A Memory Distribution Mechanism for Object Oriented Applications*

Prakash Arunachalam, Craig Chase and Jacob Abraham

Computer Engineering Research Center
The University of Texas at Austin
Austin, Texas 78712

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. 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 [12] 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.

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 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. 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 applications which are memory-intensive is the presence of large data structures containing many instances of a smaller object. These data structures, commonly called *container objects*, 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

*This research was supported by the Semiconductor Research Corporation under contract 94-DP-388 and NSF grant 9409736 at the University of Texas at Austin

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.

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

We seek a relatively generic mechanism that 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.

2.1 Implementation of the mechanism

Our 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 that have been distributed in other machines can only be accessed by a distributed pointer.

Our 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 completed 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, distributing a large tree data structure. 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 communication paradigm of Berkeley Unix.

2.2 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 Ratio* 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 [4]. 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. For example, the primary container object for our knowledge extraction application is an unbalanced tree. By placing entire subtrees on the same processor good locality can be achieved.

3 Automatic Test Knowledge Extractor

The Automatic Test Knowledge Extractor (ATKET) [9] is a software package which extracts information from the description of a circuit in a hardware description language like VHDL [8]. This information helps in guiding the search process and speeding up test generation. The high level knowledge about the behavior of a system 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 central data structure of ATKET is the Module Operation Tree (MOT) that captures the behavior of the modules in a design. 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 2-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. 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.

Table 1 shows the reduction in memory achieved through memory distribution for ATKET run on several modules from the 8085 microprocessor. The table reports the maximum memory consumed by any single process when the application is distributed over four machines. Assuming four identical machines, our results suggest that memory distribution more than doubles the effective memory capacity for this application.

Circuits	Memory Used (Kb)		Ratio
	single	distributed	
sys	29660	14084	2.106
inst_reg	82700	38524	2.147
dataaddr	95720	44228	2.164
ctl_lgc1	218172	62756	3.477

Table 1: Memory allocation with and without partitioning

The execution times of various modules of the 8085 microprocessor in different configurations is shown in Table 2. The '*' indicates that this configuration had insufficient memory to perform the knowledge extraction on the specific circuit. Three of the modules could not be analyzed by the uniprocessor configuration but could be analyzed by other multiprocessor configurations. These results were obtained with guided partitioning. Table 3 contrasts the locality ratios for random and guided partitioning. As expected, guided partitioning resulted in significantly better locality than simple random partitioning.

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 2: Time taken (minutes)

It should be noted that speedup was not our principal objective in distributing ATKET. This application is memory bound in most situations, and our goal was simply to execute it on the more complex modules of the 8085. However speedup was obtained for the dataaddr module when distributed over six processors.

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 3: Locality Numbers with different partitions

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. 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.

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

[2] 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.

[3] *Libraries and Tools for Object Parallel Programming*, Elsevier, Advances in Parallel Computing, Vol. 6, pp. 231-246, 1993.

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

[5] 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.

[6] 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. 508-511, October 1992.

[7] 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.

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

[9] 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.

[10] 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.

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

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