iMAX: A Multiprocessor Operating System for an Object-Based Computer

Kevin C. Kahn, William M. Corwin, T. Don Dennis, Herman D'Hooge, David E. Hubka, Linda A. Hutchins, John T. Montague, and Fred J. Pollack

Intel Corporation, Aloha, Oregon

Michael R. Gifkins

Standard Telecommunications Laboratory, Harlow, U.K.

ABSTRACT

The Intel iAPX 432 is an object-based microcomputer which, together with its operating system iMAX, provides a multiprocessor computer system designed around the ideas of data abstraction. iMAX is implemented in Ada and provides, through its interface and facilities, an Ada view of the 432 system. Of paramount concern in this system is the uniformity of approach among the architecture, the operating system, and the language. Some interesting aspects of both the external and internal views of iMAX are discussed to illustrate this uniform approach.

1. Introduction

The Intel iAPX 432 is an object-based microcomputer system with a unified approach to the design of its architecture, operating system, and systems programming language. Its underlying addressing structure is capability-based. [1] It incorporates support for data abstraction, typing, and program structuring, using Ada as its system programming language. The 432 also uses its object orientation as a basis for moving a number of critical software operations into the hardware.

A major goal of the 432 and therefore of iMAX has been to uniformly structure the hardware and software of the system around a single set of concepts based on objects. In addition to providing a common framework for the design of the system, this approach leads to an economy of concepts that eases learning and using the system. Once the notion of object-oriented design is understood, it can be applied equally well to all aspects of the system from the underlying VLSI hardware to the programmer's language interface. This property contrasts sharply with conventional systems that use quite different structures in their base hardware, central operating system, file stores, languages, etc. The major goal of this paper is to demonstrate the uniformity achieved in iMAX via the integration of concepts in the underlying architecture, Ada, and the operating system itself.

This paper presents an overall view of the 432 as seen through iMAX, its operating system. [2] As an object-oriented operating system, iMAX has its roots in the academic research embodied in systems like Hydra, [3,4] CAP, [5,6] StarOS, [7] and the CAL timesharing system. [8] After a brief overview of the 432 hardware, some of the most important external attributes of iMAX are described, then interesting aspects of the internal structure of iMAX and various issues that arise in designing an object-oriented computer system are discussed.

2. The 432 Architecture

Because the 432 blurs the distinction between hardware and software, it is worth distinguishing some important aspects of the conceptual architecture that are actually implemented in the two chip VLSI processor. The 432 addressing structure is capability-based. **Access descriptors** or capabilities* name entries in a global **object descriptor table**.

* The term access descriptor was chosen over capability due to its close correspondence to an Ada *access*.

Each **object descriptor** in this table describes a segment of from 1 byte to 128K bytes in length. An object consists of two parts, one containing data and the other containing access descriptors. Each part may be up to 64K bytes in length. The one object descriptor for a given segment provides the physical base address and length of the segment, indicates whether the segment contains data or accesses, indicates what type of object it represents, and includes information needed for virtual memory management and parallel garbage collection. Each access descriptor (there may be many) for a given object contains rights flags that control the access available via that access descriptor.

The simplest type of object is generic for which no additional semantics exist. Other types of objects are recognized by the processor and are used to control its operation. Examples of these are processor, process, storage resource, and port objects. These objects are used by the 432 processor as a basis for providing a number of high level implicit operations and instructions. For example, ready processes are dispatched on processors automatically by the hardware via algorithms that involve processor, process, and dispatching port objects. Interprocess communication is provided by **send** and **receive** instructions that pass any access descriptor as a message via a communication port object. Likewise, memory allocation, user-defined types, and processor control are accomplished via instructions that involve other system objects.

Finally, the 432 supports small protection domains with **domain** objects. [9,10,11] These correspond to the package construct in Ada, namely, they are a structure for grouping and restricting accesses to the implementation of a module. The 432 subprogram **call** instruction performs the dynamic transition between domains, providing the proper addressing environment for any invoked subprogram via a **context** object. Much has been made of the cost of domain switching in a domain structured architecture. For comparison with other architectures, a domain switch on the 432 takes about 65 microseconds for an 8 megahertz processor with no wait state memory. This compares reasonably with the cost of procedure activation on other contemporary processors. Full details of the 432 architecture can be found in the 432 Architecture Reference Manual. [12]

3. iMAX Design Philosophy

For Intel, the 432 and iMAX are products primarily intended to be used by original equipment manufacturers in the construction of their products, rather than by than end-users. This means that support for minimum systems, range of application, and configurability are the most important iMAX goals. This is in marked contrast to a typical end-user system for which a particular application is defined (such as general purpose time-sharing) and whose facilities are targetted to that application. For such a system facilities needed to implement the application would be paramount. iMAX emphasizes breadth of support over depth.

iMAX is fundamentally a multiprocessor operating system, providing a tightly coupled environment in which all processors see a single homogeneous memory. The 432 hardware is designed to support multiprocessing in the standard configuration, and in fact makes the existence of multiple general data processors transparent to virtually all of the system software. With the bussing schemes designed for the 432, a factor of 10 in total processing power of a single 432 system is realizable. Multiple independent I/O subsystems provide a similar expansion for the I/O bandwidth of a single system. To support multiprocessing, therefore, it is merely necessary that the design of iMAX never assume that only a single processor is running. That is, all synchronization within the system must be explicit, never assuming that process priority or other scheduling artifact is sufficient to guarantee exclusion. Since this is a design principle that should be followed in a multiprogramming environment in any case, we will not comment further in this paper on the multiprocessing aspects of iMAX.

Given the 432 architecture, the relationship between iMAX and the 432 hardware is more preordained than that of most systems and their host hardware. iMAX is obligated to complete the model of computation supported in the hardware. Operations are provided in the 432 hardware for one of three reasons: they are time critical, thus benefitting from hardware implementation; they are security sensitive, thus requiring hardware enforcement; or they are complex in a way that benefits from special hardware structures on chip. iMAX is responsible for cooperating with the hardware to provide such remaining operations as initialization of complex objects, object maintenance, and object disposal. iMAX also extends the semantics of the hardware to provide a more convenient view of those abstractions that are built into the hardware.

Although some of the facilities provided by iMAX are actually realized as hardware primitives, iMAX provides a uniform external view of the 432 system through an interface that is expressed as a set of Ada specifications. Its users can be unaware of which operations have been implemented in hardware and which have been left to software.

Ada has been chosen as the systems implementation language of the 432 because its facilities complement the 432 architecture well. Packages in Ada provide a natural representation of type managers and map exactly the protected domain structure of the architecture. The specification/body distinction agrees with our desire to blur the hardware/software boundary and with the natural implementation hiding of the domain structure. The only area in which Ada falls short of our needs is that its design focus is a static, embedded environment. Since the 432 is intended for these as well as other, more dynamic applications, a few extensions[13] have been made to Ada to permit runtime type checking and dynamic package creation.

From an internal point of view, iMAX exploits the architecture to provide a more robust and flexible system than might otherwise be possible. The small protection domains supported by the language and the hardware are used to improve reliability. The uniform approach allows us to take full advantage of the duality between the language notions of data abstractions and the operating system notion of domains.

4. iMAX and Ada

The applications interface to iMAX is a set of Ada package specifications, each of which corresponds to a particular service provided by the system. This interface provides a uniform Ada view of both the underlying hardware and the iMAX extensions to it. Heavy use is made of generic Ada packages and in-line subprograms to provide an efficient but fully Ada typed view of the hardware. Unlike many systems for which calls to the operating system are very different from calls to other subprograms, the iMAX user sees no difference whatsoever between calling an operating system subprogram and calling some user-defined subprogram. This is particularly attractive for at least two distinct reasons. Compilers do not need any special mechanism for interfacing to the system. The standard calling sequences work for both system and user defined subprograms. Perhaps more importantly, any system interface can be mimicked by a user package. This makes it straightforward for a user to extend the system interface, trap certain system calls, or otherwise alter iMAX services.

As an example of how powerful this technique can be, we will consider the example of interprocess communication via the 432 port mechanism. This mechanism is more flexible than the Ada intertask communication model. It is used by the Ada compiler to implement the Ada model but is also available to the user who wishes the more general mechanism via a set of iMAX packages. The hardware defines a **communications port** object which functions as a queueing structure for interprocess communications. There are machine instructions available for sending and receiving messages via these objects. Full details of this model are provided in a companion paper. [14]

The simplest view of this mechanism is via the iMAX package specification Untyped_Ports, a fragment of which is shown in figure 1. The type *any_access* is predefined in the standard environment for the 432 and corresponds to an otherwise untyped access descriptor. Any Ada access type can be converted to this type but unchecked conversions are needed to do anything else with it. The type *port* is an Ada access to a hardware port object. Of the three subprograms specified in figure 1, Send and Receive will correspond to single instructions, while Create is software implemented. The Ada inline pragma provides efficient implementations of the first two.

The Ada code insertion facilities are used in the package body of Untyped_Ports to implement Send and Receive as the corresponding single instructions. This means that the compiler does not need any extraordinary knowledge of the 432 high-level instructions in order to permit the most efficient implementation. The Create procedure is implemented conventionally to provide proper construction of port objects. The 432 protection structures guarantee that only this package has the necessary access environment to create port objects. To the user of Untyped_Ports none of these details are important and a uniform view is provided to both the software and hardware parts of the port abstraction.

Since it is undesirable to force the user to escape from the Ada type system, another view of ports is provided via the generic package Typed_Ports, a fragment of whose specification is shown in figure 2. The user may create an instance of this package for any access type, thus creating a new Ada level type *user_port* that can be type checked at compile time to ensure that only objects of the specified *user_message* type can be sent.

```
package Untyped_Ports is

  function Create_port(
      message_count:    short_ordinal range 1 .. max_msg_cnt;
      port_discipline: q_discipline := FIFO)
    return port;
    -- Create a port with the given size and queueing
    -- discipline.
  procedure Send(
      prt:  port;          -- port to which a message is
                           -- to be sent
      msg:  any_access);   -- message that is sent
    -- The calling process will send the message to the
    -- specified port.  If the message queue of the port
    -- is full then the calling process will block until
    -- a message slot becomes available.
  procedure Receive(
      prt:  port;              -- port from which to receive
                               -- message
      msg:  out any_access); -- received message
    -- The calling process will receive a message from the
    -- specified port.  If no message is available the
    -- process will block until a message becomes available.
    -- The received message is returned to the caller.
  . . .
private
  pragma inline (Send, Receive);
end Untyped_Ports;
```

Figure 1: Package specification for hardware level ports.

The user of Typed_Ports thus maintains the advantage of strong compile time typing. The implementation of this package is in terms of Untyped_Ports and an unchecked_conversion from any_access to the user_message type. The inline facility allows the code generated for any instance of this package to be *identical* to that generated for the untyped port package. Thus the user of typed ports suffers no penalty relative to even a hypothetical assembly language programmer.

An important observation is that this approach is very general, needing no special compiler support. It is possible to take the idea of typed ports one step further in the 432 to provide the type checking dynamically at runtime. The implementation would require a few more generated instructions making use of user-defined types but would otherwise be the same as above. It should be apparent in this example that the consistency of the architecture, system, and language contribute greatly to reducing the set of things a user need learn about process communication. A similar effect is seen throughout the 432 system.

5. The Process-Memory Model of iMAX

A good example of the manner in which iMAX provides a smooth bridge between the base architecture implemented directly in the hardware and the set of user-visible operating system facilities can be seen in the process and memory model provided by iMAX. The 432 hardware provides the essential support for both processes and memory management. For process management, the hardware defines a process object which contains the necessary information for scheduling processes, dispatching them on any one of several potentially available processors, and sending them back to software when various fault or scheduling conditions arise. All hardware operations involving a process object occur implicitly, as the result of such events as time-slice end and successful message communications. For memory management, the hardware defines a storage resource object (SRO) which describes free areas of memory and provides the information necessary to allocate both physical and logical address space. Hardware operations involving memory management occur as a result of instructions such as *create object* which explicitly request a memory allocation. For example, assuming that sufficient free storage is available, it takes 80 microseconds

```
with Untyped_Ports;
generic
    type user_message is private;
           -- Ada "private" indicates that no internal
           -- details of this type are availble within
           -- this generic package.
package Typed_Ports is
  --  This package enables the user to create ports and do
  --  simple operations on those ports involving only
  --  messages of type "user_message".

  use Untyped_Ports;

  type user_port is private;

  function Create(
      message_count:   short_ordinal range 1 .. max_msg_cnt;
      port_discipline: q_discipline := FIFO;
    return user_port;
    -- A user_port with the specified message_count and the
    -- specified message queue discipline is created.
  procedure Send(
      prt: user_port;      -- port to which to send message
      msg: user_message); -- message that is to be sent
  procedure Receive(
      prt: user_port;          -- port from which to receive
      msg: out user_message); -- received message
private
  pragma inline (Send, Receive);
  type user_port is new port;
end Typed_Ports;
```

Figure 2: Typed access to the hardware mechanism.

at 8 megahertz to allocate a segment from an SRO via the creation instruction. It is important that this function be relatively fast since storage allocation plays an important role in an object oriented system. Actually a number of other system objects are involved in providing process and memory services, but these two suffice for this discussion.

iMAX provides operations to create and maintain both SRO's and process objects. It also extends the base architecture to further support the semantics of languages such as Ada.

To understand the latter role, consider the scoping and lifetime rules of objects in Ada. If a type is declared at the Ada library level then it exists forever. As a result, the lifetime of any object of that type is potentially infinite. Such objects may cease to exist only when they become inaccessible to any agent in the system. Proper implementation of these semantics require either an infinite capacity storage system or garbage collection. Types declared at deeper nesting levels than the library level come into existence anew whenever the scope of their declaration is entered and exist only as long as this scope. The lifetime of an object of such a type is constrained to be no longer than that of its type. This constraint is due to Ada's choice of name equivalence and is not applicable to languages that support structural equivalence of types. An object of such a type may never become accessible above the tree of dynamic environments rooted in the scope defining the type. A consequence of these conditions is that any object of such a type may safely be destroyed whenever the scope of its type is exited. Nevertheless, garbage collection may be necessary for objects of such a type if the lifetime of its scope is very long.

The 432 hardware and iMAX together provide exactly this model. Each object in the 432 has associated with it a level number which indicates the dynamic depth at which it is logically defined. Each context object (i.e., activation record) within a process has a level one greater than that of its caller. Each SRO creates objects with a fixed level number. The hardware ensures that an access for an object may never be stored into an object with a lower (more global) level number. The level numbers may be viewed as an indication of relative lifetime, where objects at level 0 are called ***global*** and exist forever while

objects with higher level numbers are called *local* and have progressively shorter lifetimes. These rules are actually sufficient to ensure that the lifetime rules expressed above for Ada are maintained even though the same level number may appear in the execution of independent processes.

iMAX uses these hardware facilities to provide a uniform tree structure encompassing both processes and storage resource objects. An SRO that creates objects at level 0 is called a *global heap* and is always available to a process. A process may create an SRO with a level number corresponding to its current depth called a *local heap* and then create objects from it. Since access to these objects will not escape their proper environment, objects may be destroyed whenever their ancestral SRO is destroyed, without leaving dangling references. This SRO will be destroyed automatically when the process returns above the call depth to which it corresponds. A more detailed explanation of this model can be found in [15].

Processes themselves are each created from an SRO and have their lifetimes constrained just as described for all objects. This corresponds exactly to the Ada task model. Likewise, the 432 model of interprocess communication corresponds to the lifetime constraints on processes, ports, and messages. A group of tasks communicate with each other via ports defined in a scope common to all tasks in the group. Objects passed through these ports are of a type whose scope is no less global than the scope of the port. The ports and messages will exist at least as long as the processes which are depending on them for communication.

Once again the coherence of the architecture model with those of the operating system and the language should be noted. iMAX uses the primitive 432 objects to build a structure which corresponds directly to a model of a typed, statically scoped language with pointers. At the same time, a user is free to use global SRO's exclusively, or other combinations of local SRO's, to build models with differing lifetime properties. All objects are subject to garbage collection; those allocated from local SRO's will be collected more efficiently whenever their ancestral SRO is destroyed.

6. System Configurability

As indicated above, configurability is an important design goal for a system like iMAX. For the main function of the system, iMAX uses two complementary approaches: selection of needed packages and alternate implementations of standard specifications. Once again, both the 432 hardware and Ada aid in achieving the goal. The domain structure also provides a convenient mechanism for supporting various levels of device independent I/O.

6.1. Process Management Via Selection of Packages

As an example of the first approach, consider the case of process management. The basic process manager of iMAX completes the model of processes embedded in the hardware by providing the functions briefly described above. It does not arbitrate conflicting requests on the processor resource, however. It makes directly available to the user the dispatching parameters of the hardware and users are free to overcommit or otherwise misuse these parameters. The basic process manager's control primitives were chosen so that process schedulers can manage the physical processing resources of the system without being aware of the logical structure of process trees described in the previous section. For example, it supports nested stopping and starting of processes. Each process has a count of the number of stops or starts outstanding against it which determines if it is currently runnable.

Since starts and stops apply to entire trees, a user wishing to control a computation need not be aware of the internal structure of that process, i.e., whether it is implemented in terms of other processes. These counts are maintained by the basic process manager. Control requests can be passed through a process scheduler based on the basic process manager without being tracked, even though they will ultimately have an effect on the set of processes ready to consume system resources. Whenever an individual process would enter or leave the dispatching mix as the result of start or stop requests, it will be sent to its process scheduler. The scheduler can then make resource decisions by regarding it as an individual process without concern for the logical structure of a computation of which it is a part. Of course this structure may be examined by the scheduler if desired.

Using this basic process manager, many resource control policies are possible. For example, the null policy simply passes through the dispatching parameters of the hardware and permits its users to commit them in any way they wish. This is completely acceptable for simple embedded systems in which the system load can be preevaluated.

On the other hand, it is clearly unacceptable in a multi-user environment where the processing resource must be allocated fairly. For this and other more complex applications a user-process manager may build much more complex policies on the basic process manager to provide a safer or more tailored application interface. The protection structures guarantee that only this second manager would then have access to the basic process management facility. The system is configured by selecting those packages that provide the facilities needed in a particular application: just the basic process manager, it plus some simple scheduler, or an arbitrarily complex resource controller.

6.2. Memory Management Via Alternate Implementations

As an example of the second approach to configurability, consider the case of memory management. Virtually all processes make use of memory management facilities via a standard interface that permits allocation of new objects. Few processes depend upon whether the underlying implementation includes swapping or not. A single Ada specification defines the common interface. This interface defines mechanisms corresponding to the stack allocation, global heap allocation, and local heap allocation described earlier. Both a swapping and a non-swapping implementation meet this specification but are optimized internally to the level of function they provide. Each may provide an additional management interface that can be used by resource managers or others that need information specific to the implementation. The system is configured by selecting one of the alternate implementations; most applications will not be affected by this selection. We have implemented the non-swapping version for the first release of the system, and are currently building a swapping version for the second release.

6.3. I/O Device Independence

iMAX is implemented entirely in a superset of Ada. The extensions are defined to permit full use of the more dynamic environment afforded by the 432 as compared to the very static one assumed by pure Ada. The major extension is the raising of packages to the status of types. This allows multiple instances of a module to be dynamically created and multiple implementations of a single package specification to coexist within a single system.

The clearest example of the use of this facility occurs in I/O. A single specification is defined for device independent input and another for device independent output. Each instance of an I/O device may have a distinct implementation. The user interacts with each device identically but the code is specific to the device. This is really a different approach from conventional device independent I/O because it avoids any centralized I/O control or interface. Any user can create a new device implementation which will behave identically to existing ones without in any way altering system code, say to update a master I/O device list or to add a new element to a *case* construct in the system I/O controller. We actually go one step further with this approach by requiring only that a device implementation provide the common device independent interface as a subset. Thus device dependent I/O fits smoothly into the scheme. Any device interface will consist of a domain in which the first set of operations are the device independent ones and any additional operations are more device specific. In fact, classes of devices may share a specification which includes more than the minimum set of device independent operations, thus providing class dependent but device independent interfaces.

7. The Internal View of iMAX

7.1. Hardware Type Enforcement

One interesting aspect of the implementation is the hardware enforcement of protection both at the operating system interface and within the operating system itself. This attribute of any hardware-implemented capability-based system has several ramifications within iMAX. First, a module's access is routinely limited to the objects which it manages. Thus, for example, the process management module has no access to memory management structures. Second, the object orientation of the system implies that at any given time, a package will generally have access to only a single instance of the type that it manages. For example, there is no central table of all processes in the system. Rather, the manager acquires an access for a given process object, either from the hardware dispatching mechanism or from a user, whenever it is asked to perform an operation upon it. Damage due to a machine error or latent program bug is limited to the particular object with which the module is dealing at a given moment.

This second property has an interesting side effect. Global system inquiries which are easily answered in most systems by consulting some central table become difficult to answer in this style of system. For example, the process manager does not know what all the processes in the system are. While it

would be possible to link together all processes, this would be problematic for garbage collection since all processes would then always be accessible. It is an interesting philosophical question whether such inquiries should be permitted; it is a convenient tenet of the capability approach to protection that they should not.

7.2. Hardware Type Enforcement

Another interesting aspect of the implementation is that the hardware type enforcement dictates that even objects that originate in applications coded in a language other than Ada are fully protected from misuse. Just as important for Ada programs is that objects are fully protected even when they pass through channels which might cause them to lose their compile-time type identity. An example of such a channel is any storage system. By the definition of Ada, if a storage system exists before the compilation of a package, then it cannot know of and therefore cannot preserve the type of some object that it is asked to store. In general, unchecked conversions need to be used to store and later retrieve objects, thus compromising type security. No matter what path a system object follows within the 432, its hardware-recognized type identity is guaranteed to be preserved and checked, either by the hardware or by object filing. [16] Moreover, via the user type definition facilities of the 432 such a guarantee is available to any user defined object type as well as to those object types recognized by the hardware.

7.3. Levels and Abstractions in iMAX

The strong adherence to notions of data abstraction in the design of iMAX provide a very clear example of the difference between levels and abstractions discussed by Habermann *et al.* [17] Internally, the system is constructed as a set of Ada packages each of which provides a well defined abstraction. Even within the system the inter-package interfaces are rigidly enforced. Only the well-defined operations defined in the package specifications are available to the other parts of the system.

It is quite reasonable that a set of abstractions be mutually dependent at the module level. For example, there are interdependencies between parts of memory and process management. On the other hand, it is important that the design of the system not include any circular dependencies among functions that might cause deadlocks to occur. Additionally, the clean virtual environment provided at the user interface level of the system must be built up in stages. For example, processes at the user level should be unaware of the possibility that a segment might be being moved and therefore be inaccessible for some period of time. Processes deep within the system, on the other hand, may depend on the fact that such a situation will not arise.

To solve these problems, the implementation of iMAX defines a set of levels which dictate what operations are permitted to processes at that level. Processes below level 3 of the system, for example, are in general not permitted to fault. Processes at level 2 are actually permitted a limited set of timeout faults while those at level 1 are not permitted even these. To avoid dependency couplings, all communications between levels 2 and 3 of the system must be asynchronous and upward communication must never depend upon a reply. The implementation of a given abstraction may span several levels. These design guidelines span all abstractions and thus represent an orthogonal way of viewing the internal structure of the system. From one view each function is a part of the abstraction to which it relates regardless of the constraints under which it operates. From the other view, each function operates at the level in the system determined by those constraints.

8. Issues in the Design of an Object-Oriented System

In this section two issues are discussed that arise in the design of a system such as iMAX: garbage collection and object finalization. These will in fact be issues for any system that ruthlessly follows the approach to design implied by Ada. A third issue, object filing, is discussed at length in a companion paper. [18]

8.1. Garbage Collection

The first issue is garbage collection. The general provision of global heap allocation in most modern languages demonstrates the desirability of removing questions of memory allocation from the programmer's concern. Unfortunately, explicit deletion of heap objects is prone to dangling reference problems. Furthermore, when an object is part of a complex data base of information, it is often difficult or impossible to know when the object should be deleted. The 432 approach to this issue is to remove questions of memory deallocation as well as questions of memory allocation from the programmer's concern. We noted above that objects whose types are defined at the library level may only be reclaimed via garbage collection. iMAX provides a system-wide parallel garbage collector based upon the algorithm of Dijkstra *et*

al. [18] To support this, the 432 hardware implements the ***gray*** bit of that algorithm, setting it whenever access descriptors are moved. When there is a natural expression within the programming language of a constraint upon the lifetime of an object, iMAX can take advantage of this to optimize deallocation via the local heap memory allocation strategy mentioned above.

The iMAX garbage collector is implemented as a daemon process that globally scans the system. It requires only minimal synchronization with the rest of the operating system. The local heap and level mechanisms effectively partition the system into nested sets of objects based on lifetime. Since object references can never escape from the level of the nest at which they were created, a local garbage collection strategy could be added to our global one. It would be possible to perform garbage collection on a local basis, either asynchronously or synchronously, but we have not chosen to do this until we have data that suggests that it would be worthwhile.

It is perhaps worth noting that the Ada literature is curiously ambivalent about facing up to the issue of garbage collection. This has been noted in a different context when Ada was evaluated as a language for the implementation of AI applications. [19] Avoidance of the problem will likely lead to either contorted programming styles or to the continued presence of dangling pointer problems in complex systems.

8.2. Object Finalization

Another interesting issue in the design of an object-oriented system is object finalization and lost objects. While most languages provide some form of initialization for structures, they do not address finalization of those same objects. So long as the objects in question do not have any dual in the real world, this may be an acceptable position. When an object represents a physical resource of some sort, however, it becomes very important that its type manager be able to describe how that object is destroyed as well as how it is created. Consider for example an implementation of a tape drive in which each drive is represented by an object of type **tape_drive**. In Ada terms, this would be a private type. A user requests from the managing package a tape_drive instance, calls operations in that package to use it and eventually to close or return it. If, however, the user loses access to the object through accident or intent, it will be garbage collected and the system will be short one tape drive. This is what we mean by a lost object.

While narrow solutions may be available for individual cases, they often will pervert a natural implementation design in undesirable ways. A general solution would permit a type manager to guarantee that an object is properly disassembled when it becomes garbage. iMAX provides the notion of a destruction filter for exactly this purpose. Since all objects may be typed, the garbage collector can recognize when an object of a particular type has been found. A type manager can specify to the system via a type definition object that it wishes to have an opportunity to see any of its objects as they become garbage. The garbage collector will manufacture an access descriptor for such objects and send them to a port defined by the type manager. The first release of iMAX uses this facility only to recover lost process objects. The next release will make the facility generally available in the context of object filing.

9. Project Status

The first release of iMAX is now undergoing field test and will be shipped for general customer use in early 1982. Portions of this version have been running in our laboratory since late spring of this year. Since a fairly rigorous methodology of design and code review was followed in its implementation, most of the problems that we encountered during debugging have been the result of compiler or hardware problems. As with most new computer development efforts, the operating system has been the first major test for all other system components. The first release of the system is non-swapping and concentrates on providing a development and debugging base for customer applications. Most of the detailed design of the second major release of the system is complete and implementation is now underway. This release includes swapping support and object filing.

10. Conclusions

A paramount concern in the design of the 432 system has been the conceptual uniformity of the architecture, operating system, and language. In addition to the aesthetics of this uniformity, it has a number of practical benefits. These include a more flexible and safer programming environment. Extensibility is enhanced because the system software is not fundamentally different from user software. Once the object paradigm is learned, the user can apply it to all aspects of the system. Special treatment is not required when crossing boundaries in the system between hardware and software, language and system, virtual storage and files. The overall learning burden is reduced.

By providing the notion of domains and capability/object-oriented addressing in the architecture, proper support has been given to such language issues as garbage collection and dangling pointers. By reflecting the module structure of the language in the system, configurability has been enhanced. Overall, the 432 represents an entire system constructed around the single notion of supporting an object oriented approach to program design. iMAX plays a key role by completing the architecture within an Ada framework to provide a comprehensive base for the design of advanced computer applications.

Acknowledgements

This work has benefited greatly from the creative environment generated by all of the members of Intel's Special Systems Operation. Particular note should be paid to Justin Rattner and George Cox, principal architects, to Konrad Lai and Dan Hammerstrom, who were responsible for the microcode of the processor, and to Gary Raetz, for his work on an earlier prototype of iMAX. Special thanks should also go to the Ada compiler implementors, Steve Zeigler, Bob Johnson, Jim Morris, and Greg Burns, for their outstanding work in bringing up a very comprehensive Ada compiler in an amazingly short period of time.

References

1. R. S. Fabry, "Capability Based Addressing," *Communications ACM* **17**(7) pp. 403-412 (July, 1974).
2. K. C. Kahn and F. J. Pollack, "An Extensible Operating System for the Intel 432," *Proceedings Compcon Spring 1981*, pp. 398-404 (February, 1981).
3. W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, and F. Pollack, "HYDRA: The Kernel of a Multiprocessor Operating System," *Communications ACM* **17**(6) pp. 337-345 (June, 1974).
4. W. Wulf, R. Levin, and S. Harbison, *Hydra: C.mmp: An Experimental Computer System*, McGraw–Hill Book Company (1981).
5. R. M. Needham and R. D. H. Walker, "The Cambridge CAP Computer and its Protection System," *Proceedings of the 6th Symposium on Operating System Principles*, (November, 1977).
6. M. V. Wilkes and R. M. Needham, *The Cambridge CAP Computer and its Operating System*, Elsevier North Holland (1979).
7. A. K. Jones, R. J. Chansler, I. D. Durham, K. Schwans, and S. R. Vegdahl, "StarOS, a Multiprocessor Operating System for the Support of Task Forces," *Proceedings of the 7th Symposium on Operating System Principles*, pp. 117-127 (December, 1979).
8. B. Lampson and H. Sturgis, "Reflections on an Operating System Design," *Communications of the ACM* **19**(5) pp. 251-265 (May, 1976).
9. D. Cook, "In Support of Domain Structure for Operating Systems," *Proceedings of the 7th Symposium on Operating System Principles*, pp. 128-130 (December, 1979).
10. P. J. Denning, "Fault–Tolerant Operating Systems," *Computing Surveys* **8**(4) pp. 359-389 (December, 1976).
11. T. A. Linden, "Operating System Structures to Support Security and Reliable Software," *Computing Surveys* **8**(4) pp. 409-445 (December, 1976).
12. Intel 432 GDP Architecture Reference Manual, Intel Corporation 1981.
13. S. Zeigler, N. Allegre, D. Coar, R. Johnson, J. Morris, and G. Burns, "The Intel 432 Ada Programming Environment," *Proceedings Compcon Spring 1981*, pp. 405-410 (February, 1981).
14. G. W. Cox, W. M. Corwin, K. Lai, and F. J. Pollack, "A Unified Model and Implementation for Interprocess Communication in a Multiprocessor Environment," *Proceedings of the 8th Symposium on Operating System Principles*, (December, 1981).
15. F. J. Pollack, G. W. Cox, D. W. Hammerstrom, K. C. Kahn, K. K. Lai, and J. R. Rattner, "Supporting Ada Memory Management in the iAPX-432," *Symposium on Architectural Support for Programming Languages and Operating Systems*, (March, 1982). Submitted
16. F. J. Pollack, K. C. Kahn, and R. M. Wilkinson, "The iMAX-432 Object Filing System," *Proceedings of the 8th Symposium on Operating System Principles*, (December, 1981).
17. A. N. Habermann, L. Flon, and L. Cooprider, "Modularization and Hierarchy in a Family of Operating Systems," *Communications ACM* **19**(5) pp. 266-272 (May, 1976).
18. E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. M. F. Steffens, "On-the-Fly Garbage Collection: An Exercise in Cooperation," *Communications ACM* **21**(11) pp. 966-975 (November, 1978).
19. R. L. Schwartz and P. M. Melliar-Smith, "On the Suitability of Ada for Artificial Intelligence Applications," SRI Technical Report (July, 1980).