A System for Automatic Recording and Prediction of Design Quality Metrics *

Andrew B. Kahng † and Stefanus Mantik

† UCSD CSE and ECE Departments, La Jolla, CA 92093-0114
UCLA Computer Science Department, Los Angeles, CA 90095-1596
abk@ucsd.edu, stefanus@cs.ucla.edu

Abstract

We present recent extensions to METRICS [10] infrastructure that allow optimization of design processes at the flow level, rather than only at the individual tool level. As previously reported, METRICS infrastructure allows automatic recording of design and process information. Our extensions include (i) the collection of design flow information for use in flow optimization, and (ii) integration with datamining tools to allow automatic generation of design and flow QOR predictors. Our flow optimization experiments try to optimize incremental multilevel FM partitioner runs in an incremental (ECO-oriented) design flow. We also demonstrate QOR predictors that are generated automatically from the METRICS data warehouse by the *Cubist* datamining tool for industry placement, clock tree generation, and routing tools.

1 Introduction

Time-to-market windows for the semiconductor industry are shrinking rapidly even as product quality must continually improve to maintain competitiveness. In this regime, it becomes important to institutionalize continuous measurement and improvement of design quality. Our work focuses on improving quality of the design *process*. To a large extent, improving today's VLSI design process involves tooling decisions, i.e., designers must answer such questions as "which tools should be used for the design?", or "what flow should be applied?". Similar decisions must be made with respect to design IP. Today, there is no standard infrastructure that enables designers to make principled choices as to the tools and flows that they use.

In [10], we presented METRICS, a system that allows automatic recording of design process information. METRICS is a web-based architecture based on industry-standard components (HTTP, RDBMS, XML, Java servlet, etc.) and standardized tool metrics (naming and semantics); it allows automatic and fair comparisons of both design instances and design optimization results.

1.1 METRICS Architecture

The METRICS architecture shown in Figure 1 is a specific implementation of a distributed, client-server information gathering system. The EDA tools, which are the data sources, have a thin transmitter client embedded in script wrappers surrounding the tool or actually embedded (as an API) inside the tool's executable for more flexibility. Both the wrapper mode transmitter and the API mode transmitter allow transparent data collection within design processes. The tools – which can be located anywhere on an intranet or even the internet – broadcast in real-time as they run using standard network protocols to a centralized server which is attached to a *data warehouse*. The messages transmitted are encoded in industry-standard XML format, which is straightforwardly and robustly read, written and stored directly by data warehouses. Once the data is stored, reports or datamining-based predictions can be generated. These can be accessed from standard web browsers run from any authorized remote machine. Further details of METRICS are given in [10].

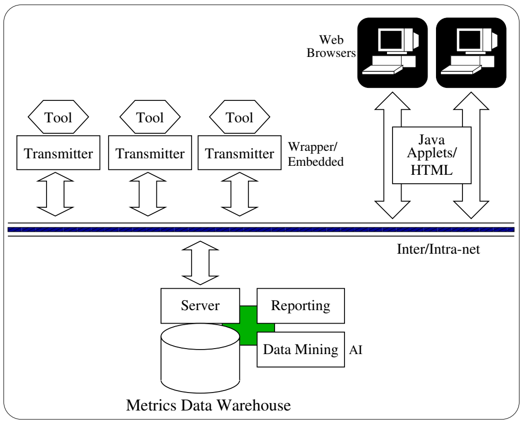

Figure 1: METRICS architecture.

1.2 Extensions to the METRICS Architecture

In this paper, we focus on two recent extensions to the METRICS architecture. First, we propose a schema that is able to capture historical flow information, even for highly iterative or ECO-oriented flows with multiple potential "backward edges". The previous METRICS architecture lacked information related to design flows, and hence it was not possible to optimize design flows. Our new schema allows us to predict, for example, how many times a certain tool or optimization subflow must be repeated before an acceptable result is obtained. Second, we integrate off-the-shelf datamining techniques and assess the ability of such tools to identify variable sensitivities, predict quality of results, estimate tool runtimes, and in general guide designers in making correct design process decisions.

Our paper is organized as follows. Section 2 reviews previous and related works. In Section 3, we describe the schema that is used to maintain historical flow data. Application of datamining is then explained in Section 4. Finally, experimental results are given in Section 5 and conclusions in Section 6.

2 Previous Works

Process data collection. Johnson et al. [13, 14] used a survey-based approach to collection and analysis of VLSI design process data in the aviation industry. In the survey-based approach, each

* This work was supported by a grant from Cadence Design Systems, Inc. and by the MARCO Gigascale Silicon Research Center.

designer fills out a survey form at the end of each design step. The form asks for the amount of time needed (for both conceptual design and actual implementation), the number of times a specific task is repeated, the reason for a specific decision that the designer takes, etc. This method of data collection is flexible in that designers can use any tools they want, but it is also obtrusive and at risk for incompleteness and inaccuracy (since designers can only enter information that they still remember). Johnson et al. also use a passive monitor, a background process that automatically collects some design process data. The passive monitoring requires designers to use predefined sets of tools and flows.

Data collection has also been performed for many other reasons beyond analysis for process improvement. In [20], design data are collected for generating a *classification* of such data. This classification is useful for identifying sets of benchmarks that are suitable for particular tools or algorithms, and for performance comparison of different algorithms. Various in-house *project tracking* systems have been developed (at LSI Logic, IBM ASIC Division, Siemens Semiconductor (Infineon), Sony, Texas Instruments, etc.), each with its own proprietary measures of designs and the design process. Such systems (e.g., at Texas Instruments [7]) are often used for tracking license activity information to assess efficiency of tool usage. Numetrics Management Systems [18] offers survey-based collection of *enterprise-level design productivity* metrics, allowing companies or entire projects to compare their productivity against entire industry sectors.

Metrics. To evaluate design quality, we must capture the right *metrics*. However, there are many metrics that could be used to measure design quality. [8] gives examples of important quality metrics that should be captured in chip design. [15, 21] add several other metrics for measuring design quality. Metrics are not limited to design metrics: *process metrics* (e.g., "how long does a tool run?", "what is the maximum memory used?", etc.) and *flow metrics* (e.g., "how many times does this tool need to be repeated?", "which tools need to be run before others?", etc.) also need to be captured. Johnson et al. [11] call such metrics *meta-data* and use them to build Markovian flow models that allow bottlenecks and total flow completion times to be estimated [13]. CAD frameworks provide environments for efficient design tool integration wherein, e.g., automated flow managers replace manually created shell scripts [2, 3, 5] and allow more user interaction in controlling the design flow. Within a CAD framework, flow data can be automatically recorded, permitting analyses that would not be possible with tool-only data collection.[1]

Datamining. Finally, datamining is important to build predictors of design and process quality metrics. Datamining is a set of techniques for identifying common rules and patterns that occur within a given set of training data, for later classification and prediction of analogous data. Datamining for prediction has been successful in such wide-ranging areas as atmospheric science, data warehousing, advertisement, etc. [9]. Shin et al. [19] use datamining on software metrics databases to extract relevant knowledge for improving software quality and productivity. In the present work, we use the commercially available *CUBIST* datamining tool [6], which creates a piecewise linear model of a pattern that it extracts from training data. We then apply the constructed model to test data to verify accuracy of the predictive model (e.g., in terms of correlation coefficient of the predictor).

3 Flow Taxonomy

Previous works establish gate count, number of routing layers, percentage of white space, etc. as standard metrics that contribute to design quality. However, such metrics must be augmented by, e.g., number of incremental placement invocations, number of loop-backs, etc. for us to improve the design process. In this section, we give a general definition of a design flow and describe a schema for tracking flow information, along with an illustrative example.

3.1 Flow Definition

A *sequential design flow* consists of several *tasks* that must be run in sequence. A *task* is an atomic part of the flow, e.g., a single invocation of a tool (which may contain several different algorithms) that solves some specific problem. For example, a placement task (or tool) may comprise multiple global placement passes, detailed placement passes, and annealing refinement passes. A task may also comprise actions that convert the input design or output result between various representations.

Formally, a design *flow* can be represented as a directed graph $G(T,E,S,F)$ where $T = \{T_1, T_2, \dots, T_n\}$ is a set of vertices that corresponds to the set of tasks that can be executed in the flow, and $E = \{E_{11}, E_{12}, \dots, E_{nn}\}$ is a set of directed edges with E_{ij} indicating a *transition* from task T_i to task T_j (i.e., T_i is followed by T_j). $S \in T$ is a special task node (with zero execution time) indicating the beginning of the flow, and $F \in T$ is another special task node indicating the end of the flow. If the graph is traversed from S to F visiting each node exactly once, then we say the tasks are executed in *sequential order*. By convention, we index tasks such that if tasks are executed in sequential order, $t_{start}(T_i) \leq t_{start}(T_j)$ $\forall i < j$, where $t_{start}(T_i)$ is the starting time of T_i. (Note that the $\leq$ comparison is used instead of $<$; this allows extension of our model to concurrent subprocesses in the flow. Flows containing optional tasks can also be represented with this model.) An example of the flow representation is given in Figure 2.

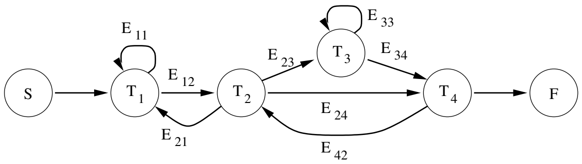

Figure 2: Sample design flow representation.

Edges E_{ij} are categorized into three types: (i) a *forward edge* ($j > i$) indicates a transition from the current task to a new task (a task that has may not have been visited yet), (ii) a *backward edge* ($j < i$) indicates a transition from the current task to a previously visited task, and (iii) a *self-loop edge* ($j = i$) indicates repetition of the current task. In a sequential flow, a forward edge typically exists only between adjacent tasks ($j = i+1$), but other forward edges can exist if the intervening tasks are optional. For example, if there is an edge from T_i to T_{i+2}, then the transition from T_i to T_{i+1} is an optional transition. On the other hand, backward edges can be from any task to any other previous task.

In any particular execution of a given flow, backward and self-loop transitions create an *execution hierarchy* as seen in Figure 3. We call a task that is executed before the current task in the sequential order a *parent* task, and a task that is executed after the current task in the sequential order a *child* task.[2]

3.2 Database Schema for Flow Tracking

The METRICS infrastructure as originally proposed in [10] does not track task relations within a flow. In executing any given design flow with possible backward and self-loop transitions, we seek to

[1] For example, running the router before placing the clock tree is not a good flow - but this cannot be deduced by looking only at the clock tree generation tool metrics.

[2] If we remove all backward edges and all self-loop edges, then add an edge for every parent-child pair as seen in Figure 3, the flow execution can be viewed as a tree whose root (not shown) is connected to all first-level tasks. From this viewpoint, it is natural to use the parent-child relationship to characterize two adjacent tasks that are in different task "levels".

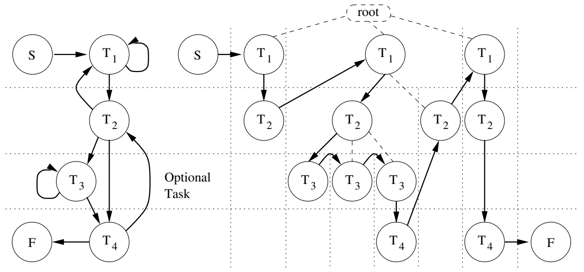

Figure 3: Any particular execution of a given flow creates an execution hierarchy.

record the exact sequence of tasks and their (parent or child) relation to other tasks. To this end, we have added a simple system for recording such "flow metrics" into the METRICS infrastructure.

Our proposed flow metrics (see `http://vlsicad.cs.ucla.edu/GSRC/METRICS`) include starting and completion times of the flow, the design environment setup, the number of distinct tools used, the number of times each specific tool is executed, etc. Two additional flow metrics, the *TASK_NO* and the *FLOW_SEQUENCE*, identify the execution order of each task and the relationship between consecutive tasks.

- The *TASK_NO* represents the number of times a given current task has been executed within the same execution of its parent task. If a self-loop or backward transition to a specific task is made, the *TASK_NO* for that specific task is always incremented. On the other hand, if a forward transition to that specific task is made, the *TASK_NO* is reset to one.
- The *FLOW_SEQUENCE* records the execution hierarchy starting from the first task. Its value is a concatenation of all current *TASK_NO*s from the first task to the current task (separated by a "/"). For example, if we are currently in task T_3 (*TASK_NO* = 2) and the *TASK_NO* for T_1 and T_2 are respectively 3 and 5 respectively, then our *FLOW_SEQUENCE* for the current task is "3/5/2". The *FLOW_SEQUENCE* metric allows us to determine the input source of the current task, i.e., which execution came before the current one.

3.3 Example

We illustrate the use of *TASK_NO* and *FLOW_SEQUENCE* metrics in Figure 4. The flow is a standard place and route flow with clock tree generation before routing. T_1, T_2 and T_3 respectively correspond to the placement, clock tree generation and routing tasks. Each task has a backward edge to T_1, indicating that another placement run is needed (if routing after clock tree generation does not converge). Similarly, T_2 may be revisited if the routing task cannot find a solution. Each task also has a self-loop edge for incremental invocation, i.e., when the result of the current task is not acceptable, the task may be repeated.

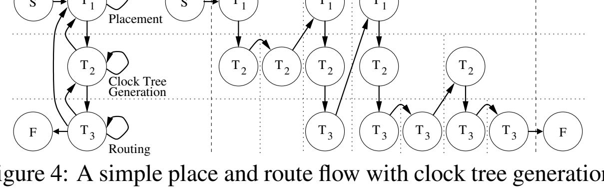

Figure 4: A simple place and route flow with clock tree generation.

The right side of Figure 4 shows an execution order of the tasks, with task sequence $S, T_1, T_2, T_2, T_1, T_2, T_3, T_1, T_2, T_3, T_3, T_2, T_3, T_3, F$. Values for *TASK_NO* and *FLOW_SEQUENCE* for each task are shown in Table 1 (note that R_i is used in the Table to indicate the specific task with run number = i). The *FLOW_SEQUENCE* metric enables unique reconstruction of the execution order.[3]

Run No.	Current Task	*TASK_NO* T_1	T_2	T_3	*FLOW_SEQUENCE*
R_1	T_1	1	-	-	1
R_2	T_2	1	1	-	1/1
R_3	T_2	1	2	-	1/2
R_4	T_1	2	-	-	2
R_5	T_2	2	1	-	2/1
R_6	T_3	2	1	1	2/1/1
R_7	T_1	3	-	-	3
R_8	T_2	3	1	-	3/1
R_9	T_3	3	1	1	3/1/1
R_{10}	T_3	3	1	2	3/1/2
R_{11}	T_2	3	2	-	3/2
R_{12}	T_3	3	2	1	3/2/1
R_{13}	T_3	3	2	2	3/2/2

Table 1: A sample flow sequence with its corresponding em TASK_NO and *FLOW_SEQUENCE* values. R_i is the run number for the task according to the execution order.

While our discussion of flow metrics has been kept simple for purposes of illustration, non-sequential design processes can also be handled easily. Observe that the "sequential order" illustrated above corresponds to a *total ordering* of tasks within the flow. If a flow has concurrent processes (i.e., two or more tasks or task subflows are independent of each other and can be executed in parallel (or, in arbitrary order)), then the flow graph represents a *partial ordering* of tasks within the flow. Our *FLOW_SEQUENCE* metric can be augmented to capture concurrent design processes, essentially by adding AND, OR, XOR semantics.[4] Furthermore, an additional identification is added to the *FLOW_SEQUENCE* to identify the specific branch (*BRANCH_NO*) of the concurrent process that corresponds to the current data. For example, if at some point our current process are split into two or more subprocesses for concurrent execution, then the *id* for the next task will incorporate the additional *BRANCH_NO*, e.g., 2/1-1, 2/2-1, 2/3-1, etc. (*x*-1 is the *TASK_NO* for the next task with *x* as the *BRANCH_NO*.) Metrics are also possible for ECO flows (where the underlying design instance is changed between consecutive task executions).

4 Usage of Datamining Tools

Datamining has been used to extract common patterns from large datasets in many domains. These patterns are then used for prediction of future data/results. We now review recent integration of a commercial datamining tool into the METRICS infrastructure, and some early observations concerning this integration.

4.1 Integration with Datamining Tools

When metrics data is sent through the transmitter, the data is stored in a centralized database. A Java interface built on top of this database is used both for receiving the metrics and for generating reports. In similar fashion, another Java interface is created for the purpose of datamining. This *datamining interface* (DMI) enables communication between datamining tools and the database. It also provides an interface for users to control the datamining process (see Figure 5).

[3]The relation between two tasks can be found by looking at their longest common prefix of their *FLOW_SEQUENCE*s. For example, the task R_{13} (the last task with run number = 13) has "3/2/" as its longest common prefix with task R_{12}. This means that both tasks are the descendants of another task that has *FLOW_SEQUENCE* = "3/2", i.e., the task R_{11}. Since both task R_{12} and task R_{13} have only one extra digit (and no more "/"), these tasks are direct descendants (i.e., children) of task R_{11}.

[4]I.e., to record how the output of a task may split into inputs of several concurrent tasks, or how after concurrent tasks are completed their outputs are merged into a single result that is the input to the next task.

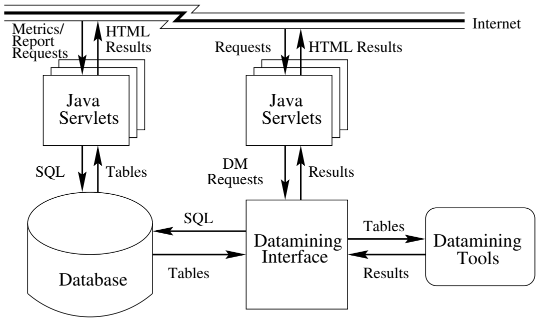

Figure 5: Integration of datamining tools within the METRICS architecture.

Our current implementation of METRICS uses *CUBIST* [6] as the datamining tool. This tool is chosen because it produces a single *absolute* number as its result (e.g., number of CPU seconds, or number of passes to reach timing closure), in contrast to tools such as *C5* or *CONTIN* that will only produce categories (without building a model from those categories). CUBIST furthermore returns a set of rules that define a prediction model. To run this tool, we need to provide (i) a list of parameters with the attributes of their values (e.g., continuous numbers, discrete values, etc.), (ii) a dataset for training, and (iii) (for evaluation) a different dataset for testing. The tool uses the training set for the generation of its (piecewise-linear) rule-based predictive model, and it uses the test set to check the accuracy of the model.

Our CUBIST integration allows users to select a variable as the prediction target and a set of variables as the inputs for the prediction. The selection can be done over the internet via an HTML form. Once the users submit their selections, the DMI sends a query to the database, results of which are passed to the datamining tool. At the end, the DMI passes the datamining results to the users through web sites. Several additional tasks can also be assigned to the DMI, e.g., data cleanup, value transformations, variable reductions, etc.

4.2 Example Applications

The basic usage of the DMI and datamining tool integration is in creation of predictors/estimators from collected data. Other benefits can include:

- *Parameter sensitivity analysis*, i.e., analysis of which input parameters have the most impact on tool results. As datamining tools give insights on how design tools behave when certain changes are made to specific parameters, we are able to use the design tools more effectively (e.g., preventing runtime wastage due to tweaking of knobs that don't matter).
- *Field of use analysis*, i.e., analysis of the (runtime, capacity, quality) limits at which the tool will break. In our interactions within the METRICS community, we have found high interest in analysis of "sweet spots", i.e., the ranges of input attributes for which a given tool will give best results. To perform sweet spot analysis, we need to run the tools with sufficiently many different input designs; whether these must be real, or can be "mutated" from real or randomly generated, is an open issue.
- *Process monitoring*, i.e., analysis of potential or likely outcomes of the current design process (while the process is still running). Since tool and flow metrics are sent directly to the database in real time, datamining tools can calculate possible outcomes and QOR metrics for the design process online. Given the computed predictions, designers can decide whether they should stop the current process (e.g., given high likelihood of bad results) or let it run to completion. Such process monitoring can potentially shorten the design cycle by reducing time spent on doomed tool runs.
- *Resource monitoring*, i.e., analysis of resource demands for given tasks. We can use datamining tools to identify unsafe resourcing conditions, e.g., if we run design tools on machines with too-small memory or disk. Again, design cycle time can be improved by preventing runs on ill-configured machines.

Most of these example applications require tighter integration (additional interfaces and controls) between design tools and datamining tools. Some design checks and resource checks can be integrated with available CAD frameworks, e.g., the "flow manager" could check if the tools will run on the given machine and if the design inputs are in the field of use for the tools. Web-based monitoring can also be implemented to monitor the current process. In the next section, we present sample experimental results from the first type of integration – using datamining tools to generate runtime and QOR predictors – which has been developed for our METRICS architecture.

5 Experimental Results

Our METRICS data warehouse has been set up on a server with Oracle8i database, Java servlets, and Apache web server as shown in Figure 1. Integration of the *Cubist* datamining tool [6] has been performed as shown in Figure 5. Two different types of experiments are performed: (i) flow optimization, and (ii) datamining.

5.1 Flow Experiments

Our flow experiment simulates the optimization of a design process. Our "process", or "flow", is built around an incremental multilevel Fiduccia-Mattheyses hypergraph partitioner which solves the *incremental* netlist partitioning problem. Given an initial partitioning instance I_{init}, an initial solution to that instance S_{init}, a perturbation ΔI, and a CPU budget, we seek to optimize the use of a V-cycling based incremental multilevel FM partitioner.[5] In other words, we wish to tune the application of the incremental partitioner so that it returns the best possible solution quality (in terms of minimizing the number of nets cut) within the prescribed CPU budget.

With this experiment, we can find the flow tuning that gives the best final solution S_{final} for the final instance I_{final}, which is derived from I_{init} by applying the perturbation ΔI. The instance is perturbed by changing the weights of various hyperedges (signal nets). The number of nets that are reweighted is the size of perturbation. For purposes of the incremental optimization, the instance perturbation can be broken down into several smaller perturbations, i.e., $\Delta I = \delta I_1 + \delta I_2 + \ldots + \delta I_n$ (the "breakup"), and various numbers of multistarts can be applied to each resulting instance. The best result from the multistarts on one instance is used as the starting point for all starts on the next instance. The quality of the result is based on the final instance (I_{final}). Figure 6 illustrates the flow setup.

We run our experiments on 8 standard test cases in the modern partitioning literature – the ibm01-06, ibm08 and ibm10 instances of [1]. For each test case, we run 10000 different combinations of ΔI, CPU budget, number of breaks, and number of starts (per break). Once the data are collected, we run the datamining tool to generate rules that give us the optimized flow for a given design with the specified perturbation size and CPU budget, i.e., the datamining tool will predict the number of breaks

[5] Such an incremental partitioner is described in [16] and [4] (see in particular the third and fourth pages of the latter reference). It has the advantage of retaining structurally similar solutions from iteration to iteration; we believe that it is appropriate for ECO-type partitioning applications, but do not specifically test this feature in our experimental setup or process optimization.

("num_inc_parts") and the number of starts ("num_starts"). Table 2 shows the first five out of the 30 rules produced by CUBIST.

```
foreach testcase
  foreach ΔI
    foreach CPU_budget
      foreach breakup (n = number of parts)
        I_current = I_initial
        S_current = S_initial
        for i = 1 to n
          I_next = I_current + δI_i
          run incremental multilevel FM partitioner
              on I_next to produce S_next
          if CPU_current > CPU_budget then break
          I_current = I_next
          S_current = S_next
        end for
        save number of cuts
      end foreach
    end foreach
  end foreach
end foreach
```

Figure 6: Flow setup for multilevel FM partitioner.

Rule	Condition(s)	Model
1	27401 < num_edges ≤ 34826 143.09 < cpu_time ≤ 165.28 perturbation_delta ≤ 0.1	num_inc_parts = 4 num_starts = 3
2	27401 < num_edges ≤ 34826 85.27 < cpu_time ≤ 143.09 perturbation_delta ≤ 0.1	num_inc_parts = 2 num_starts = 1
3	num_edges ≤ 14111 213.17 < cpu_time 0.01 < perturbation_delta ≤ 0.05	num_inc_parts = 10 num_starts = 20
4	14111 < num_edges ≤ 27401 121.38 < cpu_time ≤ 155.69 perturbation_delta ≤ 0.01	num_inc_parts = 4 num_starts = 5
5	num_edges > 14111 cpu_time < 76.13 perturbation_delta > 0.1	num_inc_parts = 1 num_starts = 1

Table 2: CUBIST-derived rules for flow optimization experiments.

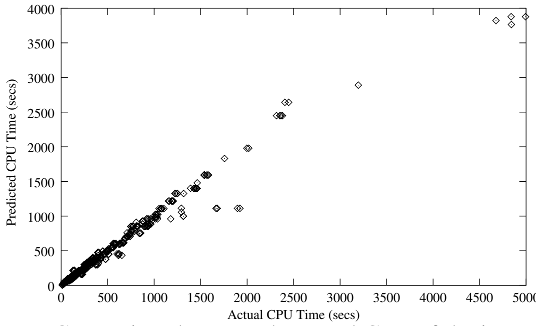

Figure 7: Comparison between the actual CPU of the incremental run using the CUBIST-suggested flow, and the allotted CPU time.

To test the predictor, we take a particular (test case, delta, CPU time) instance of the flow, and see what the predictor says should be the value of #breaks ("num_inc_parts") and #starts/break ("num_starts"). We then examine (1) whether the prescribed configuration uses about the right amount of CPU time, and (2) whether the prescribed configuration gives a better result than plain from-scratch multistart optimization of instance I_{final} for the given CPU time (we equalize the "actual" CPU times so that the from-scratch optimization uses the same actual time that the incremental optimization takes). Figure 7 shows the comparison between the actual time it takes to run the prescribed configuration, versus the allotted CPU time. The average difference between the number of weighted cut nets of the incremental run with the prescribed configuration and the number of weighted cut nets of the from-scratch optimization is -12.59 with standard deviation of 99.14 (i.e., the prescribed incremental run on average gives a better result than from-scratch optimization).

5.2 Datamining Experiments

Our datamining experiments seek rule-based QOR predictors for various tools. Our central experiments are based on approximately 10500 runs of the Cadence System Level Constraints (SLC) flow with more than 200 different industry test cases. The design tools that are used in the flow are Cadence QPlace, CTGen, and WarpRoute. The tools are executed on different machines and the metrics are sent to the METRICS server through the intranet.

Cubist is applied on the collected data to generate predictors. For the placement tool, runtime predictors and wirelength predictors are generated based on the size of the design, utilization factor, etc. For the clock tree generation tool, minimum insertion delay predictors, maximum insertion delay predictors, and maximum skew predictors are generated based on the input constraints, number of sinks, etc. For the routing tool, routability predictors are generated based on the utilization factor, the percentage of over capacity gcells, etc. Table 3 shows an example model that is generated by the datamining tool. It shows the first five out of the 15 rules in the QPlace CPU time predictor obtained from current data.[6]

Rule	Condition(s)	Model
1	num_nets ≤ 7332	CPU_time = 21.9 + 0.0019 num_cells + 0.0005 num_nets + 0.07 num_pads - 0.0002 num_fixed_cells
2	num_nets > 2902 num_nets ≤ 7332	CPU_time = 334.8 - 0.0233 num_nets + 0.0064 num_cells - 1 row_utilization - 0.0002 num_fixed_cells
3	num_overlap_lyr ≤ 0 num_cells ≤ 71413 opt_TD_routing = false	CPU_time = -15.6 + 0.0888 num_nets - 0.0559 num_cells - 0.0015 num_fixed_cells - 6 num_overlap_lyr - 1 num_routing_lyr
4	num_overlap_lyr > 0 num_cells ≤ 18180 num_nets > 7332	CPU_time = 3062.6 - 0.0532 num_cells + 0.0394 num_nets - 17 row_utilization - 160 num_layer - 0.0011 num_fixed_cells - 9 num_routing_lyr - 4 num_overlap_lyr
5	num_cells > 18180 num_pads ≤ 144 num_nets > 7332 opt_TD_routing = false	CPU_time = -2352.3 + 8.64 num_pads + 0.0128 num_cells + 45 row_utilization - 282 num_layer + 0.0014 num_nets - 8 num_routing_lyr

Table 3: First five rules of the QPlace CPU time model.

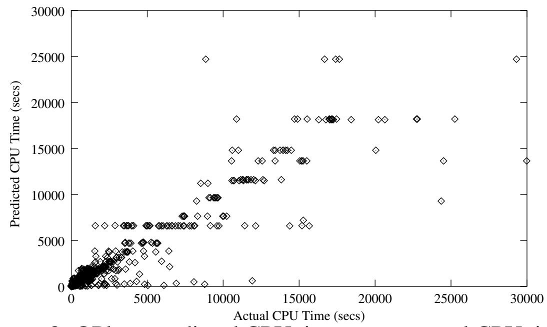

Figure 8: QPlace predicted CPU time versus actual CPU time.

The prediction result for QPlace CPU time is shown in Figure 8. This plot shows that the datamining tool produces a reasonably accurate model. The correlation coefficient between the predicted values and the corresponding actual values is 0.82 with the average (and relative) absolute error being 560.9 secs (0.24). However, the accuracy obtained may depend on the data that are selected for training. To check this dependency, we perform datamining runs on three different configurations for the training set and test set. The configurations are:

[6]In practice, this model will change as new data are collected. In general, (i) as more data are collected, the more accurate the models become, and (ii) the most recent data are more correlated to the current design.

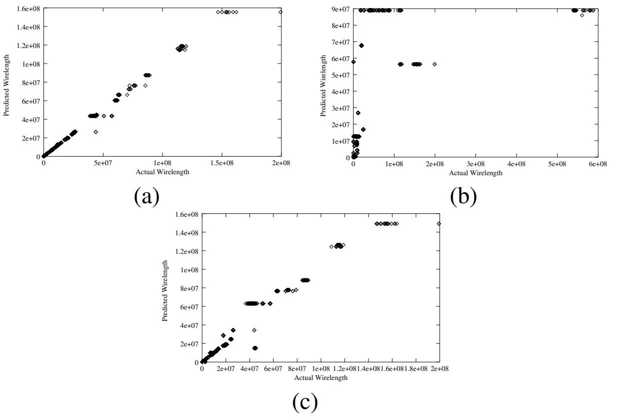

Figure 9: Predicted QPlace wirelength versus actual wirelength: (a) random, (b) distinct, and (c) representative cases.

1. the *random* case, where we randomly select runs assigned to the training set from all runs of all test cases, and leave all remainined (unselected) runs for the test set;
2. the *distinct* case, where we split the test cases into two distinct sets, the training set and the test set, and assign their runs accordingly; and
3. the *representative* case, which is similar to the distinct case except that we move exactly one run for each test case in the test set to the training set – i.e., for each test case, there is at least one representative run in the training set.

We run each configuration for predicting the placement wirelength. Results for each configuration are shown in Figure 9. The correlation coefficients for the random, distinct, and representative cases are 1.00, 0.48, and 0.82 respectively.

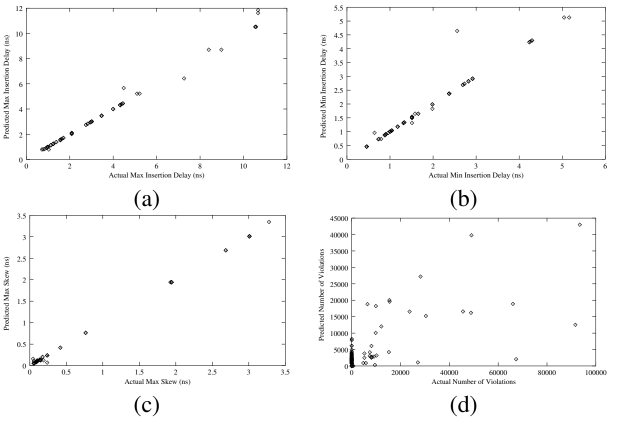

Figure 10: Plot of the predicted values against the actual values for (a) max insertion delay, (b) min insertion delay, (c) max skew, and (d) routing violation.

From the results in Figure 9 we see that the datamining tool needs to have at least one representative run for the design in order to produce an accurate prediction. In other words, if the current data does not have any tool run on designs that are similar to the current design, one may need to run the tools on the current design at least once to "sensitize" the datamining tool to the new design. This requirement may be due to our use of a rule-based datamining method instead of a formula-based one such as linear regression. However, it is clearly a weakness in our current implementation, since new designs are typically quite different from any previous ones. We believe that other prediction methods such as regressions must be used in conjunction with datamining. Finally, datamining results for CTGen and WarpRoute are shown in Figure 10.

6 Conclusions and Ongoing Work

We have extended the METRICS system of [10] to include the ability to record and optimize design flows, as well as generate predictors via integrated datamining capability. Our model for sequential design flows allows the METRICS system to keep track of tool invocations within a given flow, even with arbitrary backward edges or self-loops in the task sequence. Extensions to concurrent tasks and ECO flows are possible. With the integration of datamining tools, different predictors (CPU time, field of use, etc.), monitors, as well as flow optimizations become possible. Metrics transmittal and recording remains transparent to designers, and the web-based architecture supports reporting, datamining, and other data analysis from any authorized machine on the internet.

We have also presented several example applications that use our new extensions. In particular, after recording design metrics from a Cadence SLC flow with more than 200 real designs, we are able to generate rule-based result estimators for placement, clock tree generation, and routing tools. These estimators give almost perfectly accurate predictions as long as there is at least one representative run in the training set for each design case. Although the accuracy is promising, representative runs are typically not available for new designs. Thus, our ongoing research seeks to augment current datamining tools with other estimation methods in order to yield a more robust prediction methodology.

References

[1] C. J. Alpert, "The ISPD98 Circuit Benchmark Suite", *Intl. Symp. on Physical Design*, 1998.

[2] K. O. ten Bosch, P. Bingley, and P. van der Wolf, "Design Flow Management in the NELSIS CAD Framework", *Proc. ACM/IEEE Design Automation Conf.*, 1991, pp. 711-716.

[3] F. Bretschneider, C. Kopf, and H. Lagger, "Knowledge-Based Design Flow Management", *Proc. IEEE Intl. Conf. on Computer-Aided Design*, 1990, pp. 350-353.

[4] A. E. Caldwell, A. B. Kahng and I. L. Markov, "Improved Algorithms for Hypergraph Bipartitioning", *Proc. Asia and South Pacific Design Automation Conf.*, Jan. 2000, pp. 661-666.

[5] A. Casotto and A. Sangiovanni-Vincentelli, "Automated Design Management using Traces", *IEEE Trans. on Computer-Aided Design of Integrated Circuit and Systems*, vol. 12, August 1993, pp. 1077-1095.

[6] `http://www.rulequest.com/cubist-info.html`

[7] S. Baeder, Notes from DAC'96 Birds of a Feather meeting, *personal communication*, 2000.

[8] A. H. Farrahi, D. J. Hathaway, M. Wang, and M. Sarrafzadeh, "Quality of EDA CAD Tools: Definitions, Metrics and Directions", *Proc. IEEE Intl. Symp. on Quality Electronic Design*, March 2000, pp. 395-405.

[9] U. M. Fayyad, G. Piatetsky-Shapiro, and P. Smyth, "From Data Mining to Knowledge Discovery: An Overview", *Advances in Knowledge Discovery and Data Mining*, AAAI Press, 1996, pp. 1-34.

[10] S. Fenstermaker, D. George, A. B. Kahng, S. Mantik and B. Thielges, "METRICS: A System Architecture for Design Process Optimization", *Proc. ACM/IEEE Design Automation Conf.*, June 2000, pp. 705-710.

[11] E. W. Johnson and J. B. Brockman "Towards a Model for Electronic Design Process Refinement", *Computers in Industry*, vol(30), 1996, pp. 27-36.

[12] E. W. Johnson, J. B. Brockman, and R. Vigeland, "Sensitivity analysis of iterative design processes", *Proc. of Intl. Conf. on Computer Aided Design*, San Jose, 1996, pp. 142-145.

[13] E. W. Johnson, *Analysis and Refinement of Iterative Design Processes*, Ph.D. Thesis, Computer Science and Engineering Dept., Univ. of Notre Dame, 1996.

[14] E. W. Johnson and J. B. Brockman "Measurement and Analysis of Sequential Design Processes", *ACM Transaction on Design Automation of Electronic Systems*, Vol. 3(1), January 1998, pp. 1-20.

[15] M. Keating, "Measuring Design Quality by Measuring Design Complexity", *Proc. IEEE Intl. Symp. on Quality Electronic Design*, 2000, pp. 103-108.

[16] G. Karypis and V. Kumar, "hMetis: A Hypergraph Partitioning Package Version 1.5", *user manual*, June 23, 1998.

[17] G. Karypis and V. Kumar, "Multilevel *k*-way Hypergraph Partitioning", *Proc. ACM/IEEE Design Automation Conf.*, 1999, pp. 343-348.

[18] `http://www.numetrics.com`

[19] M. Shin and A. L. Goel, "Knowledge Discovery and Validation in Software Metrics Databases", *Proc. of SPIE Data Mining and Knowledge Discovery: Theory, Tools, and Technology*, April 1999, pp. 226-233.

[20] N. Whitaker, "Classification of Electronic Design Data", *Comp. Science Dept. Tech. Report*, University of Manchester, Document No. STEED/T1/02/2, 1998.

[21] G. Ben-Yaacov, L. Bjork, and E. P. Stone, "Advancing Customer-Perceived Quality in the EDA Industry", *Proc. IEEE Intl. Symp. on Quality Electronic Design*, March 2000, pp. 411-414.