Object-oriented Computer Architectures for New Generation of Applications

Ramesh K. Karne
Computer and Information Sciences
Towson State Universtiy
Towson, Maryland 21204

karne@midget.towson.edu

Abstract

Since the inception of von-Neumann architecture for computer design, there has been no new paradigms or revolutions in computer architectures. Computer applications have been increasing at an exponential rate, however, the basic computer architectures remained the same. The conventional computer architectures, which are based on primitive building blocks including arithmetic logic units, floating point processor units, logical shift units, and register file units created tremendous *sematic-gap* and inefficiencies in information system processing. It is about time to revisit the standard von-Neumann computation model and argue about its efficiencies, as we are entering into a new era of information processing where applications don't have any boundaries in computation, communication, and information storage.

In this paper, we propose a revolutionary computer architecture which avoids the *semantic-gap* and inefficiencies, and is based on an object-oriented paradigm to provide the benefits of abstraction, inheritance, hierarchy, modularity, extensibility, and polymorphism. We will describe the fundamental building blocks for this architecture and propose a possible approach for implementing these new generation of computers which will not make software and hardware obsolete before coming to existence. We will present the design issues related to such architectures and research directions needed to study the feasibility of these architectures.

1.0 Motivation

In the current global market, industry and government agencies are driven by a well recognized common theme, that is: low cost, high quality, reduced development time, fast turnaround time, time-to-market, plug-and-play systems, and open architecture. Virtually every product and produce that exists today demands the above theme. In order to achieve the global market objectives, researchers are constantly looking for new paradigms which will help to compete and succeed in constantly changing world. During last three decades, most notable concept, the Object-oriented paradigm, came to existence to answer some of these issues in computer programming. The object-oriented programming offers unique properties which may be adapted to other domains[12] to achieve similar advantages. Our motivation stem from this notion and we believe that it is possible to apply the distinct characteristics of object-oriented programming to other application domains. The object-oriented programming paradigm offers features such as abstraction, encapsulation, inheritance, reduced code, reusability, modularity, extensibility, fast development time, portability, open architecture, and hierarchy. These features of object-oriented programming if applied to other domains may reduce obsolescence, provide extensibility and reusability.

In information systems, computer applications are growing at an exponential rate and the underlying computer architectures are basically stuck at 16 bit, 32 bit, and CISC/RISC computer architectures. In addition, the computer hardware and software manufacturers are producing products which become obsolete

before they serve their purpose in life. For example, in last decade, the Intel processor based PCs with respect to hardware include chips based on 8086, 80186, 80286, 80386, 80486, Pentium, and P6. In addition, there are many other chips used in PCs that are based on different speeds, features, and other characteristics. How about the Intel processor based operating systems? Not too long ago, we use to keep counting DOS releases from 1.0 through 7.0, and now the DOS operating systems are history, and new operating systems such as Windows, NT, Windows-95, Cairo and other seems to penetrate in to the market place. In addition to hardware and operating systems, the application programs come and go on a daily basis, and most of them hardly had a chance to be known to the users. For example, the word processing software, spread sheets software, database software, and other applications migrated through dozens of versions and releases, where even an expert programmer or an analyst will have hard time to keep track of these moving target releases.

The characteristics that are synonymous with software versus hardware; cheaper, easier to implement, flexible to modifications, and portable across hardware and software platforms, are not valid any more as the software costs are sky rocketing and hardware costs are plummeting. During the past three decades, the hardware costs reduced by half every two years and also the density of hardware doubled in the same period of time. This trend is still continuing and one must take advantage of this trend and develop computer architectures that are suitable to new emerging applications. The new generation of applications are more communication intensive rather than computation intensive. The information superhighway era will demand processors that can communicate faster than they can compute. These processors will be moving gega or even tera-bytes of data over the network and it requires tremendous data search capabilities, data transmission capabilities, and data management capabilities. In addition, the data is multi-media type and requires different processing capabilities based on different applications. It does not make sense to treat this data as 1's and 0's any more as the data is context sensitive and it has semantic meaning during processing.

In addition to rapid hardware and software migration, today's information technology becomes obsolete and wasteful. In particular, we are wasting most valuable time and skills of programmers, engineers, scientists, and other information systems professionals to learn and relearn to perform the same tasks. In today's market, any programmer with more than five years of experience is considered as an obsolete professional and requires to relearn the new skills to successfully land on to the next job. It is even questionable now that, whether the computers are providing more productivity, or are they wasting more resources? It will be a daunting problem to assess the waste we produce in information technology including hardware, software, personnel, skills, retraining, and unused code!

Considering the above issues, we believe that we desperately need a computer architecture which allows information systems to serve their life span, preserve people's skills and knowledge, and adapt to new applications.

2.0 Introduction

Consider the number of computer applications in the world! It is conceivable that there may be at least a million or more applications that can be classified and they are possibly distinct from each other. Most of these applications are executed on machines that are based on conventional von-Neumann architecture, and the engines for these machines are constructed from primitive hardware building blocks including arithmetic logic units (ALUs), logical shifter unit (LSUs), register file units (RFUs), and so on. It is evident that the hardware and the applications are apples and oranges and they have enormous *semantic-gap*[13]. Ideally, one can classify computer applications and their uses[6] into four categories: applications, programming logic, machine logic, and electronic circuits. These four categories can be viewed either as four layers if they are implemented separately or can be viewed as a single layer if they are implemented in the hardware (electronic circuits). Each implementation strategy has its own pros and cons with respect to flexibility,

programmability, and also implementability. Ideally, if it is practical, one could implement each application with a dedicated hardware, then there is no *sematic-gap* and they are called application-oriented systems. Some of the applications such as image processing, and fast-fourier transforms have been implemented as dedicated architectures or application oriented architectures (AOA).

In order to reduce the *semantic-gap* some researchers tried to build computers based on programming languages such as Lisp[1,7,8], and Prolog[14]. Some computer applications such as simulation[9,10,15] have proven to run very efficiently on dedicated machines. Similarly, some database applications[11,18] are proven to run very efficiently on dedicated database engines. However, there are no efforts made to develop computer architectures which exploit the inherent characteristics of computer applications and consequently map these applications to underlying architectures.

At higher levels of abstraction, we can represent the computer architectures using a cone as shown in Figure 1. The next generation of operating systems[17], communication systems and networks[3,4], and database systems[2,5] clearly indicate that the object-oriented technology[16] is penetrating through all the layers of the computer cone except the hardware. In this paper, we extend the penetration of object-oriented paradigm into **"hardware"**, by forcing the hardware blocks to follow the principles of object-oriented programming (OOP).

In order to penetrate object-oriented programming features into hardware, it requires unusual ways of implementation and mind-set for building new generations of systems. Consider an example with respect to a personal computer, in last decade there are at least ten generations of PCs starting with Intel's 8086 processor to a Pentium processor. Every time a new model arrives, the old model is usually trashed as most of the components of the old PC are incompatible with the new PC. Ideally, one should take the building blocks (hardware and software elements) and only replace or plug-and-play the modules as required to build a new system. The computer systems should mimic the engineering construction that is possible to build or renovate a house. One could take a 30 year old house, and renovate it by extending, replacing, or renovating the features as needed without destroying the whole house. The building blocks (objects) for the house, have not changed over the years, as higher levels of abstractions were made long time ago. For example, bedrooms, bathrooms, tiles, flooring, roof, patio, deck, kitchen, and so on are well established house objects and they are plug-and-play objects available to build new homes or modify the existing homes. We must learn to build information systems similar to building houses in an object-oriented fashion to save time, money, and people's skills.

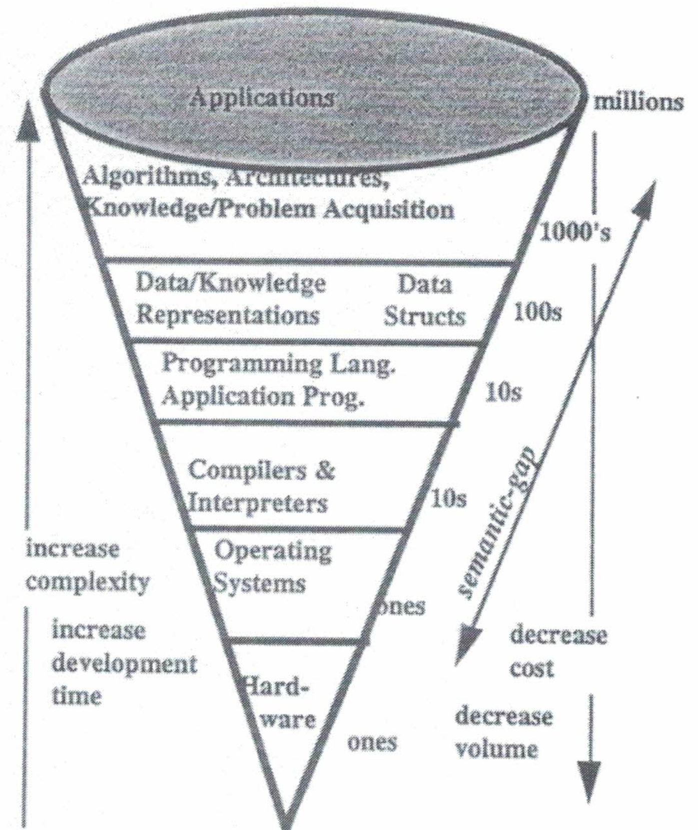

Figure 1: Computer System Hierarchy

In this paper, we will propose a novel solution to this problem, however, it is a conceptual study at this point, and the solution is based on an object-oriented paradigm. Object-oriented paradigm applied to hardware architecture implies that the object-oriented hardware behaves analogous to object-oriented programs. We believe that it is possible to build computer systems based on object-oriented paradigm,

however, there is a need for more ***non-mainstream*** research in this area, which requires revolutionary approaches in VLSI designs and technology.

The rest of the paper is organized as follows. The object-oriented programming paradigm and its features are described in section 3.0. Section 4.0 enumerates the object-oriented architecture and provides analogy between object-oriented software and hardware. Some of the implementation approaches and suggestions are provided in section 5.0. The design issues related to object-oriented computer architectures are provided in section 6.0. Finally, the section 7.0 captures the summary of this paper including our future research directions.

3.0 Object-oriented Programming Paradigm

The fundamental difference in object-oriented programming (OOP) versus conventional programming can be stated as follows. In conventional programming, the data and control are separated and there is a no easy way to associate and derive data from control. In OOP, the data and control are merged together to form an object, the interface to the object is clearly specified, and in addition, the access to the object can also be controlled by the creator of the object. This object structure offers numerous benefits and provides a robust building block for object-oriented programs. Some of the features of OOP are briefly discussed as follows.

Abstraction is the fundamental feature in OOP. Before one starts writing programs, there is an object-oriented analysis (OOA) phase where abstractions are made to obtain object classes to capture the target application. For example, in case of a stack, after abstraction is made, one can come up with stack class which is very generic to this example application. Abstraction helps to reduce redundancy in the program, and also reduces the number of data structures needed to implement the application. Once an abstraction is made, then there can be classes (templates) which captures the properties of group of objects, and objects are created as needed, these created objects are instances of a given class. Stack, linked list, window, message, node, link, bill of material, process, rule, and file, are some of the examples of objects derived from higher levels of abstraction.

Encapsulation allows OOP to integrate control and data into object thus hiding all the details of the object in the object itself. The data is encapsulated into the object and only the methods or control mechanisms provided for the object can allow or disallow to access the data. Thus, in addition to integrating data and control into the object which avoids complexity in programming, it also provides the additional security feature where the data access can also be controlled in the definition. Access mechanisms such as Private, Protected, and Public are generally used to accomplish security. In stack example, the data is encapsulated in the object and the push, pop, and empty methods are the only means of accessing the stack information. Even the pointer data is also encapsulated in the stack object.

Inheritance is deriving new objects from other objects. The new objects inherit the properties of its parent objects thus reducing the code and need for creating new objects. The derived objects can have new properties in addition to the inherited properties. This allows the derived objects to be specialized and parent objects to be generalized. Thus, one can create an hierarchy of objects and reduce the amount of code required to redefine the data and member functions that are required at each level.

Polymorphism is creating functions that have many forms and postponing the function invocation and its binding to run time (late binding). This allows, parent objects to invoke derived object's member functions selectively at run time. Thus, polymorphism provides virtual function interface which will allow late binding and provide full flexibility in program extensions.

Extensibility is the most significant property of object-oriented programming. If proper abstractions are made during the object-oriented analysis, then it is possible to extend

the existing programs by merely adding inherited objects, or creating new objects and writing the new code for the derived or new objects. The new code has no impact on the existing code, thus providing the extendibility of programs.

Modularity is provided by the object-oriented programming because the objects have the property of encapsulation and the only way to communicate to objects is through its designed interface. Thus, the software development can be modular by adding new objects or inheriting from the existing objects without making modification to the existing objects.

Typechecking is a concept where the operations performed on objects must be of the same data type. This avoids run time errors and the type checking problems are caught at compile time.

Abstraction, encapsulation, inheritance, polymorphism, extensibility, modularity, and type checking are the key characteristics of object-oriented programming. These software concepts of object-oriented programming can be used for building hardware architectures as illustrated in the next section.

4.0 Object-oriented Computer Architecture

The object-oriented computer architecture (OOCA) proposed here is based on an object-oriented programming paradigm, which implies, that the hardware architecture should support all the characteristics of object-oriented programming (OOP). The object-oriented programming referred here does not assume a particular existing programming language as none of the existing languages support the functionality required for the proposed OOCA, however, some language extensions can be made to C++ programming language to be used for OOCA systems. The Figure 2 illustrates a system view of OOCA.

A computer architecture is usually defined in terms of building blocks, assembly of these building blocks results in a computer system which can be used to run computer applications. In a typical von-Neumann architecture, a central processing unit, a memory unit, an input/output unit , and a control unit constitute the major building blocks that can be used to build a computer system. This computer system being built by the above building blocks has no imbedded knowledge of any applications that can be executed on this system. Consequently, it creates *semantic-gap* between applications and architectures. The OOCA closes this gap by integrating applications and architectures with object-oriented paradigm as shown in Figure 2. Thus, the applications are represented with objects (abstract entities) and the architecture matches the object-oriented representation. Furthermore, the implementation of the architecture to realize a computer system also matches the object-oriented functions and representations.

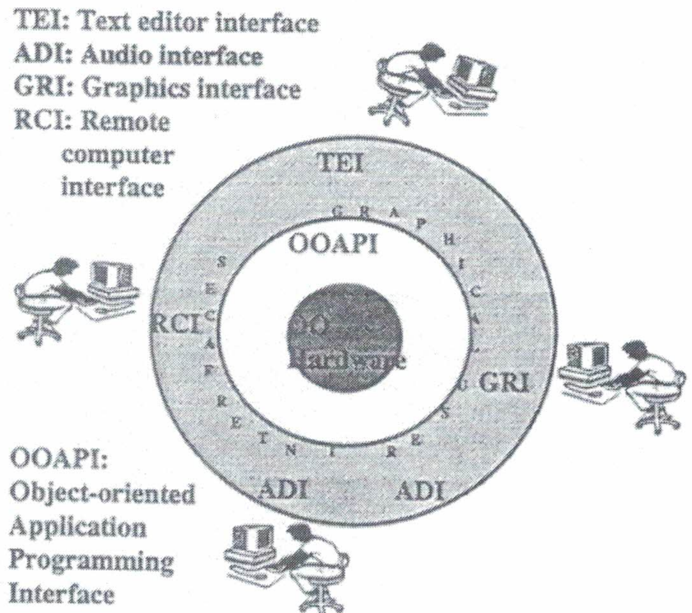

Figure 2: System View

How could we architect such OOCA which closes the *sematic-gap* from application to hardware implementation? We try to address this question with a possible approach and a top-down analytical study as presented in this paper. These concepts needs further investigation and evaluation to ultimately build computers that are based on OOCA.

We assume that all applications that are intended to run on OOCA systems can be mapped into object-oriented programs. The

applications can be single user, multi-user, single system, or distributed systems, as all of these cases are plausible in the real world environment. The applications require user interfaces as shown in Figure 2. The user must have all the above capabilities and be able to run applications such as word processing, spreadsheets, database, communications, general purpose computing, scientific computing, accounting, banking, and other. From these variety of computer applications, it is evident that there are numerous objects that can be created by the analyst which are unique to its application domain and the computer system must deal with all the operations required with these objects. For example, a word processing application may create a file object, which needs to be edited, spell checked, saved, distributed to other users, combined with other objects, and so on. Thus, the computer paradigm from computation turns into a different paradigm which includes object creation, object migration, object communication, object management, object security, object storage, and object address resolution.

A conceptual view of the proposed OOCA is shown in Figure 3. We briefly describe the architecture features in the following sub sections.

4.1 OO Characteristics in OOCA

Abstraction can be used to capture basic world object classes (or templates) and stored in memory and can be reused in memory. That means, these classes in hardware must have all the capabilities of an object-oriented programming. For example, a class stored in memory must be able to be instantiated as objects and further stored in memory as objects. This indicates that memory objects can be instantiated after the memory is designed. In hardware terms, we need an object memory which can interact with class memory and generates objects during the execution. Thus, the memory cells must be designed as objects and allow dynamic reconfiguration of cells to adapt to different types of object classes. If we can design memories in this fashion, object classes, object instantiations, and object operations can be performed directly in memory and there is no need to fetch these objects from memory to processor unit to perform operations.

Encapsulation can be used to hide the data and member functions in the object itself. That means, the objects in memory contain data as well as its operations to control the object. Thus, it is convenient to perform object operations in memory if memory technology allows such facilities. Some of the type of object operations may include: object creation, object update, object invocation, and object destruction. If object operations are performed in memory, there is a need for object execution control information along with data and member functions, which can be encapsulated right into the object as shown in Figure 4. This object execution control is called object mini kernel (OMK), which is like an operating system code and also similar to a microcode attached to the object. Thus, the object memory needs to be flexible to perform object operations and must be dynamically re-configurable under the control of OMK.

Inheritance can be used in the computer hardware to derive new classes from existing classes as needed. Inheritance operation is done on object classes and thus the object class memory must be able to derive new classes from existing classes. In order to derive inherited objects, inherited classes are instantiated in object memory. Thus, there is a need for close interaction between class memory, and object memory.

Polymorphism is the many forms of the same function and the ability to choose a function at run time. This feature must be implemented in object memory and dynamic links are created between the objects in the object memory to simulate polymorphism. How these links are created and managed in dynamic memory is a great challenge for memory designers!

Often there is no way to predict what functions are needed at abstraction time as the application may be evolving and not all requirements are collected at the conceptual level. Thus, there can be some virtual functions associated with the objects at the abstraction time and later on these functions may be implemented by the inherited objects. This allows the parent objects to invoke children's member functions

as needed and hence allowing polymorphism to occur. With respect to hardware, the pre-defined objects in memory can invoke functions related to the inherited objects and proper address links and communication in memory must be set up to achieve polymorphism.

Extensibility and modularity is the property of object-oriented paradigm and with respect to OOCA, it is naturally provided as we organize the OOCA in object-oriented manner. For example, we can add new classes in class memory with out modifying the existing classes. Using inheritance, we can derive new classes from existing classes and specialize these classes as needed. Similarly, we can create new objects by instantiating the existing or new classes from class memory.

The OOCA must embrace inheritance, encapsulation, and polymorphism. The abstractions should be made by the user unless we implement some knowledge of abstraction into the objects which is an interesting research area to explore in the future.

4.2 Overview OOCA

The object-oriented computer architecture proposed here is based on numerous revolutionary concepts and paradigms. First, it is based on an object-oriented paradigm which implies that all the features of object-oriented programming as discussed in section 3.0 are adopted in OOCA. The architecture is also based on an application oriented architecture as each application is modeled as an object-oriented program and in addition it is mapped onto object-oriented hardware. Finally, the objects that are defined at an application level are mapped into class and object memory and they are manipulated directly in memory.

The architecture can interface with a communication network and the object migration, distribution, and management can be performed over the network. The distributed nature of the system is transparent to the user. The standard architectures such as CORBA, and OLE can be used to achieve the distributed function of the OOCA.

P7

We need an object-oriented programming language (OOPL) to communicate with the systems built using the OOCA. This language should provide user-friendly ergonomically designed interface and must take advantage of the architecture features. We envision that, the OOPL, may be similar to current programming languages such as C++ and Smalltalk, however, the language may be at higher level to hide the details of OOCA.

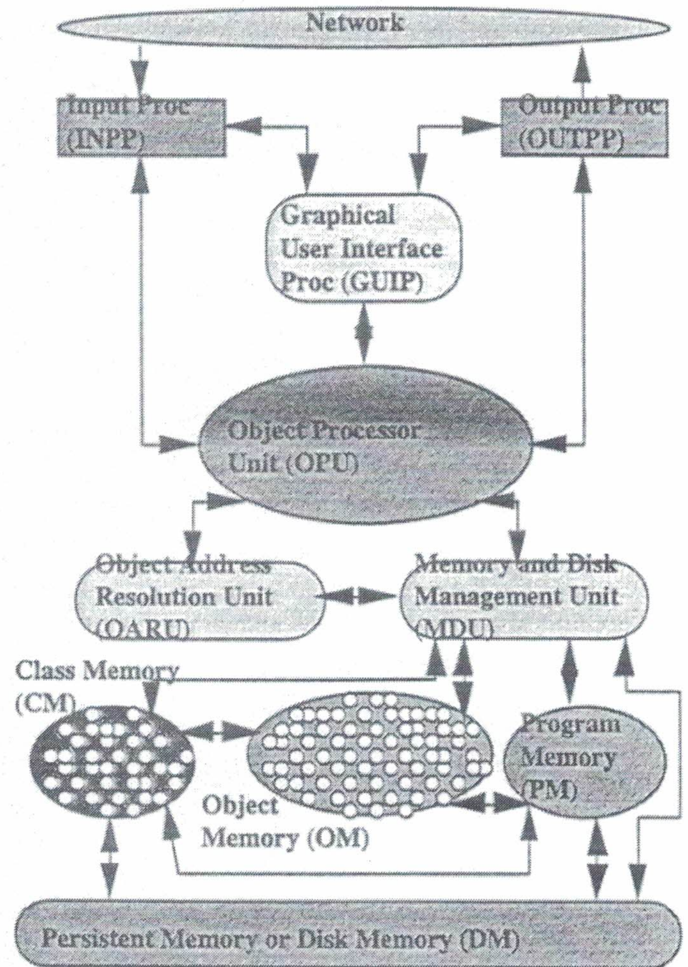

Figure 3: Object-oriented Computer Architecture (OOCA)

4.3 Memories in OOCA

The fundamental difference between OOCA and conventional architecture is the memory organization, design, and technology. The objects (real world entities) have unique address in the real world environment, irrespective of where they reside in the system. That means, objects in memory, disk, or on the network share the common address space. As we have captured the whole world of

applications into object oriented programming, and the number of objects required will be considerably less than the number of addresses required in a conventional programming. Thus, it is conceivable to have a single address space for the world of objects. Furthermore, one could classify these objects based on domains of applications to provide a large address space for each domain.

There are four basic types of memory units in OOCA with uniform addressing scheme. The class memory (CM) stores and manages object classes, object memory (OM) stores and manages objects, program memory (PM) stores and manages the user's application programs (no data), and disk memory (DM) stores and manages objects and classes stored as mass storage. Thus, CM, OM, and PM are real memory units and DM is the disk memory. Traditional virtual memory techniques can be used to manage real and virtual memory in the system.

The object address resolution unit (OARU), resolves all the object addresses and provides this address to memory and disk management unit (MDU). The MDU in turn controls all the memory operations and interfaces with CM, OM, PM, and DM units. These memory units except the DM, can perform all object operations internal to the memory.

The class memory and object memory consists of classes and objects respectively. However, for each object, there is a member function program which is common to all the objects. This program can be loaded in the program memory during the execution time and thus there is need for close interaction between OM and PM.

4.4 Object Processor Unit

The object processor unit (OPU) is similar to the central processing unit (CPU) in a conventional computer. However, the OPU acts as a control unit to drive the operations that can be executed in CM, OM, and PM. OPU monitors the control sequence of a program which is in PM, generates object addresses, and interfaces with OARU to resolve these addresses. In addition, the OPU interfaces with input processor, output processor, graphical user interface processor, and MDU.

4.5 Input/Output, and GUIs

The input processor (INPP) interfaces with the network to fetch objects and store them either in memory or in user interface processor in cooperation with the OPU. The output processor (OUTPP) is similar to input processor and stores objects on to the network and interfaces with GUIP and OPU.

4.6 Operating System Support

Unlike conventional systems, the OOCA systems do not require a separate operating system to run the computer. The operating system commands are closely integrated with the user interface and also with the object itself. The operating system functions including memory management,. process management, file management, secondary storage management, security, protection, and so on, are based on object-oriented paradigm and they are divided into two basic software management systems; object management and user management. We believe that the operating system layer can be completely avoided by integrating these features right into the user and the object functions. The object kernel functions (operating system related) in addition to the object management functions (instantiation, protection, inheritance, reconfiguration, communication, etc..) can be encapsulated into the outer ring as shown in Figure 4. The object model including the data, member functions, and object mini kernel is being referred to as a new object model (NOM) in this paper. The NOM, if developed properly can avoid the operating system layer, thus resulting in reducing the *semantic-gap* between applications and hardware.

5.0 Implementation

As it can be observed by now, that the implementation of OOCA is not feasible at this point as it requires completely new designs, revolutionary massive VLSI (MVLSI) technologies, and also revolutionary hybrid

computer and memory (HCAM) architectures. The MVLSI technologies should be object oriented and inherit all the properties of object oriented programming. The MVLSI technologies should enable to modularly add chips to the system, preserve the old chips, enhance the existing chips, and mass produce the chips to achieve lower cost. We envision that a computer system can be built by using OOCA building blocks specially tailored for each application. It is possible to achieve this, because each application domain has its own data model and occasionally different data models needs to communicate to each other. For example, a word processing application usually deals with text and graphics data, and occasionally needs to communicate to a database system.

The hybrid computer and memory architectures are new generation of MVLSI chips which allow memory and processing to coexist in a single hybrid chip environment and avoid the need for central processing unit. These HCAM chips should be modularly expandable to realize any size of memory and processing capabilities. As the memories consist of objects, classes, and their functions, modularity is natural to object-oriented hardware. Once we learn how to build HCAM chips, then it is easier to build OOCA systems using hardware objects.

As it is not possible to demonstrate the OOCA system at this point due to the above reasons, we are currently investigating object-oriented simulation techniques to study the feasibility of implementing OOCA.

6.0 Design Issues

There are numerous design issues that need to be addressed for object-oriented computer architecture proposed in this paper. We have identified some of the following design issues for OOCA.

Object Creation: as there is a need for thousands of objects, and the type of objects are not known a priori, we must design memories which can be reconfigured after the design.

Object Addressing: develop an addressing scheme for world of objects which is uniform across a single application domain or across multiple applications.

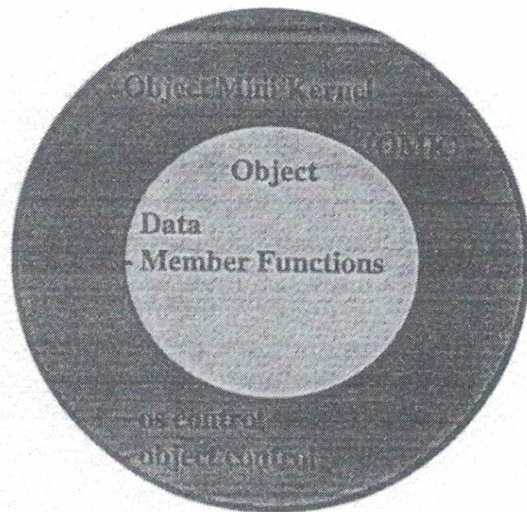

Figure 4: New Object Model

Object Migration: develop an object migration techniques where objects can be migrated across the network and preserve all the data integrity, security, locality of reference, and sharing properties.

Object Storage: Objects can be stored either in local memory, disk memory, or on the network. The object storage mechanisms should allow this transparency to the user and provide realistic performance to object access.

Object Integrity: The object integrity and data security must be preserved irrespective of its location and usage.

Object Processor Unit: The OPU is analogous to central processing unit in conventional computers. Analyze and study the requirements of OPU and find the functionality needed in this unit.

Concurrency: As it can be seen from Figure 3, the OOCA is a fully concurrent system. The OM, CM, and DM can be partitioned based on object domains and one or more OPUs can be allocated to control the object processing in the memory units. The concurrency is achieved by distributing the objects across the network and accessing as required at any given site.

P9

However, concurrency issues such as object distribution, performance, latency, and communication bandwidth will play major role in realizing a distributed system.

Hybrid Computer and Memory: We need a memory technology with processing capabilities to realize HCAM chips. Today's technology, memories are designed separate and processors are designed independently. The semiconductor technologies used to design memories and processors are same, however, the design approaches used are different. For example, memories and processors both use CMOS technology, however, memory designs are based on repetitive memory cell (where one cell is designed and replicated all across the chip), and processor designs are based on a computer architecture (for example, 386, 486, etc..). We need to bring these two design philosophies together and build HCAM chips which can work with objects.

Integration of Memories: We have isolated class memory, object memory, program memory, and disk memory. There is need for interaction of these memories during execution and it is not clear how could we define these interfaces.

MVLSI: The massive VLSI technology needed for OOCA machines demands driving object-oriented technology into hardware. Hardware is treated as 1's and 0's at this point, and this thinking needs to be changed to objects. This may demand revolutionary thinking in VLSI designs and may create new generation of computers that can be plug-and-play by merely changing or inserting hardware objects.

Shift In Paradigm: The OOCA will make a big shift in paradigm. The change in paradigm emphasizes on hardware instead of software. This implies that the hardware manufacturing costs should be still lower than the software development costs and it is absolutely necessary to preserve modularity and longevity in hardware objects.

User Interfaces: As the user interfaces with the system through graphical user interfaces, the GUI should provide all object-oriented interaction of the user to the system. The user should be able to design objects, use existing objects, build relations between objects, develop data models for applications, write implementation code and application code for objects, interpret or compile the created applications, manage objects in the memory and in the database and so on.

Object Primitives: As all the applications are written using object-oriented programming, and also the user interfaces are object-oriented, it is possible to claim that the majority of the functions required in the OOCA computer will be based on object operations. Thus, we believe that there is a need for two types of object primitives, one for the memory units, and the other for the object processor units. We need to identify these primitives and study the implications of implementing these primitives in memory units (OM, CM, PM) and object processor units.

High level interfaces: We believe that there is no need for the operating system interface for OOCA systems. We need to further investigate this and make sure that all the operating system functions can be implemented in the hardware either through the user interface processor or through the object mini kernel. However, we believe that there is a need for object-oriented programming language for the user applications.

Database Interface: It is possible to create objects which will provide all the functions related to perform database transactions. The objects can be stored as collections (lists, bags, sets, and so on), and a query interface can be provided to the user. Object relations can also be stored as attributes to the objects and a complete object model can be simulated in memory. Database addressing can be an extended address space of objects and conventional addressing schemes used in current object-oriented databases can be adopted to this new generation of OOCA. Furher research in database interface to object memory, and object-oriented queries are needed to evaluate the feasibility of the OOCA systems.

7.0 Conclusions and further research

The object-oriented computer architecture proposed in this paper offers many advantages compared to the conventional computer architectures. First, it avoids most of the *semantic-gap* that exists between an application and its implementation. The approach models the application with object-oriented analysis and design techniques such that the real world objects in the application are same as the one in hardware. It offers the possibility of longevity for the hardware and software elements to live their usefulness. Finally, if it becomes feasible to implement, it will not make computer professionals, tools, and resources obsolete as this is normal in today's changing world. We have presented a computer architecture which is based on an object-oriented paradigm, and also set a stage for object-oriented hardware implementation. We have identified numerous design issues and obstacles with respect to this proposed architecture and its implementation. Especially, we identified a need for hybrid computer and memory architectures, designs, and technologies which will enable us to build object-oriented hardware systems.

It is necessary to explore such revolutionary architectures for the future applications as these applications are very complex, and they demand long term economical solutions. The wastage, obsolescence of resources, and people's skills in software and hardware is astronomical and needs an urgent solution. There are numerous research issues in the object-oriented computer architecture presented in this paper, and there is a need for continuing research in this new direction.

Some of the key areas of further research include: studying the feasibility of hybrid computer and memory architectures, designs, and technology; understanding the implications of developing objects with mini kernel code; mapping applications to object-oriented implementation; imbedding operating system functions into user interfaces and objects; formulating object primitives required for OOCA; development of new object-oriented programming language for OOCA, or extending the current object-oriented programming language such as C++; database interface and object-oriented queries, and developing simulation models to understand and identify the design issues related to OOCA.

References

[1] Amamiya Makato, et al., Implementation and evaluation of a List-processing oriented dataflow machine, IEEE 1986.

[2] Atwood, T., The Object DBMS Standard, Object Magazine, October 1993.

[3] Bapat, S., Object-Oriented Networks, Models for Architecture, Operations, and Management, PTR Prentice Hall, Englewood Cliffs, NJ.

[4] Baras, J.S., Murad, A.H, Jang, K., Atallah, G.C., Karne, R.K., and Campenella, S.J., Object-oriented Hybrid Network Simulation, Technology 2004.

[5] Cattell, R.G., What are Next Generation Database Systems, Communications of ACM, October 1991, Vol. 34, No.10, p31-p33.

[6] Desmonde William H, Computers and Their Uses, 1971.

[7] Gene Mathews, et al., Single chip processor runs Lisp environment, Computer Design, May 1987.

[8] Hayashi, H., Hattori, A., and Akimato, H., ALPHA: A High Performance Lisp Machine equipped with new stack structure and garbage collection system, 1983 Conference Proeedings on Computer Architecture.

[9] Karne, R.K., and Sood, A.K., Feasibility Study: Massively parallel Architecture for Time-based Simulation, Transactions of the Society for Computer Simulation, December 1994, Vol. 11, No. 4, p245-p272.

[10] Karne, R.K., and Sood, A.K., PARS: A Parallel Architecture for Rule-based Simulation, Transactions of the Society for Computer Simulation, June 1992, Vol. 9, No. 2, p59-p85.

Silicon Database Machine: Rationale, Design, and Results, Database Machines and Knowledge-base Machines, edited by Masaru Kitsuregowa and Hidehiko Tanaka, Kluwer Academic Publishers, 1988, p311-p324.

[12] Mattison, R., An Object Lesson in MANAGEMENT, Datamation, July 1, 1995, p51-p55.

[13] Myers, G.J., Advanced Computer Architecture, John Wiley & Sons, 1982, p17.

[14] Nakazaki, R., et al., Design of high speed prolog machine (HPM), The 12th Annual International Symposium on Computer Architecture, June 1985, p191.

[15] Pfister, G.F., The IBM Yorktown Simulation Engine, Proceedings of the IEEE, Vol. 74, No. 6, June 1986, p850-p860.

[16] Rine, D.C., and Bhargava, B., Object-oriented Computing, Computer, October 1992.

[17] Semich, J.W., What's The Next Step After Client/Server?, Datamation, March 15, 1994, p26-p34.

[18] SHIBAYAMA, S., KAKUTA, T., MIYAZAKI, N., YOKOTA, H., and MURAKAMI, K., A Relational Database Machine with Large Semiconductor Disk and Hardware Relational Algebra Processor, New Generation of Computing, OHMSHA, 1984, p131-p155.