriolis is a back-end platform on which tools act as algorithmic engines operating on an integrated C++ database (named Hurricane) around which they consistently interact and collaborate. Coriolis provides a free CAD teaching and research open environment (both on design flows and algorithms) by offering both a set of core functionalities (such as a lef/def interface, a Python extension language and a graphical user interface) and a progressively enhanced suite of open-source CAD tools supporting academic VLSI design projects.

The global intent of the Coriolis project is to develop a fully integrated Physical Synthesis environment supporting progressive refinement design flows. Currently are available: a standard cell global and detailed placer, a global router and a timing analysis module, subject of this paper.

3 The Hurricane Database

Hurricane is a lightweight C++ object oriented database and programming platform which provides a unified and consistent modeling of hierarchical VLSI layouts through all the design steps from logic description down to detailed layout. It is outside the scope of this paper to detail Hurricane and interested readers will find documentation along with Coriolis. Here, we will summarize some concepts, with a higher focus on Hurricane hierarchy representation.

Hurricane provides a powerful object-oriented API for fast access, incremental update and consistent management of all the design views which fully relieves the application programmer from memory management issues.

It models in a unified view both the netlist and the routing (global or detailed) through "*hooking*" mechanisms which allows the seamless forward or backward transformation of a net-list into a global routing or a detailed layout (or a mix of those states), ensuring built-in consistency. For instance, segments know the contacts (or ports) on which they are anchored and the contacts (or ports) know their incident segments. Deleting some detail routing elements automatically links disconnected items by "*rubber*" fly lines, which reflects what needs to be reconnected.

Hurricane data structure embeds high performance 2D region query methods and provides a built-in high speed graphical display engine of the current state of the design, very useful for designing and debugging layout synthesis algorithms.

It provides extensibility mechanisms, notably through properties and relations which can be attached to any kind of database object (including to *occurrences*, see below).

It offers a rich (and extensible) set of powerful and generic query objects named *Collections*. *Collections* are not containers but "*set descriptors*" which provide an associated *Locator* for tracing through the corresponding set of elements. They can hide a fairly complex algorithmic trace process, visiting huge sets, but with very low memory foot print. Furthermore, *Filter* objects can be applied to a *Collection* in order to visit only the subset matching a predicate. *Collections* are very handy and flexible programming paradigms.

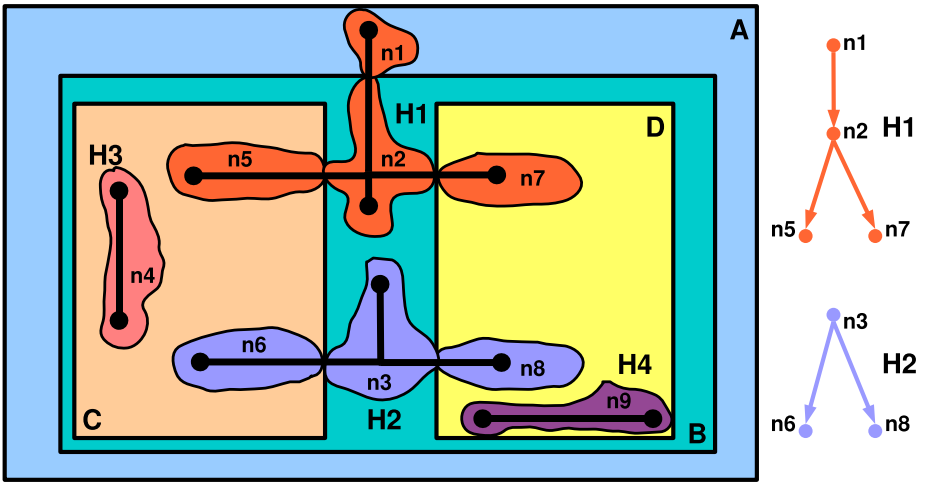

Fig. 1. Hypernet structure of interconnects

All modern design databases use *folded* hierarchy, in that every instance of a given cell points to the same master cell. This makes more manageable the representation of complex designs and the memory consumption. It is also easier to fix a problem in a cell and have it reflected everywhere instantly. However, there is no place in a *folded* database to attach data that would be specific to the context in which cells are instantiated.

To deal with this issue, Hurricane also represents hierarchical layout as a *folded* memory data model, but provides a **virtually unfolded** view to the tools tracing, annotating or displaying its content. For that purpose it manages the concept of *occurrences* which can refer any logical or physical item anywhere within the **virtually unfolded** design hierarchy. *Occurrences* are very light, volatile pointer pairs <*instantiation-path*, *item-in-the-model-cell*>, where the instantiation-path is also a compact shared object which implicitly knows the top cell, the instances composing the path and the cell master model of the lowest instance (an *Occurrence* with a NULL path refers an item of the current cell).

Two *occurrences* objects are identical if they refer to the same object of the unfolded hierarchy. If a property is attached to the first one, it becomes visible from the second one. Of course those properties are securely stored on an automatically managed hidden object which exists only if at least one property is attached to the occurrence it represents. *Occurrences* may be relative to the top cell or to some sub-cell, this allows to attach partially context dependent properties at the right level in the layout hierarchy.

*Occurrences*provide elegant ways to design algorithms for visiting, extracting and annotating hierarchical designs without the need to partially unfold the hierarchy with complex cache techniques. However this approach to annotate a *virtually unfolded* design has a memory cost which is acceptable only when the ratio of *occurrences* with attached properties (at each processing step) versus the total

number of potential *occurrences* is low. This holds for most well designed layout algorithms, and more the design is hierarchic more this approach is efficient.

Combining the previously defined concepts, complex queries like: "visit all occurrences of segments on a given layer in a given area", "find all parallel adjacent segments on the same layer which may cross-talk with the segments of a net", or "visit all layout element occurrences electrically connected to a given layout occurrence" are written in few lines of code. The graphic highlighting of the visited occurrences is simply done by attaching them the "Select" property!

Thanks to the occurrence mechanism it is easy to deal with the trans-hierarchical structure of interconnect during the process of Physical Synthesis. Indeed, the *folded* design hierarchy breaks up each interconnection into a tree of net occurrences, that we name an **hypernet**. The root of this tree is the net occurrence at the highest hierarchical level, and it canonically represents the **hypernet** interconnect. This is illustrated in the figure 1.

4 Lightning: The Hypernet Abstraction Module

Hypernet based collections provide the capability to visit the layout elements of a virtually flattened interconnect, however this tracing process doesn't follow the tree topology of the interconnect, and is not appropriate to build a RC tree.

It is the purpose of the Lightning module to provide such a canonical method to trace routing elements of an hypernet, and offer an abstraction layer upon the real state of the design which can be either globally or detail routed, or a combination of both. It takes detailed routing data when available, while for global routing it merely follows Steiner tree topology and distances. This allows to navigate homogeneously inside hierarchical designs where hypernets may overlap globally routed glue and prerouted blocks.

Lightning builds on-the-fly a temporary lightweight data structure representing the current routing state. This is done by reading each layout element occurrence of the hypernet and converting them into nodes and edges. Nodes are factored by their coordinates and layer, they may be either bifurcation or terminals points (primary I/O or I/O port occurrences of leaf cells). Edges represent wires with a given layer and width, or layer changes through a via. Nodes and segments of global routes have no layer and width specified. Once this data model is built, the trace proceeds from the occurrence of a driver port that becomes the root of the exploration tree.

The figure 2 illustrates this trace process on a composite interconnection.

During this depth first trace process, each edge, node, branch separation at a bifurcation point or terminal which is reached triggers the call of a visitor's method: a callback which provides information about position, distance, width and layer (and also the occurrence of terminal points). A "*visitor*" is a functional object that allows the application program to specify its own operations for each step within the trace process. Notice that for logical nets the routing graph should be a tree, however if loops exist, their re-convergence points are reported (in order to manage meshed nets like clocks).

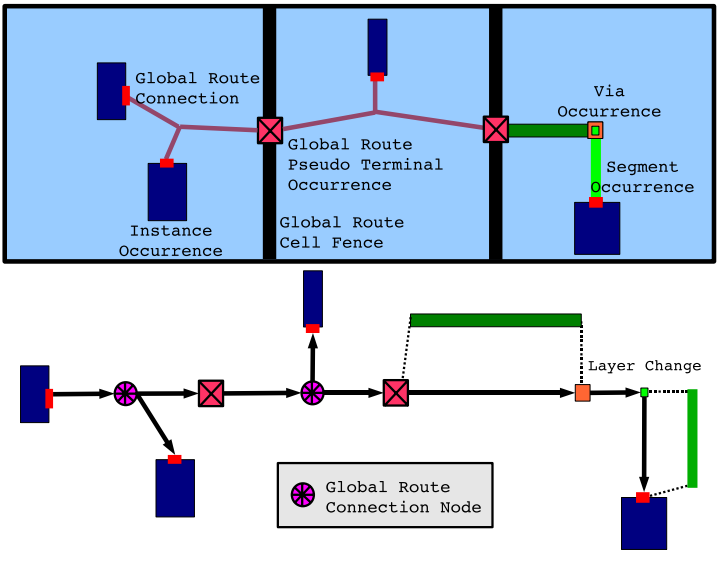

Fig. 2. Lightning trace process of a composite interconnection

5 Zephyr: The Static Timing Analyzer

Zephyr is a flexible static timing analysis engine central to the Coriolis platform and constantly accessible by optimization engines. In the current version, Zephyr is mainly intended to drive global placement and global routing engines. Therefore, the level of precision of its three internal modules (RC tree evaluator, delay calculator and static timing analyser) is adapted to this task. In the future, we consider improving progressively and jointly all Zephyr modules, in order to be able to run post Physical Synthesis precise timing analysis.
Zephyr inputs are:

- A hierarchical mixed size block and standard-cell design
- Technology timing characteristics of cell libraries in a subset of the Synopsys "*liberty*" format (*.lib*), from which we extract fixed cell delays and cell output resistances.
- User timing constraints can be provided by a subset of the *.sdc* standard format.

We will now detail Zephyr modules.

5.1 The RC Evaluator Module

The RC Evaluator builds a RC Forest (RCF), which associates to each hypernet a RC Tree object (RCT) whose root is attached to the hypernet driver port occurrence and whose leaf nodes are attached to the receiver ports occurrences, intermediate nodes being attached to divergence points of the hypernet Steiner tree. For multi driver hypernets, RC Trees are created for each driver.

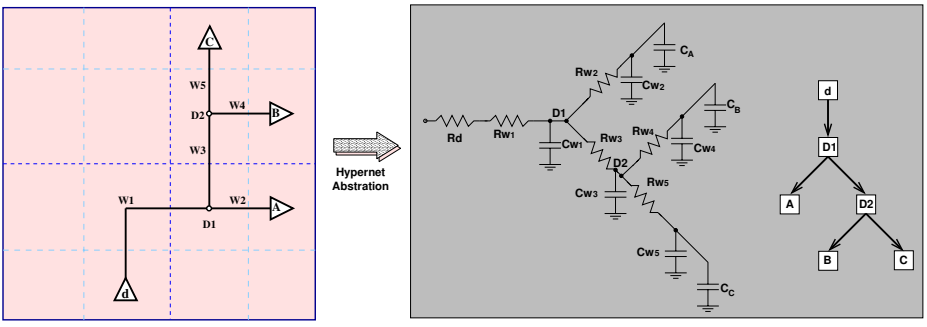

Fig. 3. Construction of a RC Tree

The RC Tree is built by an exploration of the hypernet through Lightning, with a "*visitor*" functional object which estimates the RC values according to the level of accuracy of the routing and the technological parameters. Currently it relies on simple estimates based on distance, width and layer, in the future it might proceed to more complex geometric queries to identify interfering wires in order to estimate capacitances more accurately, or even subcontract this task to an external RC Parasitics Extractor.

5.2 The Elmore RC Delay Calculator

The RC Tree object is optimised for Elmore delay computation. Each node contains the lumped capacitance of the node sub-tree as well as local resistance of the incoming wire. The delay is computed simply by back-tracking from a RCT leaf up to the root, applying either the Scaled Elmore Delay or the Fitted Elmore Delay formula [7].

5.3 The Static Timing Analyser Core

The Zephyr Static Timing Analysis engine, like OA Gear Timer, models the timing of the circuit as a directed acyclic graph, the Generalized Causality Graph (or GCG), which remains invariant (unless the logic is modified). In the GCG, two nodes (up and down transitions) are attached to each primary inputs/outputs or to I/O port occurrences of leaf cells, the edges representing the delays between the nodes they connect. There are two kinds of edges: those for gate internal delays (as provided by the pre-characterized cell-library - unless for registers) and those representing interconnect delays (provided by the RCT delay calculator).

Therefore the GCG is anchored on the virtually flattened view of a hierarchical design.

OA Gear Timer (according to [3]) relies on a simple wire delay model either based on the half-perimeter bounding-box of interconnections or brought through a callback function mechanism which only permits to define wire delays and capacitive loads on whole nets. Instead, Zephyr relies on its *generic* RC Tree

evaluator and delay calculator which allows to differentiate the loads and delays of each driver-receiver pair of an hypernet.

5.4 Edge and Net Criticality Calculator

The issue of net weighting for timing-driven placement has been thoroughly studied. Tim Kong [8] has proposed a very efficient and accurate algorithm for computing an "*all path going through*" criticality for all edges of the GCG (from which net weighting can be derived as the maximum criticality of its GCG edges), that we have integrated into Zephyr.

5.5 The Critical Path Generator

The Timing Analysis engine is able to determine any (reasonable) amount of most critical paths in the design.

Our algorithm works by progressively expanding shared partial longest paths. They are kept ordered by decreasing slack of the most critical path going through them. Path completion lasts until the requested number of critical paths is reached. This algorithm is quite effective and requires at most n operations by computed path, n being the edge count of the longest path in the GCG.

To display the critical path list, the tool provides a graphical user interface composed of two windows: a critical paths list window displaying the critical paths and a critical path window detailing the different components of a given path, which can be highlighted on the layout (a screenshot showing Zephyr timing analysis on a placed and globally routed block appears on figure 4).

5.6 Incremental Update

Both the RC Tree and GCG nodes are anchored into the the virtually flattened design as Hurricane properties on the terminal occurrences, this allows to cross-reference them as well as notify them when a layout change occurs.

On one hand, a GCG edge can access the terminal occurrences attached to its two ending nodes, and then, to their associated RC Tree nodes for computing its delay.

On the other hand, when a layout element occurrence is modified (by route or place refinement), this is notified to the net occurrence and from then to the hypernet terminal occurrences, which in turn will invalidate the corresponding RC Tree and GCG edges. Those edges will then propagate invalidate flags in their fan-in and fan-out cones. When the next timing analysis occurs, a delay request on an invalidated GCG edge will transfer the request to its associated RC Tree, which, if it has been invalidated will initiate its re-evaluation.

5.7 Validation and Experimentation

The Elmore RC Delay Calculator was validated by converting the nets of a circuit into Spice models and comparing its results to Spice simulation. Comparisons

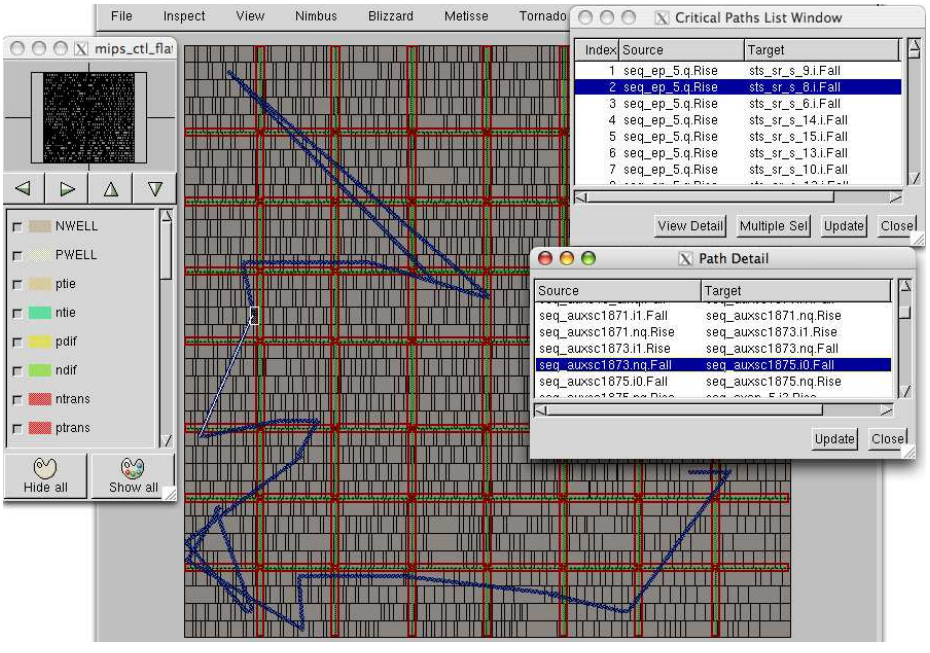

Fig. 4. Zephyr screenshot: critical paths list and path detail windows and a highlighted critical path

showed that Zephyr error margins do not exceed 10 percents, while for most nets, the error is below 5 percents (using Fitted Elmore Delay should improve those figures). Indeed the true source of errors is the way we approximate the RC Trees, which will require calibration through benchmarking on representative circuits in each technology used.

The critical paths, computed by the GCG, were checked against the ones given by the industrial tool TAS from Avertec [9]. The longest paths list was roughly the same in both tools, so we can rely on Zephyr to drive place and route flows.

Working with optimization engines, Zephyr has proven to be a non-critical part in terms of memory consumption and run time. For instance, we measured execution time of Zephyr on a fully placed 38k cells design. It was first globally routed in 157 seconds, the timing analysis was achieved in 8 seconds and the first 10k critical paths were computed in about one second. We are currently evaluating the speed improvement when working in incremental mode.

6 Conclusion

Zephyr is fully integrated in the Coriolis platform and is being experimented in a top-down progressive refinement flow for standard cell timing driven placement. This flow proceeds by a succession of interleaved phases of quadri-partitioning, global routing and static timing analysis which provides net criticality evaluation

and feedback for the next refinement loop [5]. It is also used as a stand alone tool with a netlist and layout capture language, providing coarse timing estimations, early in the design process.

This demonstrates the capability of Zephyr to be seamlessly integrated in various design flows. Zephyr is already accessible to the open source community, under the GPL licence.

References

1. http://www.gigascale.org/bookshelf/slots/
2. Adya, S.N., et al.: Benchmarking for large-scale placement and beyond. In: Proceedings of the 2003 international symposium on Physical design, ACM Press (2003) 95–103
3. Xiu, Z., Papa, D.A., Chong, P., Albrecht, C., Kuehlmann, A., Rutenbar, R.A., Markov, I.L.: Early research experience with openaccess gear: an open source development environment for physical design. In: ISPD '05: Proceedings of the 2005 international symposium on physical design, ACM Press (2005) 94–100
4. http://openeda.si2.org
5. Alexandre, C., Clement, H., Chaput, J.P., Sroka, M., Masson, C., Escassut, R.: Tsunami: An integrated timing-driven place and route research platform. In: DATE '05: Proceedings of the conference on Design, Automation and Test in Europe, IEEE Computer Society (2005) 920–921
6. http://www-asim.lip6.fr/recherche/coriolis
7. Abou-Seido, A.I., Nowak, B., Chu, C.: Fitted elmore delay: a simple and accurate interconnect delay model. IEEE Trans. Very Large Scale Integr. Syst. **12**(7) (2004) 691–696
8. Kong, T.T.: A novel net weighting algorithm for timing-driven placement. In: Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design, ACM Press (2002) 172–176
9. http://www.avertec.com