Object-based Computer Architectures

by

T. N. Mudge, G. D. Buzzard, D. J. Verhaeghe, J. Hill and D. C. Winsor
Department of Electrical and Computer Engineering
University of Michigan
Ann Arbor, MI 48109

Abstract--There has been a sizable increase of interest in object-based computer systems recently. Much of this increase is attributable to DoD's massive commitment to the Ada language project. Though Ada (DoD's proposed standard language) may not fit everybody's definition of an object-based language, it does incorporate key object-based concepts. In this paper we attempt to characterize these systems and the underlying concepts of the object-based design methodology. We also present case studies of two commercially available object-oriented computers and point to issues which require further study.

I. INTRODUCTION

The term "object-based" has been used rather loosely in several different contexts. In this paper we will characterize object-based computer architectures and discuss some of their major implications. While we will not attempt to rigidly define the phrase "object-based" as it relates to computer systems, we will explain several object concepts and show how they are applied to computer systems. It should be noted that it is actually the design methodology that is object-based, and that object-based computer systems refer either to systems which have been designed using this methodology, or to systems which provide development cycle and/or run-time support for object-based concepts. This is, perhaps, most easily illustrated in terms of software systems. There it describes software environments which incorporate the concepts of data and program abstraction [1], and protection domains [2] through the use of "objects." Objects are singly addressable entities that uniquely identify their contents. In terms of the IAPX 432's Ada implementation [3, 4], objects are generally grouped into packages that provide abstractions for either programs or data. Smalltalk objects [5] also provide abstractions, but in a slightly different manner. Implementations of both Ada and Smalltalk objects will be discussed later. In terms of hardware, the phrase "object-based" is often used to refer to the architectural support provided for data and program abstraction, and protection domains. One aspect of this support can be exemplified by an architecture where the primitive operations for memory management, process dispatching, interprocess communication, or other operating system type features are provided by the hardware. In this type of system the implementation details of the memory pool, process dispatching, or interprocess communication mechanisms are hidden; and a concise interface in the form of instructions which operate on the objects corresponding to the respective mechanisms is presented to the software operating system. Another meaning which has been attached to object-based systems refers to systems where the object-based design methodology has been used to actually specify the architecture. When carried to an extreme, this idea has some very powerful implications. When design automation tools become sufficiently sophisticated, one might be able to economically customize individual batches of VLSI processors towards specific operations which appear unusually often in a given (limited) application. This concept is closely related to the hardware/software transparency issue which will be discussed shortly.

This work was supported in part by the Air Force Office of Scientific Research under contract F49620-82-C-0089 and a Kodak Fellowship.

There are two major goals in developing object-based software. The first is to reduce the total life-cycle software cost. This entails both increasing programmer productivity and reducing maintenance costs. For reducing maintenance costs though, object concepts alone may not suffice; the syntax used in the programming environment must be such that it maps very closely to a concise natural language description of the system. The second goal, is to implement software systems which resist both accidental and malicious corruption attempts. Protection domains are used for this purpose.

The major goals for developing object-based hardware are to provide an efficient execution environment for the software system. A very attractive addition to this goal, however, would be to design the system in such a manner as to be able to extend the abstraction mechanism to effect a hardware/software (HW/SW) transparency. The availability of HW/SW transparency would allow system designs to proceed without regard for the final placement of the HW/SW boundary. The entire system could be described by a suitable system implementation language, implemented in software (as far as physically possible), and then after sufficient performance data was acquired appropriate modules could be shifted to a hardware implementation. If the design methodology was implemented in the development environment with sufficient sophistication the migration from software to hardware would not require any code to be rewritten, and the interface would already be specified. The hardware interface could occur at one of three levels: bus level, similar to current arithmetic coprocessors such as the Intel 8087; memory (I/O) level, like many existing I/O devices; or, conceivably, at a level internal to the CPU involving an actual change in the CPU architecture.

There are a few commercially available computer systems which incorporate various object-based concepts. The most notable of these systems, the Intel IAPX 432, and the IBM System/38 [6] will be described in case studies which appear later in the text. The PP250 [7] was designed by Plessey Telecommunications Research Laboratories for applications in telephone switching systems. As such, it had to meet very stringent reliability standards requiring the inclusion of software error detection and recovery facilities, which were achieved through the use of a capability [8] mechanism. Another commercial entry is the Apollo network system [9]. It employs an object oriented network operating system which supports a total address space of 2^{96} bytes. This space is comprised of an object name of 64 bits which is unique in both space and time, and an address space of 32 bits within each object. Several universities have also developed systems which employ object-based concepts. The most recent projects in this group include Cm* [10], C.mmp [11] and CAP [12]. Like most of their predecessors, these machines utilize capability addressing techniques to implement secure protection domains. These protection domains (packages in Ada) can then be appropriately structured to provide data and program abstractions.

Object-based machines are particularly well suited to applications which have stringent requirements for data security and program integrity. The high degree of abstraction provided by the architecture also facilitates the interconnection of several processors into either tightly coupled multiprocessor systems and/or distributed networks. Through the use of process/processor abstraction Intel has achieved software transparent multiprocessing in their iAPX 432 system. In addition, the Cm* system provides an example employing both tightly coupled multiprocessing and distributed networking concepts in one system.

Of course, all of the benefits of an object-based system do not come without their price. All present systems rely on some form of capability addressing. In current implementations these addressing mechanisms greatly increase the address generation and translation times, even when translation look-aside and caching schemes are employed. For example, to copy a capability on Cm* requires ten memory references [13], and on the iAPX 432 nine memory references are required.

The following section discusses some key "object" concepts and their implementations in more detail. Section III contains case studies of two commercially available systems, the Intel iAPX 432 and the IBM System/38. Concluding remarks, including comments on areas of further research, are made in section IV.

II. Object-Based Concepts

Abstraction plays a central role in the object-based design methodology and, hence, occurs at many levels. The most common of these levels being data, programs, and in the case of Apollo-type network systems, entire programming environments. Of these, data abstraction is the most widely used and best understood.

In terms of Ada, data abstraction provides the user with a "private" (hidden) type and operations on objects of that type [14]. Mary Shaw gives a more expansive definition of an abstract data type [1]. An abstract data type consists of a program unit that includes the following information:

> *Visible outside the type definition:* the name of the type and the names and routine headers of all operations (procedures and functions) that are permitted to use the representation of the type; some languages (e.g. Ada) also include formal specifications of the values that variables of this type may assume and of the properties of the operations.
>
> *Not visible outside the type definition:* the representation of the type in terms of built-in data types or other defined types, the bodies of the visible routines, and hidden routines that may be called only from within the module.

The efficient use of abstract data types requires substantial support from the programming language. Some of the more important areas of support include: naming and scope rules, type checking (including across compilation boundaries if separate compilations are supported), formal specification notations, and distributed properties. Enforcement of naming and scope rules is often implemented through the use of protection domains, which will be discussed later. Type checking between actual and formal parameters is more complex for abstract data types because new types may be defined during the compilation process. Notations for formal specifications of the abstract data types must be provided so that the users of the types are provided with all necessary information. Finally, support must be provided by the abstract data types for any type-specific interpretations required by the various constructs of the programming language, such as storage allocation and synchronization.

The construct for implementing abstract data types in Ada is the package. The Ada package [15] effectively places a wall around a group of declarations and only permits access to those declarations which were intended to be visible. Ada packages actually come in two parts, the specification and the body. The package specification formally specifies the abstract data type and its interface to the outside world. The body of the package contains the hidden implementation details. The relationship between Ada packages and objects (in the context of the iAPX 432) will be discussed in the iAPX 432 case study.

Data abstraction in Smalltalk also fits the definition given above. However, the implementation is very different. Data abstraction is very integrally related to the concept of Smalltalk objects [5, 18], so a brief introduction to objects will precede the references to data abstraction.

In Smalltalk all information is represented in the form of objects. An object is defined as a package containing information and descriptions of all the manipulations that may be performed on the information. The manipulation of information is controlled by the passing of messages between objects. A Smalltalk message is defined as a selection of one particular manipulation of an object. The object containing the information to be manipulated is called the receiver of the message. When an object receives a message, it examines a symbolic name contained in the message called a selector. The selector names the desired manipulation, but it does not specify how it is to be performed. The description of how the manipulation is to be performed is contained in the object itself. In Smalltalk, the programmer sends a message to invoke a manipulation instead of calling a procedure. However, the sending of a Smalltalk message differs from a procedure call in that the message only names the manipulation while a procedure contains the detailed steps to be performed in the manipulation. Another important difference is that in a conventional procedure-oriented system the same manipulation is performed each time a named procedure is called, while sending Smalltalk messages with the same selectors (same names) to different objects may result in different manipulations, since it is the receiver which determines exactly what manipulation is to be performed.

In addition to a selector, a message may contain the names of other objects that take part in the manipulation. These names are called the message arguments. For example, if the object is named "student," the operating system which supports a total address space of 2^{96} bytes. This space is comprised of an object name of 64 bits which is unique in both space and time, and an address space of 32 bits within each object. Several universities have also developed systems which employ object-based concepts. The most recent projects in this group include Cm* [10], C.mmp [11] and CAP [12]. Like most of their predecessors, these machines utilize capability addressing techniques to implement secure protection domains. These protection domains (packages in Ada) can then be appropriately structured to provide data and program abstractions.

Object-based machines are particularly well suited to applications which have stringent requirements for data security and program integrity. The high degree of abstraction provided by the architecture also facilitates the interconnection of several processors into either tightly coupled multiprocessor systems and/or distributed networks. Through the use of process/processor abstraction Intel has achieved software transparent multiprocessing in their iAPX 432 system. In addition, the Cm* system provides an example employing both tightly coupled multiprocessing and distributed networking concepts in one system.

Of course, all of the benefits of an object-based system do not come without their price. All present systems rely on some form of capability addressing. In current implementations these addressing mechanisms greatly increase the address generation and translation times, even when translation look-aside and caching schemes are employed. For example, to copy a capability on Cm* requires ten memory references [13], and on the iAPX 432 nine memory references are required.

The following section discusses some key "object" concepts and their implementations in more detail. Section III contains case studies of two commercially available systems, the Intel iAPX 432 and the IBM System/38. Concluding remarks, including comments on areas of further research, are made in section IV.

II. OBJECT-BASED CONCEPTS

Abstraction plays a central role in the object-based design methodology and, hence, occurs at many levels. The most common of these levels being data, programs, and in the case of Apollo-type network systems, entire programming environments. Of these, data abstraction is the most widely used and best understood.

In terms of Ada, data abstraction provides the user with a "private" (hidden) type and operations on objects of that type [14]. Mary Shaw gives a more expansive definition of an abstract data type [1]. An abstract data type consists of a program unit that includes the following information:

Visible outside the type definition: the name of the type and the names and routine headers of all operations (procedures and functions) that are permitted to use the representation of the type; some languages (e.g. Ada) also include formal specifications of the values that variables of this type may assume and of the properties of the operations.

Not visible outside the type definition: the representation of the type in terms of built-in data types or other defined types, the bodies of the visible routines, and hidden routines that may be called only from within the module.

The efficient use of abstract data types requires substantial support from the programming language. Some of the more important areas of support include: naming and scope rules, type checking (including across compilation boundaries if separate compilations are supported), formal specification notations, and distributed properties. Enforcement of naming and scope rules is often implemented through the use of protection domains, which will be discussed later. Type checking between actual and formal parameters is more complex for abstract data types because new types may be defined during the compilation process. Notations for formal specifications of the abstract data types must be provided so that the users of the types are provided with all necessary information. Finally, support must be provided by the abstract data types for any type-specific interpretations required by the various constructs of the programming language, such as storage allocation and synchronization.

The construct for implementing abstract data types in Ada is the package. The Ada package [15] effectively places a wall around a group of declarations and only permits access to those declarations which were intended to be visible. Ada packages actually come in two parts, the specification and the body. The package specification formally specifies the abstract data type and its interface to the outside world. The body of the package contains the hidden implementation details. The relationship between Ada packages and objects (in the context of the iAPX 432) will be discussed in the iAPX 432 case study.

Data abstraction in Smalltalk also fits the definition given above. However, the implementation is very different. Data abstraction is very integrally related to the concept of Smalltalk objects [5, 18], so a brief introduction to objects will precede the references to data abstraction.

In Smalltalk all information is represented in the form of objects. An object is defined as a package containing information and descriptions of all the manipulations that may be performed on the information. The manipulation of information is controlled by the passing of messages between objects. A Smalltalk message is defined as a selection of one particular manipulation of an object. The object containing the information to be manipulated is called the receiver of the message. When an object receives a message, it examines a symbolic name contained in the message called a selector. The selector names the desired manipulation, but it does not specify how it is to be performed. The description of how the manipulation is to be performed is contained in the object itself. In Smalltalk, the programmer sends a message to invoke a manipulation instead of calling a procedure. However, the sending of a Smalltalk message differs from a procedure call in that the message only names the manipulation while a procedure contains the detailed steps to be performed in the manipulation. Another important difference is that in a conventional procedure-oriented system the same manipulation is performed each time a named procedure is called, while sending Smalltalk messages with the same selectors (same names) to different objects may result in different manipulations, since it is the receiver which determines exactly what manipulation is to be performed.

In addition to a selector, a message may contain the names of other objects that take part in the manipulation. These names are called the message arguments. For example, if the object is named "student," the programmer might send a message with the selector "abuse." The message could also contain an argument which specifies how the student is to be abused.

The description of a single type of manipulation of the information in an object is called a method. A method is similar to a procedure in that it describes a sequence of actions to be performed, but unlike a procedure it cannot be separated from the object. Methods may only be performed as the result of receiving a message. Thus a method cannot call another method directly; it must send a message to the object containing the method.

Smalltalk objects use the concept of data abstraction. From "outside" an object the only thing that can be done is to send it a message. The details of its information and its methods are hidden. An object's set of messages that it can receive are called its protocol. The external view of an object is strictly limited to its protocol. Internally, an object consists of its methods and a set of variables that refer to other objects. These variables are called the object's private variables. The methods and the private variables are analogous to the procedures and the data in a procedure-oriented system. However, the distinction between procedures and data is localized strictly to the inside of the object.

Programs and subprograms provide another common level of abstraction. Program abstraction provides operations on structures of objects where the representation of the structure as well as access to the object itself are hidden from the user. This provides a stronger form of hiding than data abstraction since access to the object as well as its structured representation is hidden.

Program abstraction in Ada is realized through generic package instantiation. The generic package is really a template for packages which will accept abstractions (usually data, but in the case of Intel's extended Ada it could be another package) as actual parameters. This represents a slightly higher level of abstraction than data abstraction, because the structure of the object(s) being manipulated is completely hidden within the package body. The hidden object structure is accessed through the internal non-local variables of the operations declared in the package specification. Manipulation of the object structure occurs as a controlled side effect, which is strictly contained within the package body, of the requested operation. In this manner, generic program abstraction supports an environment in which the specified (public) operations either directly or indirectly transform a hidden internal state which depends only on past operations applied to the initial state of the system.

While it is apparent that Smalltalk objects have already met the definition of program abstraction, we have not shown how the relationship between objects of similar characteristics can be exploited. This is essential if the concept of program abstraction is to have any significant practical value. When program abstraction is efficiently implemented it leads to the elimination of redundant programming effort (e.g. a sort routine need be written only once, regardless of the different types of objects to be sorted) and a reduction of maintenance costs (e.g. improvements to our sort routine are made in only one piece of code). Operations on objects of different types, but with similar characteristics, are realized through the use of classes and instances. A class is a description of a type of object, while an instance is an object of a particular class. Every object is an instance of a class, and it is possible to have multiple instances of a particular class. The methods of an object are found in its class, so all instances of a particular class have the same protocol. The class may also specify some of the private variables for objects in the class. These are called class variables and they are shared by all instances of the class. A class of objects may also include private variables which are specified individually for each instance. These are called instance variables. All instances of a class have the same number of instance variables, but their values may differ between instances. A Smalltalk program is, then, organized as a set of class templates which define all user defined object classes. The class templates include the class names, instance variable names, and methods. A method is described by its message pattern, which specifies the selector used to invoke it, its temporary variable names, which specify dynamic local variables, and a sequence of expressions.

We have seen that the concepts of data and program abstraction in Smalltalk are essentially one in the same. It has been proposed that these concepts be merged together in Ada as well [14]. In fact, Intel has already taken a big step in this direction with their extensions to the Ada language [16]. The merging of program and data abstraction concepts would result in a unified abstraction mechanism that would eliminate two relatively orthogonal concepts. The software designer would then be relieved of the artificial choice between program-oriented or data-oriented programming methodologies.

A higher, though perhaps less sophisticated, level of abstraction is provided in the Apollo computer system [9]. Apollo provides an object-oriented network operating system to coordinate the user's access to network wide facilities. Objects are used to represent programs, data files, or even entire programming environments. The network global object spaces are selectively mapped into a process virtual address space on a given node of the system. The objects are network wide; whereas, the processes are all on a particular node running on behalf of a particular user. The address space mapping represents the only primitive in which processes can relate to objects. For the most part the operating system and all higher level views of the system relate to objects rather than processes.

Protection domains, and the inherent security that they provide, are another key object concept. The basis for secure and error-tolerant execution environments lies in the principle of system closure [2]. This principle basically states that the effects of all operations on a closed system shall remain strictly within that system. One common construct used for providing system closure is the protection domain [17]. Briefly stated, a protection domain is an environment or context that defines the set of access rights that are currently available to a specific user for objects of the system. Capability based addressing schemes are the most efficient known mechanism for implementing protection domains.

Protection domain schemes generally provide facilities for error confinement, error detection and categorization, reconfiguration, and restarting. Error confinement (and security) strategies generally involve both process isolation and resource control. The basic premise of process isolation is that processes are given only the capabilities necessary to complete their required tasks. This implies that interactions with any external objects (e.g. sending messages to other processes) must be strictly formalized and controlled. Resource control refers to the binding of physical resource units to computational objects. Examples of this include the binding of processes to processors, or the assignment of memory to currently executing contexts. The idea here is to ensure that when the resource units are released, or preempted, that all information contained within the unit is returned to a null state. This prevents any information from "leaking" out of a protection domain by being innocently left in an area that will eventually become accessible to other users. Error confinement also aids the program debugging process, since bugs should be located in the same module which contains the resulting error. Program maintenance also benefits since the protection domain defines the maximal set of modules which could be affected by a modification to the system. Error detection and categorization involves dynamic checking for object type inconsistencies and access constraint violations in executing procedures. The categorization of detected errors can then be used to aid in restoring the system to a known consistent state. Reconfiguration facilities attempt to restore the system to an operable state by removing from service the failed component, be it hardware or software. If the reconfiguration attempt is successful, the system is then restarted.

The most efficient known mechanism for implementing protection domains is the capability mechanism. While much can be done at compile time to enforce the concepts of protection domains, there are many cases where a dynamic enforcement mechanism is essential. The real-time sharing of data between programs provides an obvious example. But compile-time protection enforcement also lacks the ability to support the detection of, and recovery from, failures in the run-time system. A brief description of the capability mechanism follows in the next paragraph.

A capability can be thought of as a name of an object. An object cannot be accessed (and, in fact, its existence cannot even be determined) unless its name is possessed. The capability also contains the access rights to the object (e.g. read, write, or capability copy rights). The only subsequent modification allowed is the restriction of these rights. Capabilities are created along with their respective objects. The initial control of the capability, hence the object, belongs solely to the creating context. Specific implementation details are given in the case studies.

III. CASE STUDIES

A. Intel iAPX 432

The Intel iAPX 432 is an object-based microcomputer system, developed in an effort to curb the rising costs of software. Measured as a percentage of total system costs, the cost of software has been rising dramatically for the past several years. Intel attributes much of this increase to four common requirements that are characteristic of modern software systems [19]:

(1) Demand for systems embodying many large programs with complex interactions.

(2) Demand for increased system security.

(3) Demand for efficient concurrent programming support.

(4) Demand for truly transparent multiprocessing, where changing the number of processors in the system does not require any software modifications.

The object-based design methodologies embodied in, and supported by, the iAPX 432 provide solutions for these problems. Furthermore, it is hoped that the underlying architectural support for object-based programming methodologies will help to provide the environment necessary to aid in controlling the rise of software costs.

All information in the iAPX 432 system is represented by typed objects. An object is defined by the following four characteristics [20]. First, an object is a data structure containing organized information. Objects also define the set of operations which may be performed on themselves. In fact, these are the *only* operations that are allowed. The third characteristic of iAPX 432 objects is that they are referenced as a single entity, regardless of the length of the object. Finally, every object has a unique label that contains the information about its type.

Objects are implemented as a collection of one or more segments. Segments are of variable length, and come in one of three hardware recognized forms, access only, data only, or a combination of both. Segment types are identified by header information which is stored in the segment itself, but is virtually (in the address sense) invisible to the software. In the case of combination segments, the location of the boundary between the access and data parts is also stored with the type. Access segments (or parts of segments) can contain only access descriptors or null entries. Access descriptors are the "capabilities" in the iAPX 432 system. Data segments contain all of the other information in the system, including things such as instructions or process status information.

The ability of the hardware to identify access segments is one of the key mechanisms used by the iAPX 432 to enforce protection domain security. Any attempts to modify access segments can be closely monitored. The security mechanisms inherent in the iAPX 432 architecture are not limited to the above, however. As an example, we will consider the protection against executing data. The currently executing process references instructions via two indices, one of which is an instruction pointer that provides an offset into the current instruction object. The other is an index which selects a capability that determines the current instruction object from the domain object (described later). Since the physical base address of the currently executing instruction object is cached on chip and since length bounds checks are automatically performed by the hardware on all memory references, instruction fetching type consistency is guaranteed by checking the object type of the instruction object referenced by the indexed capability. This needs to be done only when the current instruction object index is modified, that is, only when an intersegment branch is executed.

There are a number of hardware (microcode, actually) recognized objects in the iAPX 432 which are primarily used to support the object-based concepts discussed in section II. This includes objects which represent instructions, protection domains, activation records, processes, and even physical processors. Two of the more interesting ones are the context and domain objects; these represent activation records and protection domains, respectively. The domain objects and context objects are used to realize the concept of type managers, the system's primary mechanism for implementing data and program abstraction concepts. Type managers are modules that provide information hiding (or data encapsulation) by containing a data structure and all the necessary procedures to manipulate that data structure. By allowing only a strictly controlled set of procedures to be invoked from outside of the protection domain (i.e. the domain object), the implementation of the data structure and the procedures which directly manipulate it are effectively "hidden" from the outside world. The domain object represents a type manager's static structure via an object complex (a tree-like structure objects, in this case, with the domain object as its root). This object complex incorporates instruction objects and data objects, referenced via the domain object access segment, which contains all object references in the domain, both public and private. The private references are completely inaccessible to objects outside the domain, and thus, realize data abstraction. Since the domain object contains the data object and all the instruction objects that operate on it, we can see that standard Ada packages map directly onto domain objects. Additionally, since all generic Ada packages are instantiated at compile time, each instantiation is represented by its own unique domain object.

The context object contains the dynamic run-time information which describes the execution environment of an invoked procedure; thus every activated procedure in the system has a context object associated with it. When a procedure is called, a context object is automatically created; conversely, when a procedure returns, the context object is automatically destroyed. Intel provides an extension of Ada in the form of package types. Through the use of the hardware-supported domain and context objects, package types can be passed as parameters to other packages [16]. This allows dynamically defined entities, such as dynamic generic package instantiations, to be easily handled in the iAPX 432. For example, a user may wish to write a procedure that manipulates objects of an unknown arbitrary structure by performing very general operations on them, e.g., a garbage collection algorithm. Standard Ada requires all types to remain static, and to be known at compile time. The iAPX 432, through Intel's extended Ada, supports dynamic applications using arbitrary types directly. Another possibility is for an operating system to define its I/O devices as packages types. Then, as devices are added or removed, the system could dynamically reconfigure itself without operator intervention.

To enforce system security, Intel employed capability addressing in the iAPX 432. All of the object references for a given protection domain exist in the domain object's access segment, or indirectly, in access segment objects which are referenced from the domain. In order to implement the domain object as a single object and still allow for public and private regions, Intel uses an object refinement mechanism. This allows a contiguous section of an object to be treated as a whole object in terms of capability rights. Thus when a domain object is referenced by an "outside" procedure, the calling procedure will be using a capability for the public refinement area of the domain access segment. By prohibiting capabilities for the private area to be given out, that area will remain inaccessible from outside the domain. In the public area, capabilities reside for the instruction objects that are public information.

One of the design decisions that is interesting to compare among object-based systems is the implementation of the addressing scheme. Addressing in the iAPX 432 is accomplished through a series of indexed table look-ups, similar to segment tables in a segmented memory architecture. Address translation is a two level mapping. A system-wide table known as the object table directory exists at a known, physical address. This table represents the first level of the mapping process and contains the base addresses of all the object tables in the system (maximum of 4096). Object tables represent the second level of the mapping process and contain object descriptors for all the objects associated with that object table. Roughly speaking, there is one-to-one correspondence between processes and object tables. The object descriptors found within the object tables contain a 24-bit physical base address, length, type, and other information for their respective objects.

A data reference in the iAPX 432 instruction stream consists of a 16-bit access selector and a 16-bit displacement [21, 22] (see figure 1). The access selector is run through a content-addressable memory (CAM) and if a match occurs, the base address of the object is obtained. The 16-bit displacement is then applied and the data element retrieved. If there is no match, then a rather lengthy address translation takes place. The access selector is broken up into a 14-bit displacement and a 2-bit select field. The two bits select one of four entered access segments (EAS's), which are in on-chip registers. These EAS's hold access descriptors (AD's, capabilities) for access segments. The 14-bit displacement provides an index into the selected segment to obtain the AD for the requested data object. Once the physical base address for the data object is translated, the 16-bit offset from the instruction stream is added to it to select the actual byte(s) referenced. The above description implicitly included the translation of two AD's, one for the access segment, the other for the actual data object. Each one of these translations consists of: using the first 12-bit field in the AD to index into the object table directory to select a object descriptor for an object table; then using the other 12-bit field in the AD to index into the specified object table and select a object descriptor for the desired object, either the access segment or the data object. The above addressing scheme provides a total virtual address space of 2^{40} bytes, this comprises the 2^{12} object tables which can each contain 2^{12} object descriptors for objects that are up to 2^{16} bytes in length. However, at any one instant of time a process's logical address space is limited to 2^{32} bytes. This is because there are only four (2^2) EAS registers which hold AD's for access segments that each contain 2^{14} AD's for the actually addressable objects which are up to 2^{16} bytes in length.

One can see from the diagram that six off-chip memory references are needed to retrieve a piece of data if its address is not available in the cache. Once the AD for the access segment is retrieved from the appropriate on-chip EAS register, three memory references are needed to obtain the AD for the data object. Two more references are then required to get the base address of the data object and finally, one more gets the actual piece of data. Once an object is referenced, its base address and length are stored in the cache. Once the object is cached, the number of memory references needed for data retrieval is reduced to two. While security is strongly desired in today's complex software systems, it is seen that the addressing of information in such a secure system can lead to complex and lengthy address translation mechanisms.

B. IBM System/38

The IBM System/38 was designed for general purpose data processing, supporting both batch and time-sharing environments. System/38 is aimed toward business applications rather than numerically intensive scientific applications. Consequently, the system needs to provide functions such as task management, inter-task communication, and high-level data-base manipulations. The System/38 provides this support at the hardware/architecture level. This high-level machine interface is implemented through the use of objects. High-level machine instructions are provided that manipulate objects rather than simple data types such as byte strings. This allows the execution of complicated functions such as inter-task communication to be performed with just one instruction.

There are two major types of objects in the System/38, system and program objects. System objects are used to perform machine functions such as message-queuing, process initiations, and I/O operations. Program objects are used for operations such as operand accessing, branching, and exception handling. System and program objects, respectively, will be considered in more detail in the following two paragraphs.

All system objects support data abstraction to some extent. This is done by implementing every system object with a functional portion and an associated data space--an exception to this is a space object which contains an associated space only. The functional portion of the object contains information concerning the operations allowed on the object and the microcode needed to perform those operations. This is analogous to the type manager concept supported in the Intel iAPX 432.

Pointer data objects contain pointers which are used to access both system and program objects. A specific type of pointer, the system pointer, is used to access a system object. It can be in one of two modes: resolved or unresolved. In the unresolved state the system pointer contains the name of an object. When an unresolved system pointer is referenced, a search is made in a context object to associate a location with the object name. This location is then placed in the pointer, changing its state to resolved. Each process has a name resolution list, which specifies which context object(s) to search when resolving system pointers. This is useful for easily changing the environment for different instantiations of the same process. Additionally, data pointers, which are used to access data in objects, can also be unresolved [23]. Using unresolved data pointers allows programs to operate on data whose attributes are dynamic. This implies that executing code can operate on dynamically changing data types, as on the IAPX 432. Thus, the use of unresolved pointers in the System/38 allows for a form of program abstraction.

The System/38 uses a form of capability addressing to enforce system security requirements. Pointers and user profiles (UP) together form the capability for an object. All processes execute under control of a specific UP. This user profile contains the authorization list for all permanent objects owned by the given user. Pointers basically provide the addressability for an object, while the UP decides if a module has the authority to perform the desired operation. To increase efficiency, the rights information contained in the UP is allowed to be passed into the system pointer. If this is done, the UP can no longer retract authorization for specific modules. Thus, the ability to place an object authorization in an system pointer is itself an authorization which may or may not be granted by the UP.

The IBM System/38 employs a segmented page type memory architecture. Pages are 512 bytes long and segments are from 128 to 32K pages. The virtual-to-physical address mapping is a two-level mapping scheme, similar to the past memory architectures from IBM. The hash table is the first level of mapping and provides an index into the page directory (see figure 2). The page directory is the second level of mapping and yields the most significant bits of the physical address.

An operand reference in the System/38 is a lengthy process, as in the IAPX 432. In the instruction stream, an operand field specifies an entry into both the object definition table and the object mapping table. The object directory table entry yields a 4-byte object descriptor and, optionally, an extension of this descriptor. The object mapping table provides a 6-byte virtual address for the required system pointer. The virtual address is then run through the virtual address translation mechanism. The 39 most significant bits are run through a hash generator which outputs an index into the page directory. The page directory contains a linked list of virtual page addresses for all entries having identical hash codes. The page directory gives a page address which, when concatenated with the nine least significant bits of the virtual address, yields a physical address. This address provides the system pointer which contains the virtual address for the data object. After passing this address through the virtual-to-physical address translation mechanism, the physical address of the operand is obtained. As in the IAPX 432, the use of a capability addressing scheme to ensure tight system security leads to lengthy address translations.

IV. CONCLUSION

In this paper we have characterized object-based computer architectures. This was done by illustrating key concepts with examples drawn from hardware and software systems. The concept of an object was illustrated rather than defined to avoid associating yet another definition with this term. A case was made for object-based systems reducing system development costs and providing a secure execution environment. These benefits require the use of an elaborate addressing mechanism which significantly increases address generation time. Quantifying this trade-off is the first step in evaluating object-based computer architectures. Development of the techniques necessary for this quantification, and the evaluation of any changes in the architecture that may result, present important research issues.

V. REFERENCES

[1] Shaw, Mary, "The Impact of Abstraction Concerns on Modular Programming Languages," *Proceedings of the IEEE*, Vol. 68, No. 9, September 1980, pp. 1119-30.

[2] Denning, Peter J., "Fault Tolerant Operating Systems," *Computing Surveys*, Vol. 8, No. 4, December 1976, pp. 359-89.

[3] *IAPX 432 General Data Processor Architecture Reference Manual, Revision 2*, 171860-002, Intel Corporation, Santa Clara, California, 95051, 1982.

[4] *Reference Manual for the Ada Programming Language*, 171869-002, Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051, 1981.

[5] Robson, David, "Object-Oriented Software Systems," *Byte*, August 1981, pp. 74-86.

[6] *IBM System/38 Functional Concepts Manual*, GA21-9330-1, IBM Corporation, 1982.

[7] England, D. M., "Capability Concept Mechanism and Structure in System 250," *International workshop on protection in operating systems*, IRIA, Rocquencourt, August 1974, pp. 63-82.

[8] Fabry, R., "Capability-based Addressing," *CACM*, Vol. 17, No. 7, July 1974, pp. 403-12.

[9] *Apollo Domain Architecture Manual*, Apollo Computer Inc., 19 Alpha Road, Chelmsford, Massachusetts 01824.

[10] Swan, R. J., Fuller, S. H., and Siewiorek, D. P., "Cm*: A Modular Multi-Microprocessor," *AFIPS Conference Proceedings*, Vol. 46, 1977 National Computer Conference, pp. 637-43.

[11] Wulf, W. A. and Bell, C. G., "C.mmp - A Multi-Mini-Processor," *AFIPS Conference Proceedings*, Vol. 41, part II, FJCC 1972, pp. 765-77.

[12] Needham, R. H. and Walker, R. D. H., "The CAP computer and its protection system," *ACM 6th Symposium on Operating System Principles*, 1977.

[13] Jones, Anita K. and Gehringer, Edward F. (eds.), "The CM* Multiprocessor Project: a research review," Department of Computer Science, *Carnegie-Mellon University Report CMU-CS-80-131*, July 1980.

[14] Wegner, Peter, "On the Unification of Data and Program Abstraction in Ada," *ACM Conference Record of the 10th Annual ACM Symposium on Principles of Programming Languages*, January 1983, pp. 256-64.

[15] Barnes, J. G. P., *Programming in Ada*, Addison-Wesley Publishers, London,1982.

[16] *Reference Manual for the Intel 432 Extensions to Ada*, 172283-001, Intel Corporation, 3065 Bowers Avenue, Santa Clara, California, 95051, 1981.

[17] Linden, Theodore A., "Operating System Structures to Support Security and Reliable Software," *Computing Surveys*, Vol. 8, No. 4, December 1976, pp. 409-45.

[18] The Xerox Learning Research Group, "The Smalltalk 80 System," *Byte*, August 1981, pp. 36-48.

[19] *Introduction to the IAPX 432 Architecture*, 171821-001, Intel Corporation, 3065 Bowers Avenue, Santa Clara, California, 95051, 1981.

[20] *IAPX 432 Object Primer*, 171858-001, Rev. B, Intel Corporation, 3065 Bowers Avenue, Santa Clara, California, 95051, 1980.

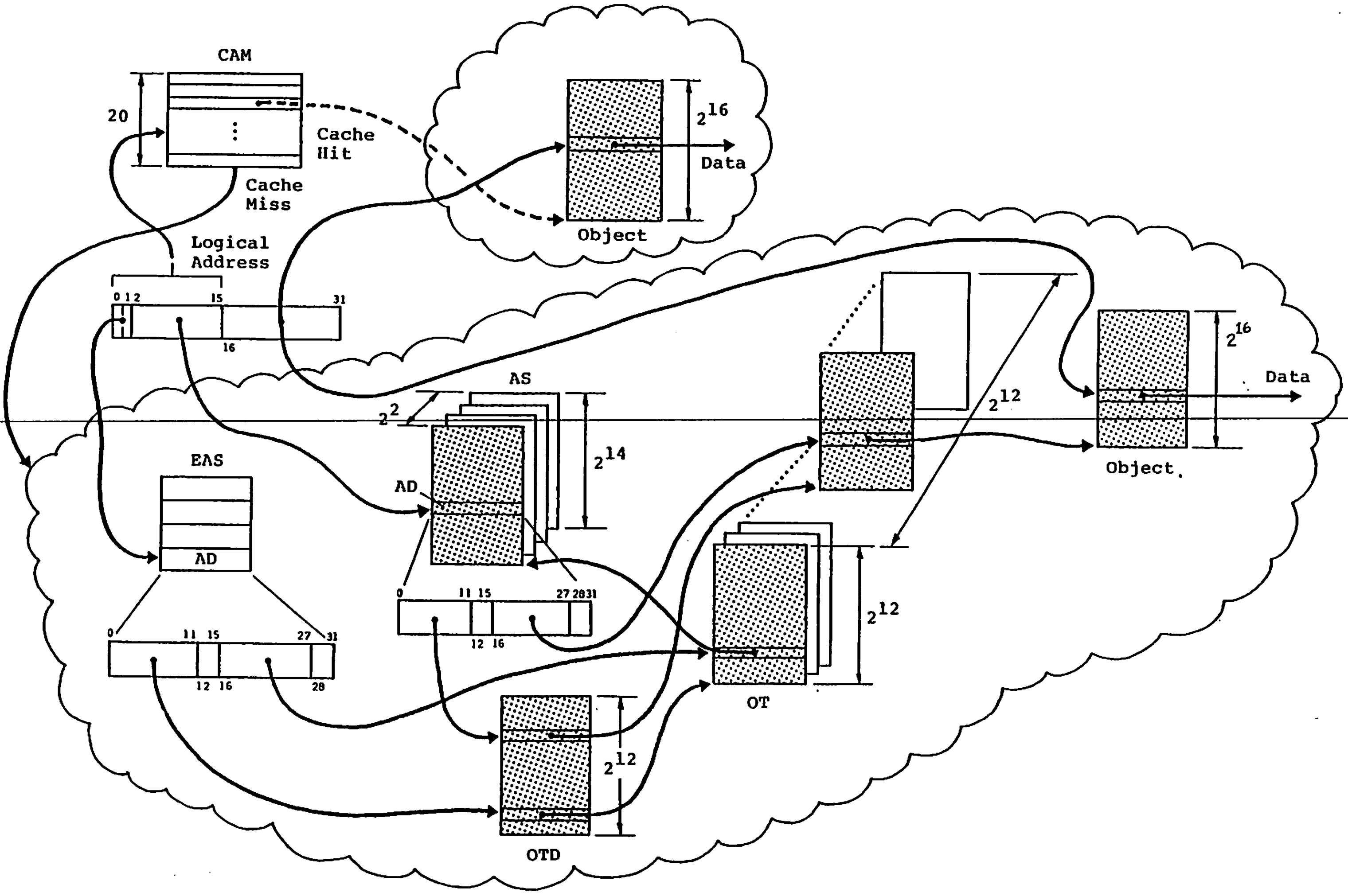
[21] Hemenway, Jack and Grappel, Robert, "Understand the Newest Processor to Avoid Future Shock," *Electronic Design News*, April 29, 1981, pp. 129-36.

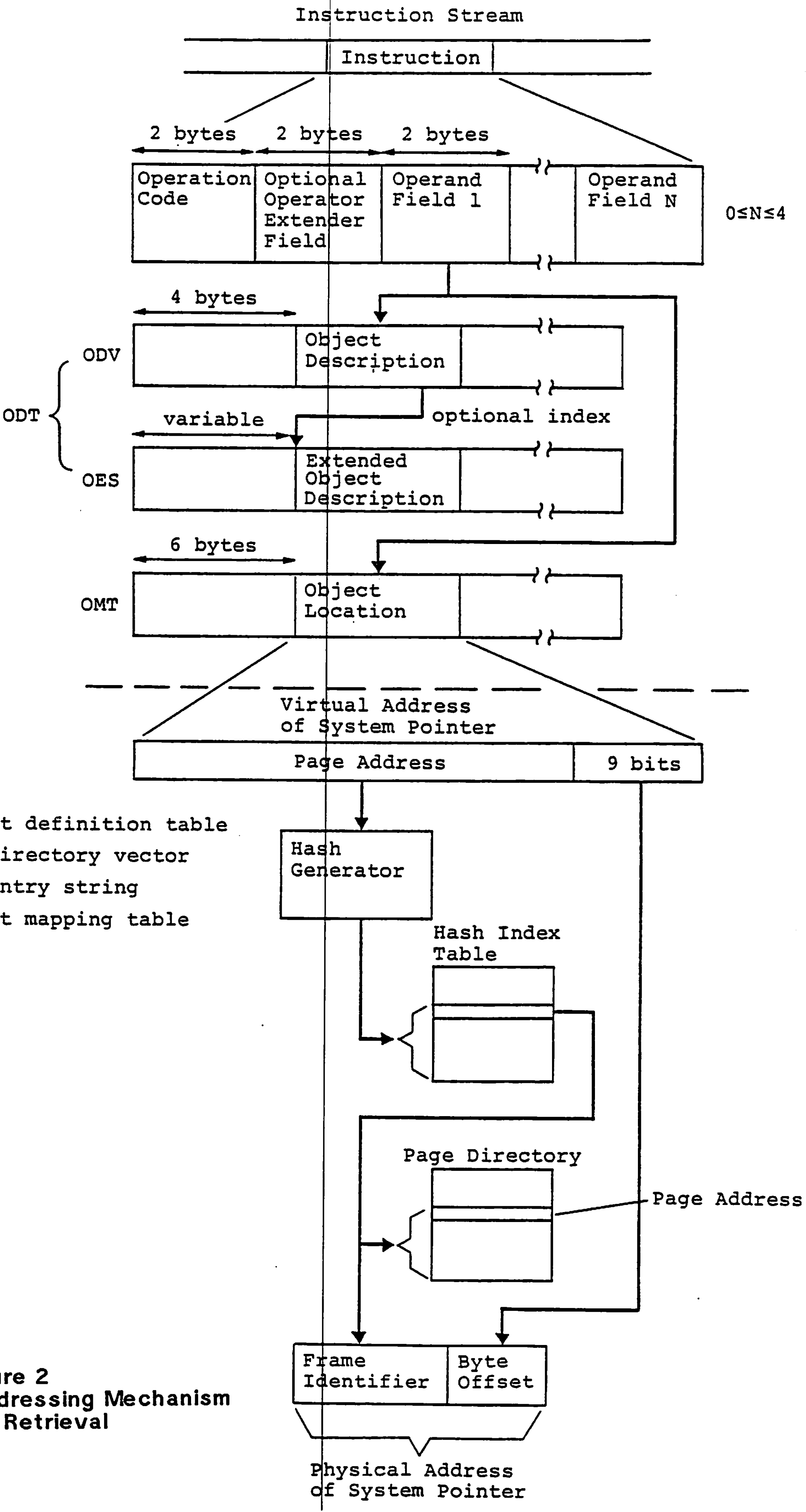
[22] Budde, David L., Colley, Steven R., Domenik, Stephen L., Goodman, Allan L., Howard, James D., and Imel, Michael T., "The Execution Unit for the VLSI 432 General Data Processor," *IEEE Journal of Solid-State Circuits*, Vol. SC-16, No. 5, October 1981, pp. 514-21.

[23] Iliffe, J. K., *Advanced Computer Design*, Prentice-Hall International, London, 1982, pp. 363-78.

[24] *IBM System/38 Technical Developments*, G580-0237-1, IBM Corporation, 1980.

Figure 1
Intel iAPX 432 Addressing Mechanism
Operand Retrieval





ODT = Object definition table
ODV = ODT directory vector
OES = ODT entry string
OMT = Object mapping table

Figure 2
IBM System/38 Addressing Mechanism
Operand Retrieval