# A Memory Distribution Mechanism for Object Oriented Applications

*Prakash Arunachalam, Craig Chase and Jacob Abraham*

**TR TR-PDS-1995-006 1995**

**Parallel & Distributed Systems group**
**Department of Electrical & Computer Engineering**
**University of Texas at Austin**
**Austin, Texas 78712**

# A Memory Distribution Mechanism for Object Oriented Applications*

Prakash Arunachalam Craig Chase Jacob Abraham

Parallel and Distributed Systems Laboratory
email: pdslab@ece.utexas.edu
Electrical and Computer Engineering Department
The University of Texas at Austin,
Austin, TX 78712

January 4, 1996

**Abstract**

Many applications, particularly in Computer-Aided Design (CAD), require large amounts of memory, limiting the size of problems which can be handled. This paper presents a new mechanism which exploits the large amount of memory available in a cluster of workstations for programs that are designed in an object-oriented manner. All the memory required for each object may be allocated in other machines on the network, and any references to the object return the desired item from the appropriate machine in a transparent fashion. Experimental results are provided to demonstrate the effectiveness of this approach for a particular memory-intensive CAD application.

## 1 Introduction

As computers become more and more powerful, demands follow for larger and larger applications to be run on them. There has been a lot of work in speeding up computation intensive applications through the use of parallel and distributed processing. The related problem of dealing with memory intensive applications has received somewhat less attention. Many problems in Computer Aided Design, particularly those involving searches of large state spaces such as test generation and formal verification, are extremely memory intensive. Because of this, it is becoming very difficult to deal with the sizes of circuits currently being designed.

*Virtual Memory* has been a simple solution to such problems, since disk memory is cheap and main memory is not. Unfortunately, if the working set of memory pages for a program is large, the time required to deal with page faults significantly impacts the performance of the application. Although disk capacities have been increasing due to advances in technology, access times have not been keeping pace; this situation is not likely to improve in the near future.

Wood [19] proposes distributed computing as a good solution to such problems, since the total amount of expensive main memory in a network tends to be quite large. Unlike trends in disk access time, network speeds have been increasing dramatically, and new technologies such as fiber optics promise extremely high network bandwidths. This will allow access to the distributed memory faster than swapping pages from disk.

*Supported in part by the Semiconductor Research Corporation under contract 94-DP-388 and by NSF grant 9409736. This paper appeared in the Seventh IEEE Symposium on Parallel and Distributed Processing, October 1995

Even with today's technology, memory distribution can be viable for some applications. Our current implementation on standard 10Mbps ethernet permits access to a remote object in less than 5ms. This figure compares with disk access latencies on the order of 10ms. Provided the distribution mechanism permits the appropriate degree of locality for an application, memory distribution can result in improved performance over a virtual memory implementation.

In this paper we describe a mechanism for memory distribution on a network of workstations. We identify the key issues for such a mechanism: identifying the objects to distribute, implementing accesses to remote objects, and determining a partitioning strategy that achieves good locality of reference. We demonstrate the effectiveness of our mechanism on a practical application in the CAD area, Test Knowledge Extraction for large circuits. Our experimental results indicate that memory distribution is successful for this application.

Our techniques are generic and can be applied to other memory-intensive applications of similar structure. We base our mechanism on the Object-Oriented Programming (OOP) paradigm. Manipulation of an object is possible only by invoking one of a small set of operations for that object. Hence, OOP is well suited for message-passing environments where objects may be distributed over several workstations. Smalltalk, which was one of the first object oriented languages was based entirely upon message passing [9]. To operate on a remote object, the operation and its arguments are encoded in message and sent to the machine on which that object resides. The remote machine applies the operation to the object and places return values (if any) in a response message that is sent to the originating machine.

A common characteristic of object-oriented, memory-intensive applications is the presence of large data structures containing many instances of a smaller object. These data structures, commonly called *container classes*, provide a natural means upon which to base the partitioning of the program's memory requirements. For example, the Test Knowledge Extraction application utilizes a large tree data structure. To distribute the objects contained in this tree, our implementation must simply redefine the notion of a pointer to a node in the tree. After memory distribution, a pointer becomes a pair, (process ID, address). Traversal of the tree and manipulation of objects in the tree are now easily accomplished via message passing.

Our memory distribution mechanism is orthogonal to parallelization techniques, which may also result in partitioning the application over several workstations. If one or more partitions require more memory than is available in a workstation, then memory distribution can be used on those partitions. An example is the use of Binary Decision Diagrams (BDDs) in formal verification [3] where the individual BDDs can be operated upon concurrently. Each of these BDDs can potentially take up a large amount of space and hence might require memory distribution to provide the increased memory space.

The remainder of this paper is organized as follows. Section 2 provides an overview of the new mechanism for increasing memory. Section 3 describes an application to which this mechanism has been applied. Section 4 presents the results obtained from an implementation of the mechanism. Section 5 concludes the paper.

## 2 A Memory Distribution Mechanism for OOP

All large problems can be dealt with in one of two ways:

- Abstraction - This deals with the system at a higher level and uses details to reduce the size of the model being checked.
- Decomposition - This breaks up the model into smaller pieces and works with the smaller pieces.

In the past, both these methods have been used quite satisfactorily to solve a number of problems. A number of features such as non-determinism can be used in the abstraction process. The decomposition of the model can be done in many ways. VLSI systems exhibit a great deal of concurrency, which can be observed

in their operation. Some examples of ways to parallelize the solution are to utilize the hierarchical nature of the model, or the steps in the algorithm to decompose the design. Unfortunately, both abstraction and decomposition have been applied in a manner which requires a very detailed understanding of the design of the system. There are very few generic mechanisms which can perform these two operations in a satisfactory manner. The result is that *ad-hoc* solutions have been developed for various classes of problems.

We seek a relatively generic mechanism which can be used to distribute the memory requirements of an object oriented program over a set of interconnected workstations. We assume a single thread of control throughout execution of the program. Control is transfered to a machine whenever one of the objects on that machine is accessed. Obviously, if the application permitted concurrent access to objects, the constraint of a single-thread of control could be relaxed. However, we do not consider parallel threads in this paper.

We identify three elements of a successful memory distribution mechanism:

1. A means to determine which objects should be partitioned.
2. A mechanism for transparently applying operations to remote objects.
3. A policy which determines how to distribute the objects among the machines.

We describe each of these elements in the subsections that follow.

## 2.1 Memory Profiling

Clearly, identifying the objects to be partitioned is very important for satisfactory performance. This can be accomplished either through the intuition of the programmer, or by collecting a memory-use profile. This latter approach is likely to be more generally applicable, and we describe a mechanism for memory profiling in this section. We illustrate our technique through the example of the Test Knowledge Extraction application.

A profile of the memory can be collected in a number of ways. For our purposes, we find out how many times an instance of each type of object (i.e., *class*) is created and destroyed. Table 1 shows a listing of the contribution of each class to the overall memory used by the Test Knowledge Extraction application for the module *ctl_lgc2* of the 8085 microprocessor.

From this table, it is obvious that the class *Port* is the best candidate for memory distribution. Since 93% of the memory is consumed by this class alone, distribution of this object will dramatically reduce the memory requirements of any single processor. In general, memory profiling should be performed on several different sets of input data (different circuits in our case) to accurately identify which classes to partition. In some cases, several classes may be identified as candidates for distribution as a result of memory profiling.

Our profiler has a number of important features:

- It can determine the dynamic memory requirements of the application.
- It can be selectively applied to all or only part of an application. This feature permits individual program modules to be profiled (and ultimately distributed) independently of other modules.
- It is generic and can be easily ported to any C++ program. [1]

Note that a class that consumes a lot of memory is not necessarily a good candidate for distribution. If that object is accessed frequently, then the performance after memory distribution may be abysmal. We adopted a simple heuristic to model this effect. If $C$ is the number of times an object of that class is created and $k$ is the average number of times each object is accessed, then the best object chosen will be such that maximizes

[1] Our implementation was based on the C++ programming language. Implementation in other object oriented languages may also be possible

| *Classes* | *Created* | *Destroyed* | Memory | %Memory |
|---|---|---|---|---|
| ComplexMode | 0 | 0 | 0 | 0 |
| Constraint | 463 | 0 | 9260 | 0.266119 |
| Expression | 2004 | 0 | 64128 | 1.84295 |
| Function | 0 | 0 | 0 | 0 |
| Mode | 36 | 0 | 1440 | 0.0413835 |
| Module | 1 | 0 | 0 | 0 |
| Operator | 0 | 0 | 0 | 0 |
| Optree | 223 | 0 | 9812 | 0.281983 |
| Port | 22553 | 0 | 3247632 | 93.3322 |
| RepMode | 4 | 0 | 192 | 0.0055178 |
| SimpleMode | 32 | 0 | 1408 | 0.0404639 |
| Statement | 2332 | 0 | 27984 | 0.804219 |
| SymbolicValue | 0 | 0 | 0 | 0 |
| Term | 8540 | 156 | 67072 | 1.92755 |
| term_const | 192 | 0 | 2304 | 0.0662136 |
| term_port | 3285 | 0 | 39420 | 1.13287 |
| term_sym | 0 | 0 | 0 | 0 |
| term_value | 706 | 0 | 8472 | 0.243473 |
| TFSolution | 71 | 13 | 464 | 0.0133347 |

Table 1: An Example of the Output of Memory Profiling

the function: $C(1 - wk)$ for an appropriately selected weight $w$. The value for $w$ should be chosen based upon the relative speed of the network. We have found that $w = 0.025$ is effective in our experience. For faster networks like FDDI, this factor would be smaller.

Essentially, we have modeled the cost of referencing an object as a constant, irrespective of the strategy used to actually distribute the object (see Section 2.3). This simple heuristic will clearly not select the optimal object for partitioning in all cases. However we have found it effective in practice at identifying *good* candidates.

## 2.2 Implementation of the mechanism

The memory distribution mechanism is based on the concept of a distributed pointer. A distributed pointer consists of the process index and the memory location within that process. Those objects which have been distributed in other machines can only be accessed by a distributed pointer.

The implementation replicates a portion of the application code in all the machines. This code contains mechanisms for:

- creation and destruction of objects
- invoking operations on the objects
- transfer of control to other machines

Whenever a distributed object is created, memory is allocated in one of the machines of the cluster and a distributed pointer to that object is returned to the creating machine. Initially, a "host" machine has control

of the entire application. When a distributed object is accessed with a method, a transfer of control occurs to the specific machine on which that object is located. After the operation is complete the result is returned to the host machine.

Typically, applications organize large collections of objects in data structures often referred to as container classes. Consider, for example, a large tree data structure. The nodes of the tree are among the objects to be distributed. During a traversal of the tree, control must to be returned to the host only after the entire tree has been traversed. Therefore, the number of messages can be dramatically reduced by considering the structure of the tree when distributing objects. We discuss this optimization in the Section 2.3.

The mechanism for operating on a distributed object is similar in semantics to Remote Procedure Calls (RPC). An efficient implementation can be transparently introduced into the application by overriding operators on the distributed classes. Our implementation utilizes the socket mechanism of SunOS. The PVM [5] environment was originally tried, however the high cost of message passing in PVM ultimately led to the decision to use the lower-level socket interface.

### 2.3 Object Distribution

Locality of reference to objects is critically important to obtaining good performance. It is considerably faster to access a local object than one across the network. The *Locality Number* is defined as the percentage of time that a reference to an object can be resolved locally. We use this metric as an indication of the quality of the object distribution strategy. Note that we assume a flat topology for the network. Thus, maximizing locality of reference reduces to the graph partitioning problem which is known to be NP-complete [7]. As an optimal partitioning is likely to be intractable, we have developed simple heuristics which give good results for some practical applications. Also note that since distributed objects may be dynamically allocated, the distribution problem must be solved dynamically at execution time.

We considered two partitioning methods. The first is a random partitioning, where the objects are placed randomly into the distributed system subject to maintaining an approximately equal number of objects on each machine. Even this simple method can be effective if the cost of performing an operation is large with respect to the cost of performing the transfer of control.

The second method is a *guided* partitioning, where the objects are partitioned into the system using information gained from the containing data structure for those objects. The Figure 3 gives a simple view of how guided partitioning works.

Guided partitioning works on a tree data structure as follows. A limit is set on the number of objects which can be placed on a single machine. This limit is adjusted dynamically as the number of objects in the system changes. Until the limit is reached, children of any node in the tree are placed on the same machine as their parent. If the limit has been reached, then the location for the new object is chosen randomly. Guided partitioning can be extended to virtually any type of container class, such as linked lists or graphs. All that is required is that a traversal order be apparent from the structure of the container. This order then provides the basis for identifying which objects should be located within the same machine.

A similar methodology was suggested by Mueller-Thuns et al. [12]. There, the strongly-connected components were extracted from the graph representing the connectivity of an integrated circuit. Each strongly connected component was located within a single processor.

## 3 Automatic Test Knowledge Extractor

The Automatic Test Knowledge Extractor (ATKET) [16] is a software package which automatically extracts high level information of a circuit. This information helps in guiding the search process and speeding up test generation. The high level knowledge about the behavior of a system (available from the specifications)

```
Partitioner(V)
    if root
      add to new machine
      number = 1
      add to list(child(V))
      while(list)
         pop list(V1)
         Partitioner(V1)
    else if(number<limit)
      add to same machine
      number = number+1
      add to list(child(V))
    else
      V = root
      Partitioner(V)
```

Figure 1: Heuristic for Guided Partitioning

can be used to combine module tests into high quality tests for complex circuits. An important part of this package was the design and implementation of a procedural interface that provides easy access to the test knowledge database. The VHDL [14] description of the circuit is used to obtain the knowledge.

The central data structure of ATKET is the Module Operation Tree (MOT) to capture the behavior of the modules in a design [15]. Test knowledge is generated using the MOTs and is represented by the various test "modes" of interest. A mode contains a set of constraints on either module inputs or primary outputs that need to be satisfied in each time frame to produce some effect on the outputs of a module. Each constraint in a mode is represented by a two tuple (E, B) where E is the VHDL conditional expression and B is the boolean value that indicates the validity of E.

The MOT is a binary tree in which the nodes correspond to VHDL conditional statements. Each node has a left child which corresponds to the conditional statement being false and a right child which corresponds to the conditional statement being true. A typical example of the MOT of a counter is shown in the Figure 2.

Depending upon the module input-output relationship, edges in the MOT are categorized so that information about the leaf nodes are captured. A combination of these categories can also be used to categorize edges in the MOT. The MOT is built for every *process* statement. The VHDL conditional constructs within the *process* statement are analyzed to generate the MOT.

# 4 Results

The techniques described above were implemented in C++ on a network of SPARC-2 workstations. The ATKET application was modified to use memory distribution and then run on various modules from the 8085 microprocessor circuit. Each processor had a main memory of 32Mbytes. The g++ compiler version 2.5.8 was used with default optimizations enabled ("-O"). All experiments were performed when the machines were in multi-user mode.

Among the more promising results of the experimentation is that some circuits which could not have been analyzed on a single machine, were succesfully analyzed using the memory distribution mechanism. Some of the examples are *reg_ctrl, regpad* and *alulogic* modules of the 8085. Table 2 shows the reduction in memory on the host machine due to the memory distribution mechanism. It can be seen that the relative improvement is dependent on the size of the memory consumed. This result is explained by realizing that

```
entity COUNTER is
  port(CLK: in bit;
       C_RST: in bit;
       LD: in bit;
       CIN: in INTEGER range 0 to 4095;
       COUT: out INTEGER range 0 to 4095);
end counter;
architecture behavior of COUNTER is
  begin
       process
          variable C_REG: INTEGER range 0 to 4095;
          begin
            if(C_RST = '1') then
              C_REG = '1';
            elsif(CLK'Last_Value = '0') then
              if(CLK = '0') then
                C_REG := C_REG;
              elsif(LD = '1') then
                C_REG := CIN;
              elsif(C_REG = 500) then
                C_REG := 0;
              elsif(C_REG <= 3000) then
                C_REG := C_REG + 12;
              else
                C_REG := C_REG - 2999;
              end if;
            else
              C_REG := C_REG;
            end if;
       end process;
end behavior;
```

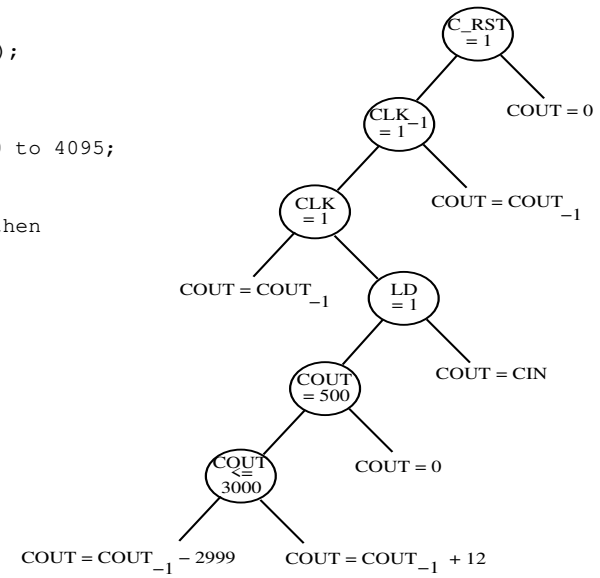


Figure 2: MOT of a Counter

the overhead of the replicated code becomes less significant as more distributed objects are present during execution.

| Circuits | Data memory used (host machine)(Kb) | | Improvement |
|---|---|---|---|
| | without partitioning | with partitioning | Factor |
| sys | 29660 | 14084 | 2.106 |
| inst_reg | 82700 | 38524 | 2.147 |
| dataaddr | 95720 | 44228 | 2.164 |
| ctl_lgc1 | 218172 | 62756 | 3.477 |

Table 2: Memory allocation with and without partitioning

The execution times of various modules of the 8085 microprocessor in different configurations is shown in Table 3. The '*' implies that this configuration had insufficient memory to perform the knowledge extraction on the specific circuit. It can be seen that for the dataaddr module, the time taken by the distributed mechanism is faster than that of the single processor. Three of the modules could not be analyzed by the uniprocessor configuration but could be analyzed by other multiprocessor configurations.

The amount of memory taken up by each processor varies greatly due to the partitioning. The *Variance* gives an idea of how much variation the distribution has from an uniform distribution. A random partitioning has small variance. A guided partitioning, however, trades off increased variance to gain greater locality. Some of the results show that the variance of the memory in a guided partitioning increases with increased number of machines.

It was seen that the locality number depended on the mechanism of partitioning. Table 5 gives the comparison of the locality number of some examples with both random and with guided partitioning. While

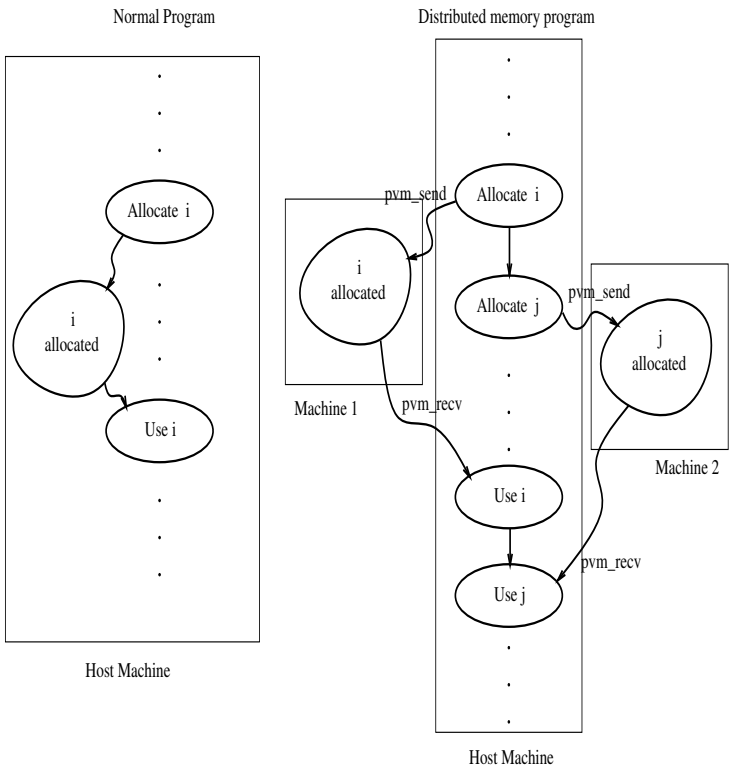


Figure 3: Distributed Memory Partitioning

locality depends upon the circuit being tested, guided partitioning always produced greater locality than random partitioning.

# 5 Conclusions

We present a mechanism for distributing the memory requirements of an application across a network of workstations. We base our mechanism on the object oriented programming paradigm. The mechanism is particularly effective when a bulk of the memory is consumed by objects in large container classes.

We demonstrate the mechanism by applying it to an important problem from the CAD area, Test Knowledge Extraction. A key result is that memory distribution enables analysis of circuits that could not be analyzed on a single workstation.

Identifying the appropriate objects for memory distribution is facilitated through the use of memory profiling. We demonstrate that the integration of a memory profiler into an object oriented application can be easily accomplished by overloading the operators for allocating and deallocating memory. A key in making memory distribution efficient is achieving good locality of reference. Our experiments indicate that simple heuristics can result in very effective distribution of the objects. These heuristics can be based on the referencing patterns of container classes.

Although actual speedup was obtained only in one case, we feel that memory distribution has great promise for the future based on the trend of increasing network speeds. With current technology, memory distribution can be effective for very large applications where execution on a single workstation may not even be possible.

| Circuits | Number of Processors | | | | | |
|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 |
| exf | 2.8 | 18.6 | 12.5 | 9.4 | 7.2 | 6.8 |
| inst_reg | 8.5 | 20.8 | 17.4 | 14.9 | 12.6 | 11.5 |
| dataaddr | 7.4 | 25.4 | 15.3 | 10.6 | 7.8 | 5.5 |
| regpad | * | * | 62.1 | 55.2 | 53.1 | 49.6 |
| alulogic | * | * | 88.7 | 74.2 | 65.2 | 58.7 |
| reg_ctrl | * | * | * | 92.4 | 78.5 | 61.3 |

Table 3: Time taken (minutes)

| Circuits | Number of Processors | | | | |
|---|---|---|---|---|---|
| | 2 | 3 | 4 | 5 | 6 |
| dataaddr | 10.2 | 12.6 | 13.8 | 14.2 | 14.8 |
| exf | 10.8 | 13.1 | 15.3 | 16.7 | 17.3 |
| inst_reg | 10.4 | 13.5 | 15.8 | 17.1 | 18.5 |

Table 4: Variance in memory with multiple processors

# 6 Acknowledgments

We would like to thank Praveen Vishakantiah for his code of ATKET and for useful technical discussions, without which this work would not have been possible. Thanks also to Ron Brashear for his suggestions and ideas.

# References

[1] H. B. Bakoglu, *Circuits, Interconnections, and Packaging for VLSI*, Menlo Park, CA: Addison-Wesley, 1990.

[2] D. Benson *et al.*, "Silicon multichip modules," *presented at the Hot Chips Symposium 3*, Santa Clara, CA, Aug. 1990.

[3] R. Bryant, "Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams," *ACM Computing Surveys, Vol. 24, No. 3*, pp. 293-318, September 1992.

[4] R. D. Chamberlain and M. A. Franklin, "Analysis of parallel mixed-mode simulation algorithms," *Proceedings of the 5th International Parallel Processing Symposium*, pp. 155-160, 1991.

[5] A. Geist et. al., "PVM: Parallel Virtual Machine, A User's guide for Network Parallel Computing," 1994.

[6] *Libraries and Tools for Object Parallel Programming*, Proc. CNRS-NSF Workshop on Environments and Tools for Parallel Scientific Computing, St. Hilaire du Touvet, France, Elsevier, Advances in Parallel Computing, Vol. 6, pp. 231-246, 1993.

| Circuits | Type of Partitioning | |
|---|---|---|
| | Random | Guided |
| exf | 0.671 | 0.894 |
| dataaddr | 0.725 | 0.927 |
| inst_reg | 0.703 | 0.911 |
| alulogic | 0.683 | 0.922 |

Table 5: Locality Numbers with different partitions


[7] M. R. Garey and D. S. Johnson, *Computers and Interactability - A Guide to the theory of NP-Completeness*, W. H. Freeman & Co., 1979.

[8] G. Geschwind and R. M. Clary, "Multichip Modules - An overview," in it Expo SMT/HiDEP 1990 Tech. Proc., San Jose, CA, 1990, pp. 319-329.

[9] A. Goldberg, *Smalltalk-80: the language and its implementation*, Reading, Mass., Addison-Wesley, 1983.

[10] S. Karthik, and J. A. Abraham, "A Framework for Distributed VLSI Simulation on a network of workstations," *Proceedings of the International Conference on Computer Design*, pp. ??, October 1992.

[11] S. A. Kravitz and B. D. Ackland, "Static vs. dynamic partitioning of circuits of a MOS timing simulator on a message-passing processor," *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 136-140, 1988.

[12] R. B. Mueller-Thuns, D. G. Saab, R. F. Damiano and J. A. Abraham, "VLSI logic and fault simulation on general-purpose parallel computers," *IEEE Transactions on Computer-Aided Design*, pp. 446-460, March 1993.

[13] S. P. Smith, B. Underwood and J. Newman, "An analysis of parallel logic simulation on several architectures," *Proceedings of the 1987 International Conference on Parallel Processing*, pp. 664-667, 1987.

[14] D. Perry, *VHDL*, New York, NY, McGraw-Hill, 1991.

[15] P. Vishakantiah, J. A. Abraham, and M. Abadir, "Automatic Test Knowledge Extraction from VHDL (ATKET)," *Proceedings of the 29th iEEE/ACM Design Automation Conference,* pp. 273-278, June 1992.

[16] P. Vishakantiah, and J. A. Abraham, "High Level Testability Analysis Using VHDL Descriptions," *Proceedings of the European Design Automation Conference*, pp. 170-174, February 1993.

[17] R. K. Watts, Ed., *Submicron Integrated Circuits*. New York: Wiley, 1989.

[18] N. Weste and K. Eshragian, *Principles of CMOS VLSI Design: A Systems Perspective*, Reading, MA: Addison-Wesley, 1988.

[19] D. A. Wood and M. D. Hill, "Cost-Effective Parallel Computing", *IEEE Computer*, vol. 28, No. 2, pp. 69-72, February 1995.