A Database System for Interconnect Parasitics*

N. S. Arun M. Krishna Jayant Haritsa

Department of Computer Science and Automation
Indian Institute of Science, Bangalore 560012, INDIA

Abstract

Due to drastic reductions in feature sizes, device interconnects have begun to occupy a considerable portion of the layout in modern VLSI chips. A related consequence is that *interconnect parasitics*, which are unintended harmful side-effects that arise as a consequence of the layout, can potentially have a significant adverse impact on chip performance. Therefore, the electrical characteristics of these parasitics have to be taken into account in the design of IC chips. This involves processing data that is both large in size and complex in nature, calling for effective data management. Current solutions employ a file-based approach that suffers from many drawbacks, necessitating a search for alternative methods of data management.

Our research explores the use of a database technology-based approach to address the parasitic data management problem. In this paper, we present a detailed object model for representing interconnect parasitics. We also describe the design and implementation of **DIAS**, a database system that implements the parasitic object model and is tuned to efficiently processing interconnect related information. DIAS uses the public-domain SHORE toolkit as the backend database engine and has been written using a combination of the Shore Data Language (SDL) and C++. Users of DIAS can interact with the system through either a Query Interface or a Tool Interface. The Query Interface is used for dynamically formulating queries while the Tool Interface is used for supporting existing (legacy) design tools that require a file-system interface, rather than a database interface. DIAS also provides a Programming Interface in the form of a library of routines that can be used to manipulate the database. These routines can be used for developing new design tools that directly operate on the database interface. For efficient processing and fast access of data, access methods such as B^+ tree, hcC-tree, nested index and path index have been incorporated.

In summary, DIAS forms part of an information framework that extends chip design database systems to include support for analysis of interconnect parasitics. Our study on the performance of DIAS indicates that significant speedups can be obtained as compared to the earlier file-based approaches. To the best of our knowledge, this system represents the first object-oriented database-related work in the area of interconnect analysis.

Keywords: Object-Oriented Database, VLSI Design, Parasitics, SHORE, Performance

*This work was supported by grants from Texas Instruments (India) Private Limited and the Department of Science and Technology, Govt. of India.

1 Introduction

Right from the inception of integrated circuit fabrication in the early sixties, efforts have been made to improve chip performance, in terms of speed, density and power, by reducing the feature size. In fact, recent newspaper reports mention new chip manufacturing processes that have less than 10 nanometer resolution [24]. Due to increased complexity *interconnects* have begun to occupy a considerable portion of the layout in modern VLSI chips. As a result, the electrical characteristics of *interconnect parasitics*, which are unintended harmful side-effects that arise as a consequence of the layout, have to be fully considered in the design of an IC chip. This is borne out by recent studies which have found that the delay due to interconnects may cause upto 70 percent of the total delay [13, 17], thereby forming the primary bottleneck in chip performance.

In the chip design process, taking interconnect parasitics into account involves a four-stage process, as shown in Figure 1. In the first step, parasitic information is extracted from the original circuit design. In the next phase, these parasitics are represented using modeling techniques such as, for example, RC trees or transmission lines. This data is then incorporated into the circuit netlist. Finally, the refined circuit description is analyzed for electrical parameters such as timing, power dissipation, etc. Based on these results, the circuit design is revised and the interconnect analysis is repeated. This feedback process continues until a circuit with satisfactory electrical characteristics is realized.

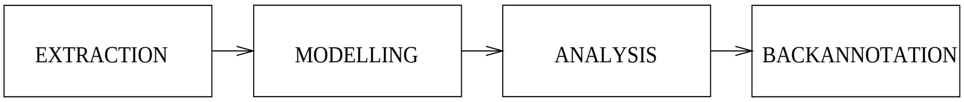

Figure 1: Interconnect Study

All of the above stages of interconnect study involve processing data that is both large in size and complex in nature. Current chip design systems typically adopt a custom-built *file-based* approach to managing their interconnect data. This approach is turning out to be unsatisfactory due to several reasons: First, the system does not provide a natural way of modeling complex objects. Second, even if only a part of the database is relevant to the processing, the whole database has to be scanned. Third, modifications to the file structure require rewriting of all applications. Fourth, extending the system to handle additional electrical characteristics is cumbersome. Fifth, there are no inbuilt mechanisms either for maintaining database consistency during concurrent access or for providing automatic crash recovery. Finally, these systems lack a well-defined query interface.

Some current chip design systems are not only file-based but are also *text-based* in that all data is stored in textual form. For these systems, the problems of the file-based approach are further exacerbated because of the following reasons: First, data processing is slow because of textual parsing. Second, the database is unnecessarily large since all data types are stored in textual format. Third, the entire database has to be parsed and stored in memory before data processing can be initiated.

From the above discussion, it is evident that current interconnect analysis solutions form a serious bottleneck in the chip design process in terms of both functionality and performance. This problem will become even worse as chip complexities increase. We feel, however, that effective data management can be provided in this environment by replacing the existing solutions with a full-fledged Database Management System (DBMS). In this scenario, various tools interact with the database system in each of the stages providing inputs as well as retrieving data to feed the next stage, as shown in Figure 1.

Using a database system provides many benefits: First, it serves as a single, central repository eliminating redundancy of multiple copies of the same data. Second, it stores the data in a tool independent form. Third, it provides an interface to the stored data which can be used to write specific applications irrespective of the form in which the underlying data is stored. Fourth, desirable features such as recovery of data, concurrency control and versioning are now the responsibility of the DBMS rather than that of application programs. Finally, special access methods can be used to query the data speedily.

1.1 The DIAS System

In this paper, we present an *object-oriented* model for representing interconnect parasitics. We also describe the design and implementation of **DIAS**, a database system that implements the model. DIAS has been developed for use in the Parasitic Work Bench process of Texas Instruments (India) Pvt. Ltd. but its design is broadly applicable to other environments as well.

DIAS is an object-oriented database solution and therefore allows users to naturally model complex objects. It uses the SHORE [23] toolkit as the backend database engine, and has been written using a combination of the Shore Data Language (SDL), a language-neutral notation for describing types of all persistent data, and C++, used for writing application code. The current system runs to approximately 40,000 lines of application code. Users of DIAS can interact with the system through either a Query Interface or a Tool Interface. The Query Interface is used for dynamically formulating queries while the Tool Interface is used for supporting existing (legacy) design tools that require a file-system interface, rather than a database interface. This is facilitated by the fact that Shore supports both database features and file system features. DIAS also provides a Programming Interface in the form of a library of routines that can be used to directly manipulate the database. These routines can be used for developing new design tools that directly use the database interface. For efficient processing and fast access of data, access methods such as B^+-tree indexing [6] and Path Index [4] have been incorporated. Issues related to object management and transaction processing are handled by the Shore server.

In summary, DIAS forms part of an information framework that extends chip design database systems to include support for analysis of interconnect parasitics. Our initial study on the performance of DIAS indicates that significant speedups can be obtained as compared to the file-based approach. To the best of our knowledge, this system represents the first database-related work in the area of interconnect analysis.

Due to space limitations, we only highlight a subset of the design and features of DIAS in this paper – the complete system description is available in [18].

2 Related Work

There has been considerable research in the area of data management for VLSI Computer Aided Design (CAD). The emphasis on making tools and data more accessible to designers have led to the *data oriented* design systems. Examples of VLSI CAD design database systems are OCT [11], Cbase [5], 3DIS [1], etc. Also, *tool oriented* systems emphasizing tool integration, for example, Cadweld [9], have been developed.

Data representation and data management are major components of electronic CAD *frameworks* [12, 10]. Framework standards have been developed by EDIF [7] and CFI [8]. These standards provide an information model and a programming interface. CAD databases such as DDB [20, 21] are based on the information model.

The systems considered above deal primarily with management of *design* data. In contrast, the focus of our work is on the complementary aspect of modeling and processing of *analysis* data.

3 Application Requirements

In this section, we outline the data handling requirements for VLSI CAD applications in general, and interconnect parasitics in particular.

Complex objects
: The designers and analysts of VLSI CAD deal with structurally and behaviorally complex objects. For example, a CAD application manipulates the circuit of an adder as a whole. Therefore, these objects should be the natural units of creation and access and the DBMS should be capable of efficiently modeling and handling large amounts of such complex data.

Support for hierarchies
: Usually, a complex design is composed of several subdesigns which in turn may be made of smaller components and so on till a primitive element is reached. The DBMS should therefore be capable of supporting hierarchies of arbitrary levels of nesting.

Uniform Representation
: A circuit is often designed and analyzed from several perspectives, such as *netlist*, *layout*, etc. Individual CAD applications frequently deal with data from a single viewpoint. This requires efficient translation of data from one format to another to facilitate intertool communication.

Fast storage and access
: The capability to incrementally access and modify small sections of the vast data is an important functionality. Frequently, users would want to deal with logical subunits of a complex design for ease of manipulation and speed of processing. The database system should support fast access of relevant data through efficient query mechanisms.

Support for legacy applications
: With the development of a new system, an immediate concern is the handling of existing data and the application programs that deal with them. For example, CAD simulation tools are typically file-based with specific formats. Very often, either the applications have to be rewritten or the data left in their original files. This problem could be avoided if the DBMS provided an interface for existing applications to interact in a seamless manner with legacy data in the new system.

4 Interconnect Object Model

Recent research in the field of databases has resulted in the emergence of new approaches such as *Object Oriented Database Management Systems* (OODBMS) based on the object model, which overcome most of the limitations of earlier database models. The object model is also particularly suited for our CAD application as it satisfies the requirements discussed in the previous section. As the first stage in the development of an interconnect parasitic database system, we designed an object model that captures all the information in the interconnect domain – this model is described in this section. In the following discussion, we use the popular Rumbaugh notation [19] for pictorially representing the model.

4.1 Classes

To model the complex data of the domain, we abstract common properties of entities in a *class*. Some of these classes are themselves abstract, in which case they are not directly instantiated. The *DiasObject* class represents any entity found in the domain. The *Property* class is used to denote a (name,value) pair which can be attached to any design object. A *Text* class is used to represent design and analysis management information.

The information that is stored in our DBMS is mainly concerned with *interconnect parasitics* data. Apart from this data, we also need to store netlist related data, layout related data, and characterization data. The details regarding the netlist are needed for storing the connectivity of the design. The characterization information is necessary for calculating parametric data. The layout geometries, location and other layout information are also required for calculating the interconnect parasitics and subsequently incorporating them as part of the designed circuit for analysis using simulation tools. We next describe the classes that we have designed for representing these various types of basic design data.

4.1.1 Basic Design Data

The class *Cell* is used to model the concept of a design entity. The classes *LibraryCell* and *Design* represent a leaf cell (or primitive element) and a complex design, respectively. The information about the transistors comprising a *LibraryCell* is modeled using the *Transistor* class. We also store the intrinsic delays suffered by a signal while traveling from the input port to the output port of a cell.

A *Design* object has several data associated with it. The data is divided into the following classes: *DesignParameter*, *ProgramStamp*, *Technology* and *Units*. The *DesignParameters* class stores information such as the date of creation, the directory in which it was created and so on. The *ProgramStamp* class has similar details about the program which created it. The *Technology* class has information regarding the technology to which the design belongs. The units for the various values used in the design are stored in an object of the *Units* class.

Other basic entities such as the ports of a cell and the nets in a design directly map to the classes *Net*, *Port*, etc. A collection of cells are grouped together based on some common characteristics such as the same technology. This grouping is done using the *Library* class.

"Instances" of *Cell* and *Design* classes are represented by the *Device* class. Note that the term "instance" here means a copy of another object (as is standard practice in VLSI design methodology) and not as an instance of an object type as in the context of programming languages [2]. In the rest of the paper, the usage of the term will be clear from the context.

A net could be physically divided into a number of disjoint interconnects. We make a distinction between two kinds of nets: the ones that are connected to the boundary ports of a cell and those which are fully internal. In case of the former, additional information regarding the external driver is stored. These fall into the *IONet* and *Signal* classes respectively.

The characterization data for cells are represented by two classes, the *PinInfo* and the *BufferChar* classes. The *PinInfo* class stores data about the pin-name, the type and input capacitance for all pins of a cell. The *BufferChar* class stores the rise time and delay time for different capacitance values.

4.1.2 Interconnect Parasitics Data

We now describe the major classes designed for parasitics related data. A net is logically divided into a number of Parasitic Nodes and Parasitic Devices. Therefore, the *Net* class is specialized into

ParasiticNode and *ParasiticDevice* classes. The parasitic effects between nets are represented by the *Coupling* class.

The interconnect parasitics information is modeled using electrical circuit equivalents. The following are some of the major types of interconnect models with varying levels of accuracy. We discuss how each of them is represented by means of classes.

- Single Lumped Capacitance
  This is the simplest electrical equivalent. In this model, an interconnect of a net is represented by a single capacitance from the interconnect to the ground. Thus, this ignores resistive effects. This is modeled by means of a *LumpedCapacitance* class which stores the capacitance value.

- Resistance Capacitance (RC) Network
  This includes the resistive effects also and hence a net is represented by means of an RC tree. An RC tree comprises of several RC elements, each element representing a section of an interconnect. We model the RC Network for a net by treating it as a graph modeled using the *Node* and *Edge* classes. A *Node* object is connected to a set of *Edge* objects. An object belonging to the *Edge* class stores the resistance and capacitance values of a section of the interconnect. In a model of a long-branched interconnect, thousands of such *Nodes* and *Edges* are present in an RC tree. This data tends to be too vast for analysis applications leading to delay in processing. As a result, at the cost of accuracy, the number of nodes and edges is often reduced.

- Transmission Line Model
  This is more accurate than the previous form since inductive effects of the interconnect parasitics are also taken into account. Thus, a single signal is represented by means of a Transmission Line model consisting of several uniformly distributed elements. Each such element is represented using an object of the *UnitElement* class, that stores the information regarding the resistance, capacitance, and inductance values for unit length of the transmission line.

The object model can be extended easily for other interconnect modeling techniques. The *InterconnectionModel* class serves as the abstract base class which is derived by all these models.

The effects of the interconnect parasitics on the performance of a circuit are modeled by the *ParametricInfo* class. These include timing, crosstalk and power effects. The information obtained at the *Prelayout* and *Postlayout* stages of a design are separately considered. Also, the effects are found at different granularities of the circuit such as in a path of the circuit, at a physical point, etc.

4.2 Relationships

Relationships among the entities in the domain are modeled using associations. A *link* is an instance of an association. We next discuss some special relationships among the entities and the modeling of these relationships.

4.2.1 Aggregations

An aggregation represents the "is-a-part-of" relationship between objects. An aggregation hierarchy is a hierarchy of instances. We now discuss the various aggregation hierarchies used to model the information in this domain.

- Design elements: A *Design* object is used to store a large amount of data. The parts of a design are represented by the *BasicParameters, ProgramStamp, Technology* and *Units* classes (Figure 4.2a).

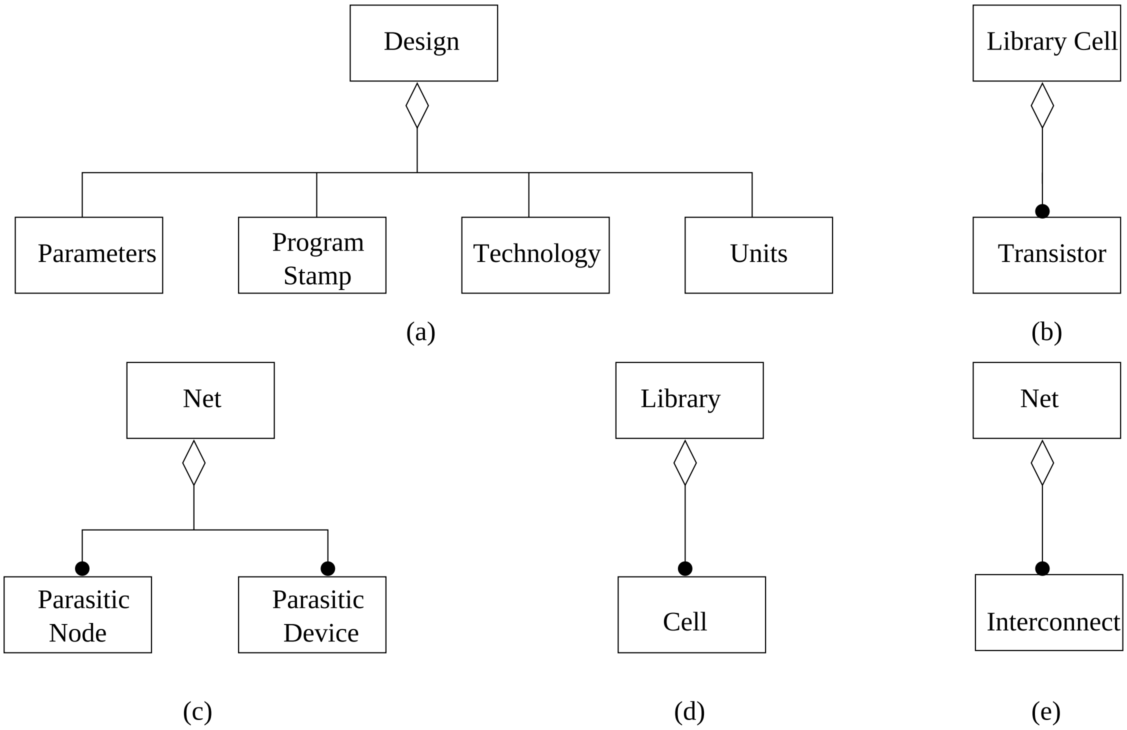

Figure 2: Example Aggregations in the DIAS object model

- Transistors: Though the *LibraryCell* is the leaf cell in a design hierarchy, we store details of the transistors also as an aggregation using the *Transistor* class (Figure 4.2b).
- Net hierarchies: A *Net* figures in two aggregation hierarchies. A net is physically composed of disjoint *Interconnect* objects of the layout (Figure 4.2c). From a parasitic viewpoint, the parasitic extraction can be done on the basis of breaking down a *Net class* into *ParasiticNode* class and *ParasiticDevice* class (Figure 4.2d).
- Collection: A collection is a logical grouping of classes having some common features. A *Library* class is a collection of *Cell* classes which have the same characterization information (Figure 4.2e).

4.2.2 Recursive Aggregations

A hierarchical design consists of several levels of nested subdesigns. A recursive aggregate contains, directly or indirectly, an instance of the same kind of the aggregate with an arbitrary number of such levels. In our case, the class *Cell* forms the superclass for two subclasses, *LibraryCell* and *Design*. The *LibraryCell* forms the primitive element of a design which cannot be subdivided further. The *Design* class is composed of *Device* class, which is an “instance” of a *Cell* class. A *Design* object is in turn composed of other *Device* objects and so on till a *LibraryCell* object. Here, the recursive aggregation is indirect, with the *Device* class forming the implicit link.

4.2.3 Homomorphism

A *Cell* has a set of *Port* objects which form the electrical interface to the external circuit. A *Device* is an "instance" of a *Cell*. A Device has *Terminal* objects which are "instances" of *Port* objects. The relationship between a *Device* and a *Terminal* maps to an equivalent relationship between corresponding entities, namely a *Cell* and a *Port*. This special relationship is called homomorphism. In general, a homomorphism involves four relationships among four classes, here the *Cell*, *Port*, *Device* and *Terminal* classes. The four relationships are:

- R: *Device* $\rightarrow$ *Cell*, a many to one relationship, *is-instance-of*
- S: *Terminal* $\rightarrow$ *Port*, a many to one relationship, *is-instance-of*
- T: *Cell* $\rightarrow$ *Port*, a one to many relationship, *has-ports*
- U: *Device* $\rightarrow$ *Terminal*, a one to many relationship, *has-terminals*

Then, the homomorphism maps links of relationship U into links of relationship T as a many to one mapping, that is, U(*Device*, *Terminal*) $\Rightarrow$ T(*Device.R*, *Terminal.S*)

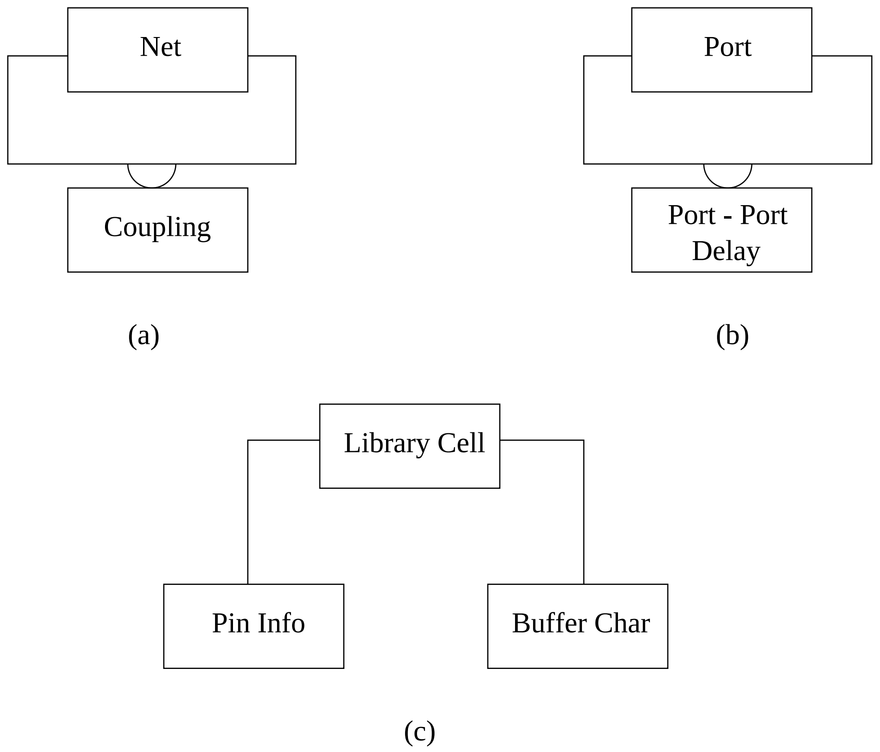

Figure 3: Example Associations in the DIAS object model

4.2.4 Other Relationships

We now discuss other kinds of associations among classes. Because of strong interdependence, we see many associations among classes. We briefly discuss the major ones here.

- *Inter-net Association:* The effect of the signal in one net causes parasitic effects in neighboring signals. This is represented by the *Coupling* class. It may be denoted by a uniformly distributed capacitance or a lumped one. The values of these are stored in a *Coupling* object (Figure 3a).

- *Layout Details:* Typical layout details such as width, length of wire and geometries are stored in the *LayoutDetails* class. A *LayoutDetails* class is associated with an *Interconnect* class for which it stores layout details.

- *Parametric Information:* The parametric information for a net is stored in the *ParametricInfoList* which has a list of *ParametricInfo* objects each belonging to a type derived multiply from three hierarchies. In case of parametric information between *Net* objects, the *ParametricInfoList* serves as a link connecting the two *Net* objects whose information is stored.

- *Port-Port Delay:* As noted earlier, a cell has a list of ports. The *Port-Port Delay* object is a link object connecting two *Port* objects. It stores information regarding the intrinsic delay suffered by a signal in traveling between the input port and the output port (Figure 3b).

- *Characterization Information:* Each *LibraryCell* is associated with a *BufferChar* class. The *BufferChar* object is a table of rows consisting of capacitance values and the corresponding rise and delay times. Additional characterization information is stored in the *PinInfo* class. It is also a table with rows consisting of values for the pin name, pin type and input capacitance for all pins of the cell (Figure 3c).

4.3 Attributes and Operations of Classes

The values of the attributes represent the state of an object and actually store the data. The attributes on the *Design* class store information about the design as a whole. The parasitic data is distributed among the attributes of the various classes that represent the structural interconnect model and the classes that represent parametric information.

Operations (methods) are used to modify and retrieve the state of an object. Simple operations get/set the attribute values. Others are used for querying. For example, given the name of a signal of a design, an operation retrieves the signal object.

4.4 Inheritance

Some of the classes defined so far exhibit similarities in their structure and behavior. The inheritance relationship defines a mechanism wherein one class refines the structure or behavior of other classes. The following inheritance hierarchies are noted in the object model. Figure 4 shows some of these hierarchies.

- *Base hierarchy:* The *GenericObject* class forms the base class for all objects. Design Management Information is stored in the form of free text in the *Text* class. A *Property* class is used to store one or more name, value pairs with any object. The refinements of the *GenericObject* form the basic entities of the design domain, namely *Net*, *Cell* and *Port* (Figure 4a).

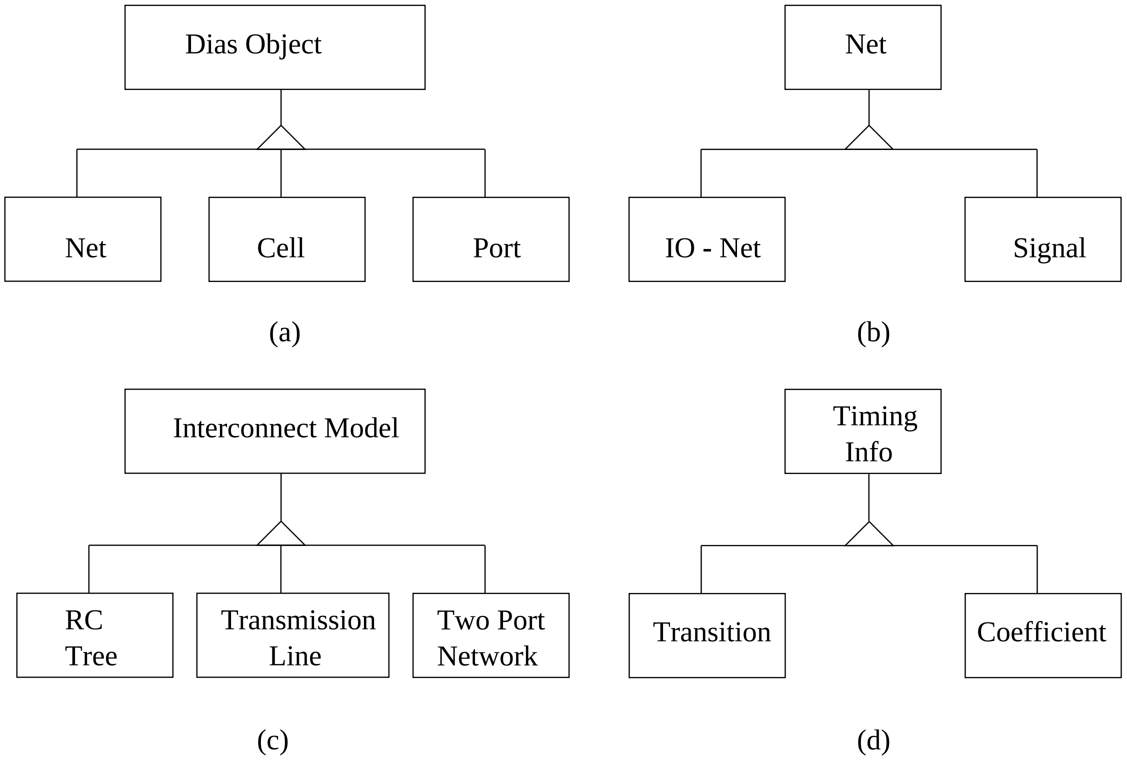

Figure 4: Example Inheritances in the DIAS object model

- *Derivations of Net class:* The *Net* base class is subsequently derived into *IO-Net* and *Signal* classes (Figure 4b). The *IO-Net* class represents those *Net* objects that connect *Terminal* objects of a *Device* with *Port* objects of the owning *Design* . On the other hand, *Signal* objects connect only *Terminal* objects of *Device* objects. In all other respects, the two are similar. In case of *IO-Net* objects, additional information regarding the driving net that acts as input from outside the interface of the *Design* object is stored.

- *Interconnect Model hierarchy:* The abstract base class *InterconnectModel* represents the structural interconnect model used for modeling interconnect parasitics. The specific model used forms a derivation of the base class. Examples are the widely used *RCTree*, *TransmissionLine* and *TwoPortNetwork* using y-parameters (Figure 4c).

- *Parametric Information:* The parametric information calculated using a specific interconnect model is represented using a *BaseNetComponent* class. The derivations of this class are the entire *Net* class itself, the *Terminal* class and *LeafSubcomponents* of *InterconnectModel* which map to say a *Node* or *Edge* of an *RC Tree.*

- *Timing Types:* For different timing related data to be easily represented, we have two derivations of the base class, the *Coefficient* class and the *Transition* class (Figure 4d). The *Transition* class is used to represent the different combinations arising among transitions of Low, High and Zero states such as LH, LZ, ZH, ZL etc. For each of these transitions, we store a triplet of Minimum, Nominal and Maximum values. The *Coefficient* class consists of a list of coefficient values which form the coefficients of an equation used to calculate the delay values.

The object model we have proposed attempts to satisfy the requirements mentioned earlier in Section 3. The aggregation and inheritance features help in representing the complex nature of the information directly in a natural manner.

5 System Architecture

In this section, we move on to describing the architecture of DIAS, which is shown in Figure 5. DIAS is built on top of **SHORE** (Scalable Heterogeneous Object Repository), a persistent object system developed at the University of Wisconsin [23]. We chose Shore as the backend engine for the following reasons: First, Shore supports both file system and object oriented features, thereby ensuring that legacy application tools can continue to operate in the new environment. Second, it allows for databases built by an application written in one language to be accessed and manipulated by applications written in other object-oriented languages as well. This feature is especially important in the VLSI CAD domain where tools are implemented in a variety of languages. Finally, Shore provides application independent features such as support for transactions and object management.

5.1 SHORE Server

The Shore server forms the lowest layer of DIAS. The Shore Storage Manager (SSM) is responsible for object management and transaction related features such as concurrency control and recovery. The Value Added Server (VAS) uses these features of the Storage Manager and presents an interface targeted at specific client applications. DIAS uses two such VASs: the Shore Value Added Server (SVAS) and the Shore NFS server. The SVAS is useful for applications using the object oriented database features mainly while the latter is useful for applications relying on the file system features of Shore.

5.2 DIAS Kernel

The DIAS kernel implements the object model described in Section 4 and uses the Shore server for supporting both object-based and file-based applications. It is responsible for query processing, for maintaining access methods, and for all the operations on the parasitic database. The access methods used in DIAS and the query processing strategies of DIAS are described in detail in Sections 7 and 8.

5.3 DIAS Interfaces

The DIAS system provides a variety of interfaces to support ad-hoc querying, legacy CAD tools and design of new CAD tools. These interfaces are called *Query Interface*, *Tool Interface* and *Programming Interface*, respectively, as shown in Figure 5.

The Query Interface is used to retrieve specific data from the DIAS database. Query processing is done based on the nature of the query. DIAS uses the SVAS and retrieves the objects that form the result of the query.

Existing file-based tools can interact with the data using the Tool Interface without any changes having to be made to the tool. For supporting this feature, the DIAS system uses the SHORE NFS value added server to provide the tool-specific view of the objects. This form of access is specially of use for migrating legacy data to the object store.

The Programming Interface forms a library of routines which abstracts away many of Shore's implementation dependent features to make it easier to write application programs. For example, this interface can be used to write applications that convert data from specific file formats to the DIAS format.

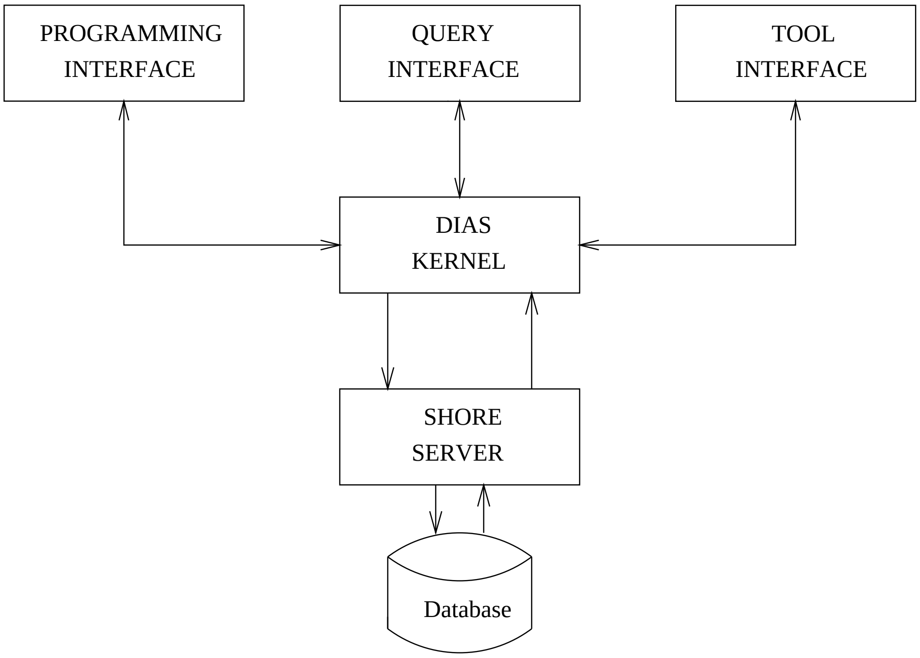

Figure 5: System Architecture

6 Model Implementation

We now outline how the object model described in Section 4 is implemented by the DIAS system: The classes of the object model are first converted into declarations in the Shore Data Language, SDL. Figure 6 shows a sample SDL declaration. The SDL compiler is used to generate the language binding in C++. The methods implementing the operation declarations are then written as part of the application in C++. The Shore runtime library is finally linked with the application.

```
interface Cell
  // This class holds all information that are associated with
  // both Library Cell and Design.
  {
  private:
    attribute string    CellName;                  // name of the cell
    attribute set<Port> Ports;                     // ports of the cell
    attribute CellType  Type;                      // type of the cell
  public:
    void       Construct(in lref<char> NewName); // initializing function
    void       SetName(in lref<char> NewName);    // set the cell name
    lref<char> GetName();                          // get the cell name
    short      NumberPorts() ;                     // get port count
    ref<Port>  GetPort(in short PortIndex);        // get a specific port
    CellType   GetType() ;                         // get cell type
    CellType   SetType(in CellType NewType) ;      // set cell type
    ref<Delay> FindPath(in ref<Port> Input,
                     in ref<Port> Output);         // get delay between two ports
    ref<Port>  GetPort(in lref<char> Name) ;       // get port with given name
    void       AddPort(in ref<Port> NewPort);      // add a new port
  };
```

Figure 6: Sample SDL declaration

7 Access Methods

In object oriented database systems, query processing throws up challenges that were non-existent in relational query processing. For example, object oriented query processing systems need to support queries on objects forming a hierarchy. The hierarchy itself maybe a result of specialization or aggregation and queries based on each of these hierarchies need to be processed differently. New access methods are required to efficiently traverse such hierarchies while retrieving embedded objects. We describe a representative set of these special methods below.

7.1 Inheritance Hierarchy Access Methods

Queries on inheritance hierarchies may be against a particular class in the hierarchy or against the hierarchy tree itself. The main index structures proposed for such queries include CH-trees [14] and H-trees [15]. CH-trees are known to perform well when the target of a query is the hierarchy tree itself. On the other hand, H-trees perform well when the target of the query is a particular class in the hierarchy. A variation of these trees is the hcC-tree [22] (hierarchy class chain tree).

The hcC tree is based on the B^+-tree and attempts to efficiently process both class-based queries and tree-based queries. The organization of the non-leaf nodes is similar to that of B^+-trees - the only modification is that they contain an additional bitmap that provides advance information to prevent an unnecessary traversal of the tree till the leaf, thereby speeding up evaluation of queries. The leaf nodes consist of the key value, a set of pointers each pointing to the list containing the instances of a particular class in the hierarchy, and a pointer pointing to the list of instances of all classes in the hierarchy. In other words, the structures maintain information both on the individual classes in the hierarchy and also on the entire tree structure itself thus making it possible for both class-based as well as hierarchy-based queries to be answered efficiently.

7.2 Aggregation Hierarchy Access Methods

A number of indexing techniques that support efficient evaluation of queries which involve aggregation graphs have been proposed [3]. In our domain, the configuration for aggregation hierarchies assumes more importance as these hierarchies have many more levels of nesting than inheritance hierarchies. Hence, we have considered a variety of index organizations. In the following discussion, we refer to the object at the beginning of a path instantiation as the root of the hierarchy and the object the end of the path instantiation as the leaf of the hierarchy.

An aggregation hierarchy is rooted at one object and a *forward* (top down) traversal of the hierarchy is required to extract an object at the leaf of the hierarchy in the absence of any access method. It can be seen that a path exists for the navigational retrieval of the required object. A potentially more difficult problem is to have a reference to an object at the leaf of the hierarchy and then try to find the root object of the hierarchy which requires a *reverse* (bottom up) traversal of the aggregation hierarchy.

Access methods suggested for aggregation hierarchies include *nested index* and *path index* mechanisms [4]. The nested index uses a modified B^+-tree organization. The nested index maintains references to the root and the leaf of the aggregation hierarchy in question. This association between the root and the leaf of the aggregation hierarchy facilitates retrieval of the reference to one end of the hierarchy if the other is determined. It can be seen that retrievals are cheap. However, updates cause problems in the management of the structure. An object updated anywhere along the aggregation graph implies two forward traversals and one reverse traversal of the graph to update the index, making updates costly. The nested index thus requires reverse references to be maintained.

The path index, on the other hand, maintains a reference to all objects along an aggregation hierarchy. In this case, updates do not require any reverse traversal since the reference to the root exists in the path index itself. For queries that do not require path information, the path index comes with an overhead since more leaf nodes may require to be read to process the query. Thus, the choice of the index depends to a great extent on the nature of the queries.

7.3 DIAS Access Methods

Based on the above discussion, we have incorporated the following access methods in the DIAS system: B^+-tree, hcC tree, nested index and path index. The B^+-tree is already provided by Shore, whereas the remaining access methods have been implemented by us.

8 Query Processing

To facilitate ad-hoc querying, DIAS incorporates a Query Processing System (QPS) that provides a query language interface that is based on the Object Query Language (OQL) specification [16]. The query interface implements the basic features of OQL, and is sufficiently powerful to express queries with multiple constraints. Nesting of queries is also permitted. The system also handles parametrized types in an efficient manner. The query processing system uses a unified query model to express query semantics, thereby enabling sophisticated semantic checking and processing strategies.

The complete details of the QPS architecture and implementation are available in [18] – due to space limitations, they are not included here. We restrict the discussion in this section to identify the components of typical DIAS queries. In particular, we identify access methods for each component and describe how it is evaluated. It should be noted that a typical query may incorporate combinations of the identified components.

8.1 Simple Value-based queries

Handling simple value-based queries entails extraction of an object-based on a simple attribute value. An example query is

List the names of all signals used in the design FPU_206N.

In answering this query, it is sufficient to locate the design instance FPU_206N of a floating point unit design. To speed up execution of the above query, a B^+-tree index is maintained on the design objects for attribute *DesignName*. Once the reference to the design object is obtained, signals used in the design are listed.

8.2 Inheritance-based queries

Inheritance-based queries require the traversal of inheritance hierarchies and identification of all instances along the inheritance path that satisfy the query predicate. An example query is

List all the Nets which have fanout greater than 17.

The *Net* class specializes into *IO-Net* class and *Signal* class and hence the query processing engine needs to locate all the objects of these classes which have the required fanout. A *hcC*-tree maintained on the inheritance hierarchy is used to identify all objects in the hierarchy satisfying the said predicate.

8.3 Aggregation-based queries

Aggregation-based queries require either forward or reverse traversals of the aggregation hierarchy. An example of a forward traversal aggregation-based query is

List the source to load RC delays due to terminal RPO.IN5.OUT_3
in the signal SIM5AND_IN of the design FPU_206N.

The query processing engine needs to locate the design FPU_206N in the database, navigate the object reference to the particular signal SIM5AND_IN, pick the particular source, RPO.IN5.OUT_3, and then list the delays contained in the *Timing* objects for all the loads. An example of a reverse traversal-based aggregation query is

List all signals in design FPU_206N which suffer slew greater than 5 ns and their sources.

This query needs to access objects which form neither the root nor the leaf of the hierarchy. This necessitates use of the path index on the path

$$Design.Signal.Source.Timing.Value$$

The query is processed by first identifying by *Timing* objects satisfying the constraint provided in the query. Once this is done, the path index is used to get the respective signal names. The source terminals can also be obtained with the *same lookup* since the path index indexes all objects in an aggregation path.
For queries targeted at the end points of a path, however, the nested index is more appropriate since it eliminates the overhead of maintaining references to all instances along a path. An example of such a query is

List all signals with an effective capacitance greater than 10 pF.

A nested index keyed on capacitance values establishes direct connections with the respective signal objects. Thus, signals satisfying the above query for all designs in the database can be retrieved efficiently.

9 Performance

The DIAS system is currently operational and we have conducted a set of tests to evaluate its performance. These tests were made using real design data samples (provided by Texas Instruments) on a Sparc20 machine running Solaris 2.4.

Design	FileSize(MB)	Terminals	Signals	Load Time	Dump Time
Design1	1.3	542	212	2:09	4:08
Design2	2.0	945	349	3:24	8:52
Design3	3.1	1583	542	7:16	16:11
Design4	5.2	2830	889	14:00	30:10

Table 1: Load and dump times for varying design sizes (in minutes:seconds)

Our first experiment investigated the time taken to load data from legacy text-based design files into the DIAS database, and the time taken to subsequently dump (retrieve) the entire design contents from this database. For this purpose, translators developed using the Programming Interface of DIAS were used. The translators also monitored system resource requirements in term of disk space and main memory.

We evaluated the load time and dump time for four different designs that represent varying degrees of complexity. These results are presented in Table 1, which shows the designs, the size of the text files that contain their parasitic information, the number of terminals, the number of signals and the corresponding load and dump times. From these results, we observe that the times taken for the transfer of data in either direction are relatively small compared to the overall time taken for all the four stages of interconnect analysis, which is typically in the order of several hours. Thus, DIAS provides efficient mechanisms for bulk transfer of data to and from the system. (The dump times are higher than the corresponding loading times due to asynchronous I/O in the cases of writes.)

In our second experiment, we evaluated the time taken by DIAS to answer a representative set of queries on the various designs. We compared these times against the times taken to answer the same queries when executed on the textual versions of the designs. This helped us to quantitatively evaluate the benefits, if any, of moving from the current file-based system to the DIAS system.

The results of this experiment are summarized in Table 2. They show that DIAS provides a

Design	FileSize(MB)	DIAS System	File (Text)
Design1	1.3	0.057	4.00
Design2	2.0	0.057	6.29
Design3	3.1	0.063	11.29
Design4	5.2	0.066	18.71

Table 2: Query times for varying design sizes (in seconds)

significant improvement in query response time as compared to the file-based system.[1] The main reason for the performance improvement is that in the file-based system, accessing an object requires sequentially parsing the file until the desired object is located. This scan can take considerable amount of time depending on the size of the file and the position of the required object in the file. In contrast, in DIAS, data objects can be directly accessed by utilizing the appropriate access methods. We expect that these performance differences will increase even further for more complex designs.

10 Conclusions

In this paper, we have addressed the issue of data management for VLSI interconnect parasitics. We identified the requirements for management of this data and proposed an object model for parasitic data. We also presented the architecture of DIAS, an object-oriented database system that implements the proposed model.

DIAS allows users to naturally model the complex objects encountered in the interconnect parasitic domain. It supports a variety of interfaces to cater to on-line querying, legacy tools and the development of new tools. A variety of access methods that are specially designed for efficient access in object-oriented databases have been implemented. Sample performance numbers indicate that DIAS performs considerably better than its file-based counterparts. To the best of our knowledge, this system represents the first work in using database technology for the analysis of VLSI designs.

Acknowledgements

We thank the members of the Parasitic Work Bench team from the Design Automation Division at Texas Instruments (India) Private Limited for their assistance in providing domain knowledge.

References

[1] H. Afsarmanesh, D. Knapp, D. McLeod and A. Parker, "An Extensible Object-Oriented Approach to Databases for VLSI/CAD", *Proc. of VLDB Conference*, August 1985.

[2] D. Batory and W. Kim, "Modelling Concepts for VLSI CAD Objects", *ACM Trans. on Database Systems*, March 1985.

[3] E. Bertino, "A Survey of Indexing Techniques for Object-Oriented Database Management Systems", In *Query Processing for Advanced Database Systems*, Morgan Kaufmann, 1994.

[1] The results shown for the DIAS system include transaction overheads and thus represent system response time.

[4] E. Bertino and W. Kim, "Indexing Techniques for queries on nested objects", *IEEE Trans. on Knowledge and Data Engineering*, 1989.

[5] M. Breurer, W. Cheng, R. Gupta, I. Hardonag, E. Horowitz and S. Lin, "Cbase 1.0: A CAD database for VLSI circuits using Object Oriented technology", *Proc. of ICCAD*, 1988.

[6] D. Comer, "The Ubiquitous B-tree", *ACM Computing Surveys*, June 1979.

[7] Electronic Industries Association EDIF Steering Committee, "EDIF Information Model", version 3 0 0, 1993.

[8] CAD Framework Initiative, "Design Representation Programming Interface", version 1.4, 1994.

[9] J. Daniell and S. Director, "An object oriented approach to CAD tool control", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, June 1991.

[10] R. Gupta et al., "An Object Oriented VLSI CAD Framework - A Case Study in Rapid Prototyping", *IEEE Computer*, May 1989.

[11] D. Harrison, P. Moore, R. Spickelmier and A. Newton, "Data management and graphics editing in the Berkeley design environment", *Proc. of ICCAD*, 1986.

[12] D. Harrison, A. Newton, R. Spickelmier and T. Barnes, "Electronic CAD Frameworks", *Proc. of the IEEE*, February 1990.

[13] S. Khanna, "Sorting out Signal Integrity", *Electron. Engin. Times*, June 1991.

[14] W. Kim, K. Kim and A. Dale, "Indexing Techniques for Object Oriented Databases", In *Object-Oriented Concepts, Databases and Applications*, Addison-Wesley (ACM Press), 1989.

[15] C. Low, B. Ooi and H. Lu, "A Dynamic Associative Search Index For OODB", *Proc. of ACM SIGMOD*, 1992.

[16] R. Cattell, *The Object Database Standard: ODMG-93*, Morgan Kaufmann, 1994.

[17] C. Ratzlaff and L. Pillage, "RICE: Rapid Interconnect Evaluation using AWE", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, June 1994.

[18] *DIAS: An Object-Oriented Database for Interconnect Analysis*, Project Report PR-97-02, Database Systems Lab, SERC, Indian Institute of Science.

[19] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.

[20] A. Singhal, R. Arlein and C. Lo, "DDB: An Object Oriented Design Data Manager for VLSI CAD", *Proc. of ACM SIGMOD*, 1993.

[21] A. Singhal, N. Parikh, D. Dutt and C. Lo, "A Data Model and Architecture for VLSI/CAD Databases", *Proc. of ICCAD*, 1989.

[22] B. Sreenath and S. Seshadri, "The hcC-tree: An Efficient Index Structure For Object Oriented Databases", *Proc. of VLDB Conference*, 1994.

[23] M. Zwilling et al., "On Shoring Up Persistent Applications", *Proc. of ACM SIGMOD*, 1994.

[24] IEEE India Bulletin, "Newsletter of IEEE India Council", April 1996.