

# The AppCard: a Migrating Processor Card for General-Purpose Computing

Vsevolod V. Panteleenko, Theron W. Nelson and David L. Cohn

Distributed Computing Research Laboratory  
Department of Computer Science and Engineering  
University of Notre Dame  
Notre Dame, IN 46556, U.S.A.  
E-mail: {vvp,twn,dlc}@cse.nd.edu

## Abstract

*In this paper, we present a new approach to mobile computing based on an all-silicon computer. It facilitates application and data migration as a user moves between heterogeneous, disconnected machines. General-purpose applications permanently run on a removable processor card, called an AppCard. An AppCard connects to a host computer through a standard interface, such as a PCMCIA slot. The card has its own operating system which provides basic services, but it relies on the host for many resources, such as a user interface and a network connection. This paper discusses the design and implementation of an initial AppCard prototype. It includes early performance measurements and presents ideas on further development in this area.*

**Keywords:** mobile computing, embedded systems, operating systems.

## 1 Introduction

Mobile users who deal with multiple computers during the course of a day often experience a loss of productivity when switching machines. The new host may not run a given application, critical data may not be available, or data generated on one machine may not be compatible with the next. Continuous use of a laptop computer or Personal Digital Assistant (PDA) might solve these problems, but laptops are bulky and inconvenient to carry, and tiny PDA user interfaces are acceptable for only a small number of applications.

A better solution involves a new mobile computing paradigm. The paradigm recognizes that classic portable systems are approaching a size barrier. Even though technology continues to decrease the size of their silicon components, user interface devices such as displays and keyboards cannot be further reduced and retain full functionality. Also, there are many applications that do not require all of the computing capabilities offered by a fully configured portable system. The proposed mobile computing paradigm subdi-

vides computer system functionality into both hardware and software components. It uses simplified computers, made entirely of silicon parts, to permanently contain and run sets of applications. These devices can either shrink as technology advances, or, once they reach a small enough form factor, increase in functionality. They provide a way to dynamically compose computing devices. Such computers interface with stationary host systems in a standard way and make use of the host's I/O. In a world full of hosts capable of interacting with such devices, a new level of portability would be achieved.

This paper discusses the development of prototype hardware and system software for an all-silicon computer that we have called the *AppCard*. The next section further explains the AppCard and its advantages, and the subsequent section is a survey of related work. The following sections discuss the design and implementation of the prototype AppCard and its initial system software. Finally, we conclude with some performance measurements, a discussion of further work needed in this area, and key ideas learned from the project.

## 2 The AppCard Concept

An AppCard is an all-silicon computer, approximately the size of a credit card, which consists of a processor and persistent memory. It is permanently assigned a set of applications. The persistent memory holds both the application code and the user data. It plugs into a standard port, such as a PCMCIA Card<sup>1</sup> slot, and relies completely on the host computer to provide a user interface and other peripherals. Because of its small size, a person can conveniently carry multiple important applications and files loaded on a single AppCard. The AppCard's persistent memory stores application

<sup>1</sup>The PCMCIA Card is an interchangeable Integrated Circuit Card that meets the PC Memory Card International Association (PCMCIA) standard. The standard was initially focused on IBM PC-compatible systems, but is now committed to allowing a variety of computer types and non-computer consumer products to freely interchange these cards [14].

execution state information during periods of disconnection. When the AppCard is connected to a new host, its processor awakes and the applications continue from where they left off. In this way, applications and user data can *migrate* to a new host computer with the mobile user.

The host computer could be many different things: a PDA, a laptop, a personal computer, a workstation, or possibly an application-specific device such as a telephone, an automatic teller machine, or a retailer's point-of-sale machine. All that is required is that the host have an AppCard slot and that it provide the base server software to handle communication and other requests. The host need not even have a normal keyboard and display, as the AppCard can query the host to see what types of I/O are available. For example, the host's display device could be a color monitor capable of running a graphical user interface, or it could be a single-line LCD capable of text only, or it could be no display at all. The AppCard's requests will be mapped as well as possible to a given host's capabilities.

We see several advantages of such systems over simply carrying application state on persistent storage devices like disks or FLASH memory. Persistent storage devices can store application state for migration to a different computer but the functionality is completely dependent on the computer that will run the application. The application cannot run on different architectures or an improperly configured system. By contrast, the AppCard hardware need only implement the functionality required by its applications. It could provide hardware support for specific application requirements like handwritten text recognition or vector processing for numerical computation. Different AppCards can be built on different processor architectures and still work together. Software distribution using AppCards implements a plug-and-play approach with full control over software copying. Although security issues differ for each system component, security is generally easier to achieve for AppCards since they do not rely on the host to control application state. Unlike a regular personal computer, a system consisting of hosts and a set of AppCards can be easily functionally extended. A user can configure an appropriate system by composing a set of AppCards and easily change or upgrade the system capability by changing particular AppCards.

The first candidates among the applications to be ported to the AppCard platform are personal productivity tools like a mailtool, a word processor, a day planner, etc. A user obtains a familiar environment regardless of what platform he or she is currently working on. The AppCard may also contain authentication and cryptographic software that would assist in logging into the host computer and accessing protected data. Other applications that would benefit from the AppCard platform are those that are built specifically for a distributed environment, such as a distributed meeting tool, those that require some specific hard-

ware capabilities not expected to be found at a regular host, such as fast hardware-assisted animation software, or those that are responsible for interaction with specific devices, such as banking applications.

Of course, there are limitations to the AppCard paradigm. Some applications, especially those that require intensive computational or memory resources, may not fit. For compatible applications, the dedication of hardware, even when the applications are active, may seem a waste. But as the cost of silicon components decreases, the price of an AppCard will soon be smaller than the price of the software it contains.

### 3 Related Work

The AppCard's method of providing user mobility has its roots in several projects. Two designs, Wit [18] and Rover [8], allow application downloading to a mobile computer. However, they both use the notion of a home host from where the application may be downloaded to several mobile devices. The other main difference from the AppCard design is that a mobile application is presented a special application environment which is unlike the AppCard's API is quite different from what is usually present for regular applications.

The concept of putting a processor on a small, removable card is not new. In fact, the *smartcard*, a credit-card sized computer that is normally composed of a single integrated circuit chip between two thin layers of plastic, has been around since 1980 [11]. However, smartcards are used for very specific applications, and do not have the hardware or software to run general-purpose user-interactive applications or to take full advantage of the host's resources.

An alternative approach that addresses the same goal of the AppCard--facilitating movement of users in a heterogeneous environment--is the distributed window system. One such system is X-Windows, which divides the user-interface from the application, allowing an application running on the *X-client* to use a different machine on the LAN as the X-server display. Columbia University's Xmove project even allows an active X display to follow the user to a new host using unmodified client and server software [17]. Because the X-client and X-server software can be ported to different architectures, it succeeds in carrying out the illusion that an application has migrated in a heterogeneous environment (even though the application continues to run on the original host, and only the display migrates). Unlike AppCard, a distributed window system does not allow migration between disconnected hosts.

There are many projects that address the problem of file system access for mobile computers. We can mention several [3,9,16]. All of them try to provide the single image of the file system for mobile hosts. Although these projects have different goals than the

AppCard has, the technology could be useful to provide file access for AppCard applications. The other technology that could be used in the AppCard design is the mobile IP [7]. It could provide IP access for AppCard applications that is independent from the host to which the AppCard is connected.

## 4 Design Issues

The novel design issue from a software view is that an application running on the AppCard makes use of resources located on the card (memory, IPC connections, local files) and on the host system (keyboard, display, network connections, remote files). Further, the resources available to an application can change dynamically as the card is moved between hosts.

Applications could be developed specifically for the AppCard environment with application developers responsible for managing both remote and local system resources. However, we consider the more general approach in which applications expect services typical of those found on regular platforms. This will make it easier to port existing applications designed for classic operating systems to an AppCard.

The choice of the AppCard operating system plays a key role in the AppCard software implementation. The operating system would ideally be powerful enough to support multitasking/multithreading and standard APIs, but would take up very little memory, a scarce commodity on the AppCard. In addition to the operating system, other software components must be available on an AppCard. Among them should be those that provide an AppCard application user and file interfaces with the host.

### 4.1 System Services

Application interfaces vary from one operating system to another. We have used common features from many operating systems as the basis for the AppCard API design. The API includes several interfaces: program control, user, file system, and communication. Many of the API requests are actually transferred to the host operating system for service.

#### *Program Control Interface*

We collect under this heading several basic services: task/thread control, memory management, and inter-process communication (IPC). Requests for these services cannot be transferred to the host and must be served by the AppCard kernel. They are also the basis for other interface implementations.

#### *User Interface*

An AppCard application uses a windows interface as the primary way to interact with a user. Because hosts may have different operating systems, different host

windowing systems might be available to the application. Nevertheless, most existing window systems or toolkits have enough similarity to allow definition of a neutral window interface and translation of this interface into the particular host window interface by a given host server. When the host does not have windowing system, some subset of the user interface API must be able to interact with the user through a simple device such as a single-line LCD.

There are two types of windowing systems and, hence, two types of window applications. One type, which we will call *monolithic*, integrates a window management module with a set of *widgets*. The widgets include menus and dialogs and are built above the management module. Examples of this type are MS Windows and Presentation Manager [12,15]. The second type includes pure window management systems, the most popular of which is X Windows [2]. Applications for such systems are usually built by some toolkit, which actually provides a set of widgets. Code for such widgets, as well as other support, is linked to the applications.

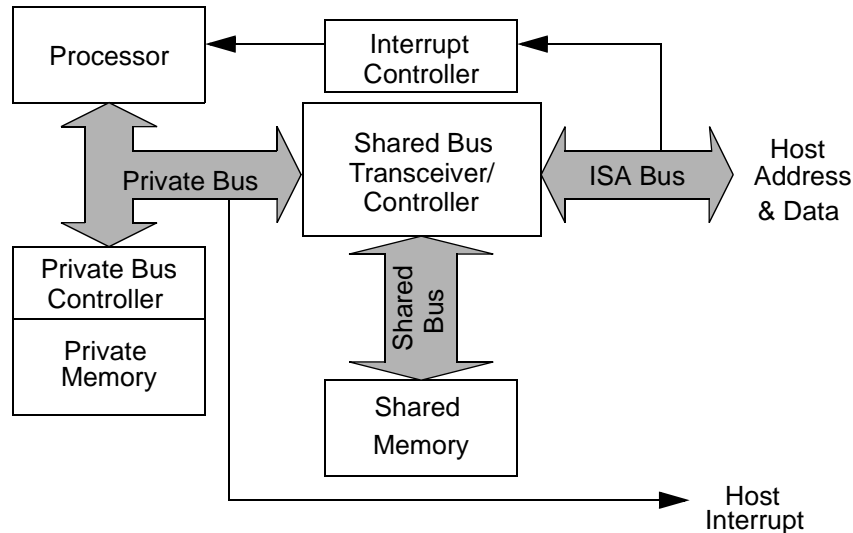
Although choosing the X Windows system as a standard base was very tempting, a number of factors compelled us to choose a monolithic window system instead. The first was application size. When we use a monolithic approach, we do not need to link all of the supporting windowing code into the application. We found it possible to further reduce application size by assigning part of the windowing code and resources to the host side. The second reason was that the most suitable type of host for the AppCard approach is a personal computer, which usually has a monolithic window system. Finally, we could not use the X interface directly because it would not support AppCard movement from one host to another. Our choice of the monolithic window system does not mean that we are not able to interface to a non-monolithic one. In this case, the host server would have to provide additional functionality using some toolkit.

#### *File System Interface*

There is no hard drive expected to be on an AppCard. Nevertheless, local data storage that is visible as the file system would be required for many applications. There has been significant research addressing file caching on the mobile devices for a single image for file system [3,9,16]. Most of these projects use standard file operations as the interface for applications. This type of technology with a standard interface for the AppCard would be preferable in the future.

### 4.2 AppCard Mobility

There is a further issue that deals with AppCard mobility. When an application is moved from one host to another, the current state of the application must be



**Figure 1:** Hardware Components of AppCard

restored at the new location. This means that the interface software which resides on an AppCard must keep track of resource usage status transparently to the application in order to provide this information to the new host. Consider, for example, opened files. When the host is changed, all files currently opened by AppCard applications must be reopened at the new host (assuming that they are accessible there) and file pointers must be properly set. But even this is not enough in some cases. If some file descriptor is created by the first host operating system and then given to the application, the same file descriptor must be valid at the new host. However, it cannot be guaranteed that operating system at the new host will use the same descriptor for the reopened file. Similarly, if the application has received a window handle from the window system server, this handle must work on the window at the new host. As usual, the solution to this problem is one extra level of indirection. The AppCard user interface software provides host-independent handles to the application and maps them to the actual host-dependent handles. In the next section, we describe how these issues affected our design.

## 5 Implementation

This project had several goals in the design of AppCard system software. First, it set out to show that it is possible to build software which provides a set of standard services partially locally and partially remotely, leaving this division transparent to the application. The second goal was to build software that would hide changes in location and host operating system type from the application. Finally, these two goals were to be approached without significant loss in application performance.

In the initial design, we did not try to build a full-scale programming environment. Rather, we built an

application program interface that provide just the basic capabilities. However, this environment is sufficient to construct applications that are complex enough to test the idea.

In order to assess the performance of applications running on the AppCard, a hardware prototype had to be designed and built. Its design permits disconnection and reconnection to different hosts, but the actual implementation does not. Thus the mobility aspect of the system has not been fully implemented.

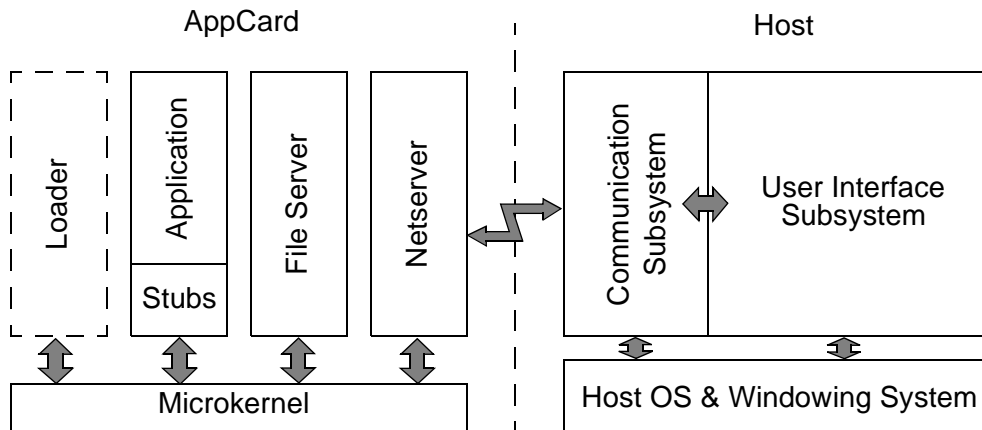
### 5.1 Hardware Prototype

The first prototype board is built as an ISA<sup>2</sup> card for an IBM-compatible PC-AT host computer. The ISA bus was designed as an expansion bus, and its architecture does not allow powered-on insertion and removal of adapters. However, it does offer a well-known and easy-to-work-with host interface for the prototype card. It is very similar to the PCMCIA PC Card standard bus interface, which is the target interface for AppCards and which does allow hot connection and disconnection of cards.

Figure 1 shows the hardware components of the AppCard prototype board, each of which is described briefly below.

The Processor block on the prototype AppCard is an Intel 80376 embedded processor clocked at 8 MHz (half the external frequency) [6]. The 80376 has a 32-bit internal architecture, and a 16-bit external bus. (Thirty-two bit, double-word data is automatically split into two 16-bit accesses.) The 80376 is essentially an 80386SX minus the virtual 8086 mode and paging

<sup>2</sup>The ISA bus is composed of a 16-bit data bus and a 24-bit address bus, allowing for a 24 MByte memory space and a separate 64 KByte I/O space. The maximum clock rate is 8MHz allowing for a peak bandwidth of 8 MBytes/second. [4]



**Figure 2:** AppCard Software Structure Design

mechanisms. The processor is compatible with the Intel 80x86 family of processors, which allows the AppCard prototype board to draw from an abundance of existing 80x86 software and tools.

The board contains private and shared with the host memory. The Private Memory block contains 1 MByte, and the Shared Memory block has 128 KBytes. All memory on the prototype is SRAM to simplify design, and to allow future versions to add battery-backed persistence. The Shared Bus Transceiver/Controller serializes host and AppCard accesses to the Shared Bus with a sharing granularity of a single bus cycle. The Processor can access both Private Memory and Shared Memory with zero wait states; however, accesses to the Shared Bus are sometimes delayed to wait for a host access of shared memory.

The AppCard and host can interrupt each other by writing to a special *interrupt port* in their respective I/O spaces. The card has a full Interrupt Controller, but only one other source of interrupts: the programmable interval timer used by its operating system for time-slicing.

## 5.2 System Software

The AppCard system software is structured as a microkernel and a set of servers. The public-domain operating system MMURTL [1] with multitask/multithread support was modified to be the microkernel. (MMURTL was based on paged memory, and the 80376 only supports segmented memory. Thus, substantial changes in MMURTL's memory management were required.) With the AppCard's limited memory, MMURTL's small size made it suitable as a base for the card's operating system. (The total size of the code and data segments for the modified MMURTL kernel is 64 KBytes.)

The microkernel is responsible for four main parts of the program control interface: task/thread management, memory management, IPC, and interrupt/timer

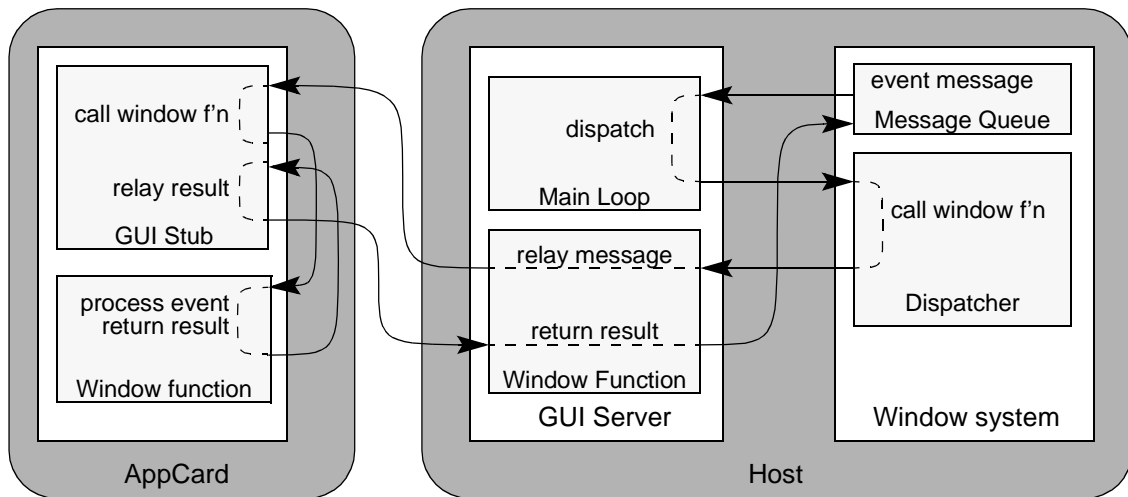
management. In our implementation we also included a name server. All other functions are supported by a set of servers that run in separate address spaces and at the same protection level as regular applications. All interaction with them goes through IPC and is based on message passing.

As we can see from Figure 2, the microkernel provides services to applications as well as to servers. Above the kernel we can see the Netserver, which is responsible for communication with the host and transfers most of the service requests to it. Another server is the Loader which is actually used only during the process of initialization and can be discarded later. The last server is the File Server which manages local file storage and also provides a uniform view of the host's file system regardless of where files are actually stored. Although it is part of the design, the File Server has not yet been implemented. Basic file operations that allow applications to access host files are currently provided in libraries. In the future, an approach of porting an existing mobile file system that supports disconnected operation might be more productive.

System services are accessed either by direct kernel calls or through IPC. However, a set of stub routines hides the details of service access through IPC. The stub routines are in a library that is linked into the application. The application makes a normal procedure call, and a stub translates the calls into a message. The message may be destined for a local server or for a server on the host. Thus, by using these stubs, the application sees a set of APIs similar to those found on a standard platform, and the details of communicating through the netserver to the host are hidden from the application developer.

### *Netserver*

All communication with the host is directed through a user-level task called the Netserver. Its functionality can be divided into two layers. The top layer presents a



**Figure 3:** User interface software structure

port communication paradigm to applications and host server subsystems. Its function is to multiplex the ports and divide messages sent through these ports into packets. The bottom layer manages the actual physical connection and provides a reliable packet transmission mechanism. The functionality of both of these layers was implemented such that the total size of the data and code segments for the netserver is 28 KBytes.

To implement the lower layer, a simple packet-passing protocol was built on top of the shared-memory hardware. There are two packet FIFO buffers in shared memory, one for packets traveling in each direction. Each buffer has a pair of shared pointers that mark the first and last packet in the buffer. After placing a new packet into the buffer and updating the shared pointers, the communication thread fires an interrupt to the other processor to indicate a packet-send event.

#### *Host Communication Subsystem*

The host server software is currently built as an OS/2 application and uses regular procedure calls for subsystem interaction. Each AppCard server has a counterpart subsystem on the host. The host's communication subsystem has functions similar to those of the AppCard netserver. It invokes other host services on behalf of the AppCard or requests services from the AppCard. The file service subsystem provides an interface to the host file system. The user interface subsystem enables window support for the AppCard application. It can request services from the host window system, or can send requests from the window system to the AppCard.

#### *User Interface Subsystem*

Graphical user interface services are based on communication between the AppCard and the host. The sys-

tem is composed of three primary pieces as shown in Figure 3: A GUI Stub on each AppCard application provides access to GUI services, a GUI Server is implemented as a host process that interacts with a local Window System. The host-side software is window system specific; the AppCard-side portion is not.

The local window system treats the GUI server process as a normal GUI client. As usual, when an interface event occurs in an appropriate window, the window system queues a message for the main loop of its client. This message is retrieved by the GUI server and dispatched back to the window system. The window system calls the indicated window function which it thinks will respond to the event. Instead, the GUI server's window function relays the message to a stub on the AppCard application, indicating which window is involved. This stub, in turn, hands the message to the proper local window function which does the actual event processing. The window function may need to ask for further service from the window system. If so, it uses the stub to relay its request back to the host.

The responsibility for translating events from the host window system to those of the AppCard window system lies with the GUI server. As a result, the AppCard GUI is essentially a least common subset of several popular GUIs.

Before any GUI service can be invoked or any message received, an AppCard application thread must initialize the system with `WinInitialize()`. This asks the host to create a message queue and service thread for this client. Then, to start processing messages, the AppCard thread invokes `WinEnterLoop()` that switches the corresponding host thread from server mode to client mode. Thus, the host thread can initiate a conversation with the AppCard when messages are available. Meanwhile, the AppCard application thread waits in the GUI stub for a message

from the host. When the message is received, the stub calls the corresponding window function.

When processing a message, the AppCard application thread might request other services from the host. It then sends a message to the host indicating whether this is an RPC to the server and not just a response to the previous message. The host thread then either performs the requested action or dispatches the next window message.

Like any window application, an AppCard application has a set of associated window system resources. The system currently supports menus, accelerator tables, bitmaps, icons, text string and some form of dialogs. In the prototype system, they are OS/2 Presentation Manager specific and cannot be ported easily to other window systems. However, the set of window system functions is a subset of those found on various window systems. They include groups that deal with:

- window creation/destruction
- graphical functions to manage screen output
- input from the keyboard and the mouse
- functions to control the window system resources

### 5.3 Moving the AppCard to Another Host

Although prototype AppCard is not mobile, mobility did influence the software design. When an AppCard is moved, the new host must restore the host-side application state. This state consists mainly of file and window subsystem interface data. All other host server subsystems can be reset to an initial state when the AppCard is connected.

There is another issue closely related to the mechanism of state restoration. Much system information that is passed to an AppCard application is host dependent. For instance, file and window handles are chosen by the host and, when these resources are restored by a new host, the old handles are no longer valid.

Both problems are solved by keeping track on the AppCard side of those opened resources for which this problem exists. This information is managed by interested application stubs and servers. (Actually, only the file server and the user interface stubs.) When a new resource is created, information about it is saved in the appropriate stub or server. For instance, when a new window is opened, the actual handle information and the window parameters are saved by the user interface stub. The application is given a local handle chosen by the stub. When an application request contains this handle as a parameter, the stub substitutes the actual handle. When the AppCard is moved to a new host, the communication server requests state information from all the stubs by sending a message to a specially allocated IPC port (called an *exchange* in MMURTL). Each interested stub then requests necessary service

from the new host. For example, the user interface stub would open a new window with the parameters that it saved. Thus, the handle that the application initially received before would still be valid.

A specific window management issue is that there is no need to provide an explicit way to restore window contents. The nature of the window system is that the system can request restoration at any time by invoking the proper window function request. The first thing the system does when a new window is created is to request that the window be drawn. In this way, the old window will be moved to the new host.

## 6 Results

This section lists early performance measurements for round-trip message-passing from an AppCard application to the host server. It also gives times for AppCard calls to some basic window system routines and compares them to measurements taken for the same application running directly on the host. All measurements were taken with the AppCard prototype plugged into the ISA bus on a HP Vectra QS16 personal computer, which has a 16 MHz Intel 386 and 8 MBytes of RAM. The host's operating system is OS/2 3.0, so all window system calls on the AppCard are mapped to a corresponding Presentation Manager routine.

### 6.1 Message-Passing Performance

For AppCard-host system service invocation, we have measured the time for a null-call which consisted of sending a message of four words (16 bytes) to the host and sending a message of one word back. Timing for this experiment revealed a message-passing latency of about 9.5 msec on average. For large messages (order of 64 KB), throughput of the communication channel between the AppCard and the host is about 485 KB/sec. The above figures may be compared to the TCP/IP performance for a Gateway i486 33MHz computer with 3Com 3C503 Ethernet interface (on a private 10 Mbps ethernet) running Mach 3.0 and Unix server [10]. On this system, round trip latency for short messages (order of 1 Byte) is about 4 msec. Throughput for the large messages is 415 KB/sec.

### 6.2 Window System Calls

Table 1 lists the time needed by a simple text editor running on the AppCard to perform several window operations. For comparison purposes, the times are also given for the same application running directly on the host. Even though some of the window operations take several times longer on the AppCard, the general response time was still acceptable for the simple text editor application. For instance, processing 80 characters from the keyboard takes 6.7 times longer on the AppCard than directly on the host; however, this slower processing rate still translates to 32 characters

Window Operation	Host	AppCard
Create and destroy text editor window (title, scroll bars, menu bars)	0.94 sec	1.38 sec
Process 80 characters from keyboard (no scrolling involved)	0.37 sec	2.47 sec
Process 1000 characters from keyboard	16.0 sec	43.7 sec

**Table 1:** Application Execution Times

per second, which is far beyond typical human reaction times. Currently, pointer device input is handled entirely on the host by system-provided window procedures, so the performance of pointer event processing by the AppCard application has yet to be tested.

## 7 Conclusions and Future Development

Experience from this project has shown us several areas for possible improvement. Very early into the project we decided to use a processor that does not support paging. The AppCard had no secondary storage, and it seemed that paging was unnecessary. However, this made the kernel harder to implement because physical space could only be allocated with the bulkier segments. Applications that frequently use dynamic allocation of memory can quickly fragment the physical memory on the card.

Our performance measurements for message-passing latency times were slower than hoped. We believe much of this latency is due to task-switching and IPC times, especially with the system software being implemented as a microkernel with separate user-level servers. Much of the kernel was implemented in the best way to keep the kernel size small, not fast. We now feel that concentrating more on performance in the specific areas of IPC and task switching could possibly increase performance with only a small increase in kernel code size.

The project showed that although it is possible for an application to run on a removable card and rely on the host for all I/O, the effectiveness of the final system greatly depends on how the interface between the client and the server is defined. For instance, the OS/2 Presentation Manager window system is defined in such a way that all events directly or not directly related to a window are passed to the window procedure. Every mouse movement causes the sending of a message to a window. This means that the implementation of a full-scale distributed PM on the AppCard would cause substantial degradation in performance. On the other hand, an implementation of a slow interface with much encapsulation of service like a file system should perform well. We think that in this design

we have found a good balance between implementing full-compatibility with the operating system API and a building completely different application environment. At the same time, we realize that one might consider our API too primitive for complex application design.

There are several directions in which the AppCard model can be developed. Further work in both hardware and software needs to be done in the area of migration to new hosts. And it is only when the host server is ported to a new platform that we will determine whether our “host-independent” APIs can in fact be easily mapped to a new host. Also, the issue of how to handle hosts that write bytes in different orders (i.e. Little or Big-Endian) needs to be addressed.

In the existing model, a collection of AppCards are connected to a host and pass all functionality that they don't have to the host. One could consider a more symmetrical composition where a set of AppCards are connected as a micronetwork on a peer-to-peer basis. Even the user interface part which can include a graphic processor and terminal is connected to the micronetwork using the same protocol as all other AppCards. Using this approach, general-purpose desktop and portable computers may be built as a collection of AppCards connected by a micronetwork. We could easily reconfigure and upgrade such collections.

Most existing operating system are moving towards decomposition of the software into distributed objects. Several implementations of the CORBA [13] protocol like distributed SOM [5] are already incorporated into commercial operating systems and existing software is evolving to use these technologies. One way to implement the AppCard software would be to assign such distributed objects instead of applications to the AppCards. CORBA provides the ability for remote communication among several computers. Augmented with ability for migration and disconnected operation, this technology could be well suited for AppCard software interface.

We hope that the AppCard approach to build computer systems can change how computer systems look like today and bring mobility and flexibility to the areas that currently lack them.

## 8 References

- [1] R. Burgess. *MMURL: Your Own 32-Bit Operating System*. Dr. Dobbs's Journal, v.19, no. 5, May 1994.
- [2] A. Davison et al. *Distributed Window Systems*. Addison-Wesley Publishing Company. 1992.
- [3] L.B. Huston, P. Honeyman. *Disconnected Operation for AFS*. Proceedings of the First Usenix Symposium on Mobile & Location-Independent Computing, pp. 1-10, August 1994.
- [4] IBM Corporation. *Technical Reference: Personal Computer AT*. 1984.
- [5] IBM Corporation. *IBM SOMobjects Developer Toolkit User Guide*, Version 2.0
- [6] Intel Corporation. *Embedded Microprocessors*. 1994.
- [7] J. Ioannidis, D. Duchamp, G. Maguire Jr. *IP-Based Protocol for Mobile Internetworking*. Proceedings of Sigcom 91.
- [8] A.D. Joseph et al. *Rover: A Toolkit for Mobile Information Access*. Proceedings of Fifteenth Symposium on Operating System Principles, December 1995
- [9] J.J. Kistler and M. Satyanarayanan. *Disconnected Operation in the Coda Filesystem*. ACM Transactions on Computer Systems, 10(1), February 1992.
- [10] Chris aeda and Brian N. Bershad. *Protocol Service Decomposition for High-Performance Networking*. Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles. December 1993.
- [11] J. McCrindle. *Smart Cards*. IFS Publishing, UK, 1990.
- [12] *MS Windows Programming Guide*. Microsoft Press. 1989.
- [13] *Object Management Architecture Guide, Revision 2.0*. Second Edition, 1992, OMG TC Document 92.11.1.
- [14] Personal Computer Memory Card International Association. *PC Card Standard, Release 2.0*. September 1991.
- [15] *Presentation Manager Programming Guide*. OS/2 Technical Library. IBM, 1993.
- [16] J. Saldana and D.L. Cohn. *A Hybrid Model for Mobile File System*. Workshop on Mobile Computing Systems and Applications, Santa Cruz, December 1994.
- [17] E. Solomita, J. Kempf, and D. Duchamp. *Xmove: A Pseudoserver for X Window Movement*. The X Resource, Number 11, pp. 143-170. July 1994.
- [18] T. Watson. *Application Design for Wireless Computing*. Proceedings of the Workshop on Mobile Computing Systems and Applications. December 1994.