Timing Analysis and Optimization of a High-Performance CMOS Processor Chipset

Uwe Fassnacht
IBM Entwicklung GmbH Boeblingen
Schoenaicher Str. 220
71032 Boeblingen, Germany
fass@de.ibm.com

Juergen Schietke
Research Institute for Discrete Mathematics
University of Bonn, Nassestr. 2
53113 Bonn, Germany
schietke@or.uni-bonn.de

Abstract

We describe the timing analysis and optimization methodology used for the chipset inside the IBM S/390 Parallel Enterprise Server - Generation 3. After an introduction to the concepts of static timing analysis, we describe the timing-modeling for the gates and interconnects, explain the optimization schemes and present obtained results.

1. Overview

After introducing the chipset, the used library and differentiating static timing analysis from simulation in section 2, we go over the basic concepts of static timing analysis in section 3. In section 4, we describe our clocking structure and then, in sections 5 and 6, we explain how circuits and interconnects are modeled for the timing-tool. Thus sections 2 through 6 set the stage for the introduction of the optimization scheme, which is presented in sections 7 and 8. In section 9 we share measured results, section 10 gives an outlook on our current work, and the paper wraps up with conclusions in section 11.

2. Introduction and Motivation

The described chipset consists of 6 chips and is based on standard-cell elements from a 0.25 um L_{eff} library. These were combined with some custom-designed blocks in order to achieve an overall operating frequency of 155MHz.

All analysis and optimization was done on flat netlists, ranging up to 209000 objects for timing, and up to 17.9 million transistors per chip.

When working with such large designs, a timing-simulator based methodology becomes extremely complex and time consuming. Instead we de-coupled functional analysis, which was done by simulation, from the timing verification process.

Timing verification was based on a static timing methodology, which has several advantages over conventional simulation. Due to its exhaustive nature, static timing analysis guarantees 100% coverage of paths in the designs. Calculating every possible path raises the problem of non-functional or “false” paths. These have to be flagged through knowledge-based input from a human, a tedious process, which is not needed for simulation.

The closed-form nature of a static timing approach enables the analysis of paths in early stages of the design. This offers the advantage of being able to detect and correct timing-problems before the logic-designer

has fully completed the function. And, as static timing analysis does not require detailed knowledge of the logic, it can easily be used during the place & route process by engineers who are not intimately familiar with the logic.

We chose to use EinsTimer from IBM Microelectronics as our timing-analyzer. It offered all the features we were looking for, was built to run within our environment and could easily interface to our design databases.

Once set up, we used it to point out cycle-limiting paths in the design as it evolved through the front- and back-end methodology. During this process we continuously refined our interconnect model in order to take advantage of more detailed information as it became available from place & route. Finally, we used static timing analysis as the final sign-off criterion for release of the mask-data to fabrication.

3. Concepts and Terminology

Basically, static timing analysis propagates signals through logic-gates and interconnects, adding up delays along the paths. EinsTimer computes arrival-times (ATs) for all timing-points in the design for both rising and falling edges. It can identify both slow and fast paths by propagating both latest and earliest ATs for the same signal-edge. Along with their ATs, the tool also propagates the changing slew-rate of the signals, as they propagate through the timing-graph.

These calculated ATs are then checked against required arrival-times (RATs), which have been propagated back upstream through the logic. The checks (e.g. Setup and Hold) and their guard-times are coded into the timing models of the gates.

The difference between RAT and AT, which we refer to as “slack”, is the amount of excess timing resource in a given path. A positive slack on all tests in the design signifies that all tests have been met and the design will work at the asserted conditions and cycle-time.

4. Clocking

The chipset is clocked fully synchronously and based on strict testability-rules, e.g. LSSD [1].

Our design-approach splits a PLL-regulated pulse into several varieties of launching and capturing clocks (Figure 1) which are then distributed to the latches.

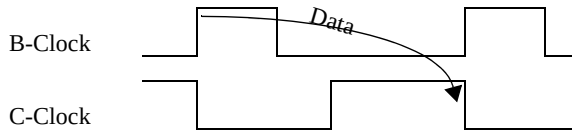

FIGURE 1. Clocking Scheme

On almost all paths in the designs, data is launched from a latch with the rising edge of the B-Clock, propagated through combinatorial logic and captured at the receiving latch with the falling edge of the C-Clock.

In order to obtain minimal skew between clock arrival-times at different latches, the clocktrees were symmetrically generated and optimized after placement had been completed.

Manufacturing tolerances aside, and assuming a perfectly symmetrical clocktree, the non-overlapping clocking-scheme would prevent short-path problems. In order to take realistic manufacturing tolerances into account, a SPICE-like circuit-simulator was used to analyze the clocktrees under different process and environmental parameters. These calculated margins were then added as guardbands to the clock ATs at the latches.

5. Modeling

The propagation delays through the blocks, coded in the timing-models of the gates, were based on a suite of circuit-simulation runs at various environmental conditions. Results from these runs were abstracted into coefficients for the equation-based models. These equations for gate-delay and slew-propagation are nonlinear and take output loading, input slew, temperature, VDD and process variations (sampled as worst-case, nominal and best-case) into account.

The interconnects were modelled in several ways, depending on the stage in the methodology. During synthesis, capacitance and RC-delay of the interconnects were estimated based on fanout. The formulas for these estimations were derived by analyzing earlier versions of the design or similar designs, that have been placed & routed in the past.

After receiving the netlist from the synthesis-process, we asserted all interconnect-lengths to be zero instead of estimating them based on fanout. This provided a lower bound on the cycletime, which the place & route process could strive towards. As we are employing techniques for drive-strength optimization and buffer-insertion, this is not an absolute lower bound, but still an interesting metric to compare different versions of an evolving netlist.

During the timing-driven placement phase, estimations were replaced by 2D-rectilinear Steiner-trees generated from the current placement.

Following layout, we extracted the actual wirelengths from the design using an abstracted stick-figure-representation of the wires. This model, which was characterized earlier from generic shapes, is provided with each IBM Microelectronics library. Through this process, we reduced turn-around-time and data volume significantly compared to an extraction from actual shapes. The resistance- and capacitance-per-wirelength model assumed an average channel usage of 80%, in order to conservatively model the overlap and fringing capacitance due to adjacent routing.

The RC-delays caused by the interconnects were calculated based on the Elmore delay model [2].

As the clocktree was generated and inserted after placement (in order to minimize skew), the ATs of the clock-signals at the latches had to be asserted based on an “ideal” clocktree during the synthesis and placement step. We then replaced these overrides with calculated ATs as soon as the clocktree was placed, routed and extracted.

6. Assertions

EinsTimer is controlled by assertion-files, which serve several purposes.

- They establish clockphases and their relationships, so that the tool can automatically adjust clock edges, e.g. by one cycle.
- They define chip boundaries with asserted ATs and slopes at the inputs, as well as RATs and capacitive loading at the outputs.
- They contain false-path declarations and individual signals adjustments, defined by the designer, as the tool understands very little about the logical function of gates.

7. Optimization

Due to the large design-complexities and associated turn-around-times of the tools, it becomes unfeasible to separate the placement and delay-optimization phases from each other. For this reason, we have chosen to use an optimization-program which was developed and implemented at the University of Bonn. It is tightly integrated into their suite of place & route tools [1] and its results correlate very well with EinsTimer.

This coupling makes it far easier to make correct decisions concerning the different optimization options (e.g. drive-strength change, buffer-insertion, net-weight). Especially, if it is done in between the steps of the placement-phase [3].

At these points a placement program can be led in the right direction by first optimizing the logic according to the current placement and then increasing the weight on the remaining critical paths. This not only achieves the fastest physical implementation of the logic, but also minimizes the area and power consumption. And furthermore, design rule checks, such as maximal capacitance loads to be driven and maximal input slopes, must be honored by the resulting solution.

For these different tasks a common optimization scheme has been developed. It interacts with place & route tools in order to get as accurate information about the interconnects as possible. Figures 2 and 3 show the optimization scheme, which is embedded in the higher-level environment shown in figure 4.

In contrast to other optimization methods (see [4] for an overview) this algorithm has two advantages:

- The solution found in each iterative step is almost always optimal.
- The chosen operations to improve the network are independent of each other. This ensures that their application will result in the computed outcome.

Other methods, like [5], take the neighborhood of an operation into account to compute its outcome accurately. However, during the selection of the operations which will ultimately be applied, they discard this neighborhood information. This results in a situation where the subnetwork used to compute the outcome of the operation is different from the one where the operation is actually applied.

As delay-minimization is the focus of the optimization, we present it as an example of the overall scheme.

8. Delay Optimization

The basis of the optimization scheme is a directed graph which is derived from the logic as follows:

- each node in the graph represents a pin in the design

- each arc in the graph is either a source to sink connection of a net or a timing dependency between an input pin and an output pin of a block

- the direction of an arc is defined by the “natural” signal flow. Bidirectional pins are represented by two different nodes, one for the input direction and the other for the output direction.

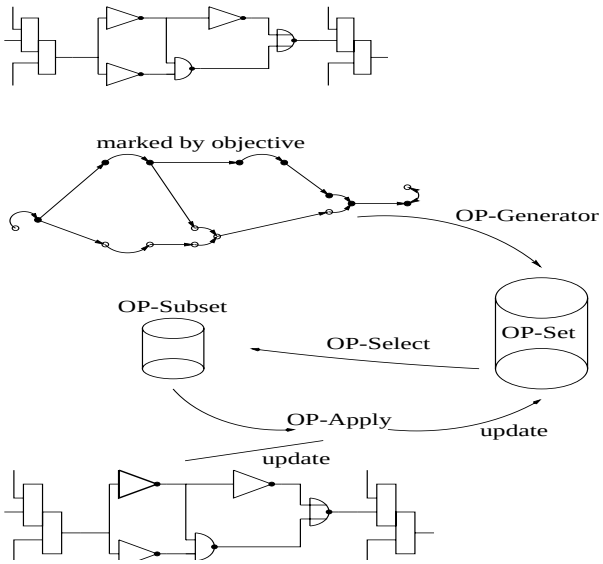

FIGURE 2. Visualization of the optimization scheme

```
(1) Optimize(graph, NodeObjective, OperationDriver,
                NodesetSelection) {
(2)    op_pool = 0;
(3)    lap = 0;
(4)    cnt = -1;
(5)    while (lap < MAXLAPS && cnt != 0) {
(6)       objective = 0;
(7)       if (!mark_active(graph, NodeObjective,
             &objective))
(8)       /* no nodes marked active */
(9)       break;
(10)      if (circular_objective(objective) ||
(11)          variance(objective) < THRESHOLD)
(12)      break;
(13)      ops = OperationDriver(graph, op_pool);
(14)      assign_cheapest_operations(graph, ops);
(15)      nodes = NodesetSelection(graph);
(16)      ops = collect_operations(nodes);
(17)      cnt = apply_operations(ops, op_pool, graph);
(18)      if (cnt) compute_timing(graph);
(19)   }
(20) }
```

FIGURE 3. The common optimization scheme

Each node corresponding to a pin with a negative slack is considered to be an *active* node.

The *NodeObjective* for the delay optimization is the negative of the sum of the slacks over all active nodes. Any reduction of this objective will make at least one path faster. The *OperationDriver* checks at each active node if there are any applicable operations and computes their effect. The operations to reduce the delay along a path can vary from a simple power-level change or repeater-insertion/deletion to more complex logic changes. These include swapping of logically equivalent pins and local logic resynthesis.

The delay computation depends on the slope at the tail of an arc as well as on the capacitance at the head of the arc. Any operation thus does not only effect the delay along the arcs the operation will modify, but also along some arcs preceding the modified part as well as some succeeding arcs. For example, a power-level change might influence the capacitance of the input pins of the cell. In this case the delay over the arcs leading in to the pins driving these inputs, must be recomputed. Furthermore, the slope at the output pins of the modified cell changes, forcing at least the recomputation of the delay along the paths to the subsequent output pins. According to this observation, each operation defines an “environment”. On one hand it is necessary to compute locally the total effect of the operation, on the other hand the computed effect of an operation will be shown if and only if the environment remains unchanged.

For this reason, operations at two different nodes are called *conflicting* if their environments intersect. The *gain* of an operation is obviously the difference of the worst slack of the actual implementation $SLK_{act,}$ and the worst slack after applying the operation SLK_{op}. The *cost* of an operation is the difference between the actual and the new area/power consumption. These values are stored at the operation.

All computed operations are stored in a pool *op_pool*, so that they can be used as long as the environment of the operation remains unchanged. The returned set *ops* is a subset of *op_pool* of operations with positive gain and no negative side-effects (i.e. creates no new design rule violations). Out of this subset the operation with the lowest cost per gain is stored at its node.

The decision *(NodesetSelection)* which nodes should be optimized is done by a minimal-cost node-cut procedure.

To compute the cut, a directed graph with capacitized arcs is derived from the active nodes of the timing graph in the following way:

- First the graph induced by the active nodes is constructed with arcs of infinite capacity.

- Each node for which an operation is stored, is split into two nodes, the first one connected to the inputs, the second one to the outputs.

- The two nodes are connected by an arc with its capacity equal to *cost* divided by *gain*, as derived from the corresponding operation.

- Two new nodes, *source* and *sink*, are created. The node *source* is connected to all nodes with no input arcs, the node *sink* is connected to all nodes with no output arcs. These arcs also have infinite capacities.

- Every path consisting only of arcs with infinite capacity is replaced by a single arc of infinite capacity. And any arc between source and sink is deleted.

In this graph, a minimum-capacity edge-cut which separates *source* and *sink* is computed. This is done by using the FIFO preflow-push algorithm presented in [6].

It follows that the cut must consist of edges created by splitted nodes only. In case that two adjacent cells of the logic are chosen, the corresponding operations are conflicting and one of them is deleted. This is sufficient because the deleted operation will be recomputed in the next lap with a correct environment, if the path still violates the timing constraints. This algorithm guarantees that at least one operation is selected for almost all paths with negative slacks.

```
(1) Late_Delay(logic) {
(2)     graph = create_graph(logic);
(3)     Optimize(graph, LoadLimit, LoadDriver,
(4)                 AllMarkedNodes);
(5)     Optimize(graph, SlackLimit, SlackDriver,
(6)                MaxNodeCut);
(7)     Optimize(graph, SlopeLimit, SlopeDriver,
(8)                 AllMarkedNodes);
(9)     Optimize(graph, LoadLimit, LoadDriver,
(10)                AllMarkedNodes);
(11)    Optimize(graph, PowerLimit, PowerDriver,
(12)                MaxAnitChain);
(13)    backannotate(graph, logic);
(14)    delete_graph(graph);
(15) }
```

FIGURE 4. The optimization environment

The scheme described above is used not only to optimize long path problems, but can be used to handle short path violations as well. This is achieved by inserting as little delay as possible, without introducing new long path problems.

9. Results

As is common for a static timing analysis methodology, coding the assertion files for the tool was not a trivial task. It took time and effort to correctly model our very complex clock-tree which contained many gated sub-trees of different clocks.

In addition, coding a coherent set of ATs and RATs for the off-chip nets proved to be more difficult than anticipated. These had to cover a set of different applications, packages and environments in which the chipset was to be used. Table 1 shows EinsTimers run-times and memory-requirements for different steps in the analysis of one of the larger designs (778k pins) in the chipset.

Step	Runtime	Memory
loading tool and timing-models	30 sec	9.4 MB
loading netlist	7 min	290MB
loading assertions and interconnect extractions	10 min	100MB
timing analysis of all paths	150 min	207MB

TABLE 1. Requirements for an IBM RS/6000 Model 590

The most useful results from an analysis run were:

- The “slack-report”, sorted by increasing slack, which pointed out the cycle-limiting paths with their origin and endpoints.

- A violations-report to alert the designer to overloaded outputs of gates as well as excessive slew-times at their inputs.

- The “clock-skew” report with the distribution of clock ATs at the latches, pointing to skew-problems in the clocktree generation.As all ATs (early and late) for all timing-points are propagated in a single run, it was not necessary to time the designs twice.

Solving the false-path problem can easily appear to be a daunting task. However, after overcoming some initial problems due to feedback-loops in the netlist, we found this to be much easier than anticipated [7]. To eliminate non-functional paths (e.g. scan-chains and test-only signals) from the design, the originating pins must be flagged as “Don’t Care”. Although the total number of non-functional paths in the design is quite large, we

needed only add a few dozen “Don’t Care” flags by confining our analysis to the paths with negative slacks.

We found EinsTimer’s graphical browser (Wizard), which annotates the schematic with timing-information, to be extremely useful while interactively debugging timing-problems.

As EinsTimer is a fully incremental tool, quick “what-if” analysis of paths was almost instantaneous, even on very large designs.

The optimization-tool was able to decrease the slack on all of our designs substantially. Table 2 shows the results for different stages in the optimization of the processor chip.

Optimization phase	worst slack	# of negative slacks
after placement	-8.46 ns	6771 endpoints
after powerlevel optimization	-1.87 ns	4478 endpoints
after repeater-insertion	0 ns	0 endpoints

TABLE 2. Optimization results

10. Current work

Although the described methodology has served us well, we are presently implementing extensions to take into account the effects of smaller geometries.

In order to calculate delays through gates and interconnects more precisely [8], we are now using AWE-based algorithms [9, 10] and are taking capacitive shielding by interconnect-resistance [11, 12] into account.

The rather crude assumption of an average population in adjacent channels of wires is being replaced by an interconnect-extraction that takes overlap- and fringing-capacitances into account.

Another difficult topic under consideration is that of taking manufacturing variations into account by using more refined methods than simply adding additional guard-times at tests.

To further enhance the interactive timing-capabilities, we are looking for the ability to selectively “zoom-in” on portions of the logic and analyze these with a SPICE-like simulator.

11. Conclusions

In order to keep the gate-level flow from synthesis all the way to tape-out under tight control with respect to timing, we used static timing analysis extensively throughout the methodology.

Having a single timing-tool and using the same timing-models for both synthesis and layout, we achieved a high level of consistency regarding timing-results.

By refining the interconnect models from step to step, we could asses the state of a design at any point in the layout-process and had a clear, accurate metric for tool-performance and design-decisions.

12. Acknowledgments

We would like to express our appreciation to Alex Suess of the EinsTimer development-team for his ongoing support of our programming efforts.

Furthermore, we would like to thank David Yearack of the EinsTimer support-team for relentlessly answering questions and providing insights into the operation of the tool.

Many thanks also to Asmus Hetzel, who implemented the operational part of the optimization system, and his useful and constructive discussions during the development of the running program.

13. References

[1] J.Koehl, U.Baur, B.Kick,T.Ludwig and T.Pflueger, “A Flat and Timing-driven Design System for a High-Performance CMOS Processor Chipset”, To appear in proceedings of DATE 1998

[2] W.C. Elmore “The Transient Analysis of Damped Linear Networks with Particular Regard to Wideband Amplifiers”, J. Applied Physics, vol. 19(1), 1948

[3] J.Vygen, “Algorithms for Large-Scale Flat Placement”, to appear in Proceedings of the 34st ACM/IEEE Design Automation Conference, June 1997

[4] O.Coudert, R.Hadded, S.Manne, “New Algorithms for Gate Sizing: A Comparative Study”, Proc. 33rd ACM/IEEE Design Automation Conference, June 1996

[5] O.Coudert, “Gate Sizing: a General Purpose Optimization Approach”, Proc. of ED&TC’96, March 1996

[6] A.V.Goldberg, “A new max-flow Algorithm”, Technical Report MIT/LCS/TM-291, Laboratory for Computer Science, MIT, Cambridge, MA

[7] Krishna P. Belkhale, Alex Suess, “Timing Analysis with known False Sub Graphs”, Proc. ACM/IEEE Intl. Conf. Computer-Aided Design, November 1995

[8] V. Rao “Delay Analysis of the Distributed RC Line”, Proc. 32nd ACM/IEEE Design Automation Conference, June 1995

[9] C.L. Ratzlaff, N.Gopal and L.T.Pillage. “RICE: Rapid Interconnect Circuit Evaluator”, Proc. 28th ACM/IEEE Design Automation Conference, June 1991

[10] L.T. Pillage and R.A. Rohrer, “Asymptotic Waveform Evaluation for Timing Analysis”, IEEE Transactions on Computer Aided Design, April 1990

[11] J.Qian, S.Pullela, L.T.Pillage, “Modeling the *effective capacitance* for the RC Interconnect of CMOS Gates”, IEEE Trans. Computer Aided Design, vol. 13, no. 12, December 1994

[12] F.Dartu, N.Menezes, J.Qian, L.T.Pillage, “A gate-delay model for high-speed CMOS circuits”, Proc. 31st ACM/IEEE Design Automation Conference, June 1994