Magi: A System Software Model for Intelligent Devices*

H. Richard Kendall and Vincent W. Freeh  
{rkendall,vin}@cse.nd.edu  
Department of Computer Science and Engineering  
University of Notre Dame  
Notre Dame, IN 46556

July 19, 1999

Abstract

Architectural advances have pushed general-purpose processors into devices. These “intelligent” devices provide great opportunity for mitigating the I/O bottleneck. This paper presents principles that an intelligent device programming model should adhere to in order to achieve the full potential of these emerging devices. It also presents a prototype implementation of an intelligent device programming model.

1 Introduction

Computers no longer have the ability to keep processors running at peak speeds due to lagging architectural advancements in I/O devices. One method of combating the growing performance gap is to embed “intelligence” (processing capability) in peripheral devices in order to more efficiently utilize I/O. However, in order to fully realize the performance opportunities presented by intelligent devices, it is necessary for system software and devices to collaborate on tasks. Unfortunately, intelligent device functionality in current systems is ad hoc at best and is unnecessarily limited. Many intelligent devices today use proprietary, closed interfaces that do not allow an application to export operations to a device. In addition, system software interfaces to devices are often legacy code from the days in which devices were “dumb” and do not allow a host system to collaborate with its devices.

This paper presents Magi, a simple, efficient software model that mitigates the I/O bottleneck by leveraging the capabilities of embedded systems on intelligent devices. In Magi, applications apply transformations, modular stream-oriented operations to data. A transformation may either respond to an external trigger or initiate actions. In addition, a

*This research was supported by the JPL HTMT project, Arthur J. Schmitt Fellowship, ND Faculty Research Program, and NSF Career Grant CCR9876073.
transformation operation may execute either on the host or on the device, as appropriate. Furthermore, a transformation may dynamically migrate between the host or device based on runtime information, such as system load.

The Magi programming model places the control for data operations onto the device that is best suited for managing the data. This contrasts traditional systems in which the control typically resides on the processor of the host. However, a device often has more information than the host system and is in a better position to decide how to manage data. As a result, policies in a conventional system, such as prefetching, are often underinformed and do not fully exploit their potential.

There are many opportunities for a software model that has full access to intelligent devices, such as Magi. First, latency to a device can be reduced by actively pushing data lower in the memory hierarchy and reorganizing the structure of data during device idle periods. Next, device bandwidth can be increased through data compression [18] and active requests that process data locally on the device. For example, consider a database search for entries matching two selected fields. On a traditional system, the entire database is transferred to the host processor where matches are found and the majority of the information received is discarded. In contrast, a database on an intelligent disk drive can search for the selected fields locally and transfer only the results back to the host. Finally, intelligent devices can be treated as a coprocessor in order to exploit pipeline parallelism and offload work from the host.

This paper characterizes the features that should be found in an intelligent device programming model and explains why they are important. Additionally, it pinpoints several types of opportunities in intelligent devices. The paper also alerts device designers to the advantages of open interfaces on intelligent devices. The organization of this paper is as follows. Section 2 characterizes the features of an intelligent device software model. Section 3 presents our testbed and implementation. Section 4 describes related work. Section 5 draws conclusions and outlines the direction of future work.

2 Intelligent Device Model

Intelligent devices have several advantages over conventional devices. This section explains how these advantages can lead to a better programming model than is currently in use. Following this discussion, an analysis of the characteristics of an intelligent device programming model is presented.

The primary concern with intelligent devices today is not whether to use them, but rather how to use them. One can treat intelligent devices exactly as before. The advantage of this is that existing applications work with intelligent devices. Unfortunately, it restricts the services that can be provided. In particular, all intelligence is implicit—performed transparently by the device. On the other hand, device processors can be peers to the host processor(s). In which case, an intelligent device shares the workload in any imaginable way. Unfortunately, all applications must be re-written to support the new interface. Moreover, each intelligent device has capabilities very different from the host and each other. We argue for a programming model that sits in the middle: support existing interfaces, but allow for customization. Furthermore, without a general intelligent device programming model,
the future will continue to see ad hoc divisions of labor propagating nonuniform, inefficient resource use.

Our research group has focused on extensible systems over the last several years. Our experiences have highlighted many characteristics that we feel are important in extensible systems in general and intelligent devices in particular. The following wish list outlines five principles for an intelligent device programming model that will yield substantial increases in performance and functionality. These include:

- Downloading operations to a device,
- Enabling operation adaption based on runtime information,
- Permitting passive and active operations,
- Providing persistent and transient bindings, and
- Supporting application-specific operations.

**Download operations to a device.** First, the model must enable downloading operations to an intelligent device. Often devices are not extensible and support only built-in functions. However, it is not possible to foresee in advance all possible uses for a device. Therefore, a general model must allow operations to be pushed into the device. The result is a system that is more able to share a workload between a host and its devices. Moreover, by offloading an operation onto an intelligent device, where it can most efficiently execute, the cache of the host system is not polluted with unnecessary data. Tests have shown that the net cost of context switching can be thousands of machine cycles due to residual values in the cache [12].

**Enable operation adaption.** Next, adaptive operations are essential for maximizing performance. Exposing runtime information to an operation allows it dynamically to fine-tune the system by choosing among several possible modes of execution. One use of adaptation involves migrating an operation back and forth between the host system and the device based on system load. For example, consider reading a compressed file on an intelligent disk. If the load on the disk is low and bandwidth is not a bottleneck, it is most likely beneficial to perform the decompression locally on the disk. This reduces computation on the host system and does not disturb the host’s cache with unnecessary data. In contrast, if bandwidth to the intelligent disk is a bottleneck, the disk may decide to pass the compressed data to the host system which then executes the decompress operation.

**Permit passive and active operations.** In addition, it is necessary to support passive and active data operations. A passive operation responds to an external trigger, such as a request for data. For example, consider an operation that calculates a checksum whenever data is passed through it. The operation is passive because it only executes when another entity gives it data. Passive operations by themselves are quite useful; in fact, our previous system exclusively uses passive operations that trigger on data accesses [8, 10]. However, intelligent peripherals need to initiate actions. For example, an active operation on an intelligent device might reorganize its data into a more efficient structure during idle periods.
Furthermore, it could predict future requests and actively push data into the appropriate level of a memory hierarchy.

**Provide persistent and transient bindings.** Fourth, it is beneficial to support both *persistent* and *transient* operations. A persistent operation is permanently associated with a set of data. Once the operation is bound to its data, it is always associated with the data until it is explicitly unbound. For example, an encrypted file system persistently binds the encrypt and decrypt operations with its files and does not require explicitly specifying the operations on each access. In contrast, a transient operation is temporarily bound to its data either because the data or the effects of the operations are temporary. Although they are functionally equivalent, having access to both types of bindings makes a system much more usable.

**Support application-specific operations.** Finally, the model should support application-specific operations. The first principle has already established that the ability to download operations to a device is a necessity. However, downloading application-specific operations allows even more flexibility. Several systems have shown that application-specific customization leads to significant performance improvements [10, 5, 1]. Unfortunately, due to security concerns, many extensible systems only let privileged users implement extensions [19, 7, 11]. This limits the advantages of intelligent devices by preventing ordinary, unprivileged applications from customizing the system. Furthermore, numerous well-documented methods of implementing safe extensions have been developed [9, 3].

A model satisfying the five principles listed above is both necessary and sufficient. The desirability of each characteristic has already been shown. Furthermore, each principle has a direct correspondence to capabilities or practices in conventional systems. The novelty lies in applying these principles to the new field of intelligent devices. Consequently, the proposed model is sufficient because it is functionally equivalent to the conventional programming model that is in use today.

## 3 Magi System

The *Magi* system (named after a group of cooperating wise men) is a software model for cooperating intelligent devices. The Magi prototype consists of intelligent device hardware and a software interface for accessing it. The implementation is capable of supporting numerous intelligent devices; however, at present, our primary focus is an intelligent disk drive.

### 3.1 Magi Model

The Magi programming model is heavily based on our previous extensible system, the Modify-on-Access (Mona) file system [8, 10]. Mona is an extensible file system that supports pushing *transformations*, application-specific streaming operations, onto the input and output streams of files.

In Mona, any user, regardless of privilege, may write and apply a transformation to a file. However, in order to maintain system security, the location where a transformation
executes is determined by privilege. Only superuser processes are permitted to download a transformation into the operating system kernel. Mona exports any unprivileged transformation to a user-space daemon which runs the operation under the permissions of the user who instantiated the transformation. Likewise, we envision two modes of execution on an intelligent device. Other techniques for secure extensions also exist [3, 9, 13] but are complicated by the need for typesafe languages or static recompilations.

Magi, like Mona, associates transformations with files through transient and persistent bindings. A transient binding applies a transformation to an open file through a call to the /ioctl system call. When the affected file closes, the file system discards the transformation bindings associated with the file. In contrast, a transformation in a persistent binding is resident in the file system until it is explicitly removed. A persistent binding is accomplished through a transformation link, a file type derived from symbolic links. Like a conventional symbolic link, accessing a transformation link redirects a file access to a base file. However, a transformation link also contains meta-information specifying which transformations to associate with the base file.

Mona has demonstrated the utility of application-specific operations, even when an intelligent device is not used. By dynamically customizing data accesses according to an application's needs, speedups of up to 5 times have been observed [10]. We expect that the figures for Magi will show even better results due to the additional advantages of intelligent devices detailed in Section 2.

3.2 Magi Prototype

Magi is capable of executing operations on either the host system or an attached intelligent device. Like Mona, Magi applies transformations to data streaming to and from devices. The primary difference between the two systems is that Magi also has the capability to export a transformation to an intelligent device.

In the prototype, a host transformation stack and a device transformation stack are attached to each data stream between the host system and an intelligent device. A simple protocol supports the insertion and removal of transformations for both host and device transformation stacks. Consequently, Magi enables an operation to dynamically migrate between the host and device based on runtime information.\(^1\)

The Magi system emulates numerous intelligent devices with a general purpose coprocessor card. This device is a complete computer embedded on a card that is attached to the host system’s motherboard. The device contains a processor core, main memory, disk and floppy drive controllers, ethernet and video adapters, and both parallel and serial ports. Consequently, the system emulates numerous intelligent devices, such as disk drives or network adapters. For example, if the host mounts a file system on /dev/intelligent_disk, the intelligent disk device driver communicates with the intelligent device to logically mount the file system on the device’s disk drive. Applications then may apply transformations to data streams corresponding to files on the remote disk. The transformation operation may execute either on the host side of the stream or on the device side.

\(^1\)Migrating a transformation between a device and the host system is currently greatly simplified due to a shared Intel x86 instruction set. The final implementation will most likely use a device independent transformation format, such as java, with interpreters on both the host and device.
In summary, Magi supports all of the principles outlined in Section 2 and does so primarily through the existing interface to devices (open, read, write, ioctl, etc.). As a result, the Magi system has little cost of adoption while providing many benefits of extensibility. Our current work focuses primarily on the benefits of intelligent disk drives, yet also has great potential for emulating other devices, such as intelligent network adapters [5, 15].

4 Related Work

The concept of embedding “intelligence” into devices it not new and many familiar devices are already intelligent. Unfortunately, a general programming model to leverage intelligent devices is still lacking because each new device typically introduces a new specialized interface. As a result, several related systems also are addressing this issue.

The Active Disk architecture simulates a disk drive containing an embedded x86 processor and memory [1]. A limited form of transformation, a disklet, operates on streams of data entering and exiting the disk drive. For security reasons, disklets are highly restricted and do not support active operations or dynamic memory allocation.

The ISTORE project is designing a fault-tolerant and scalable system based on a set of intelligent disk drives [4]. Each disk incorporates a runtime capable of monitoring its environment in order to adapt to a workload. The software model allows users to define application-specific views which select data and triggers which invoke functions when specified conditions are satisfied.

The Active Pages [16] computational model examines the effects of placing programmable logic within a memory chip. The memory system of a simulated processor architecture was modified to support uploading application-specific code to memory and associating the code with a page (512 KB) of memory. Accessing an Active Page operates on data locally on the memory chip. The Active Pages system demonstrates application speedups of up to three orders of magnitude.

Alteon Networks Inc. produces a gigabit network interface card, ACEnic, that is one of the few truly configurable intelligent devices available today [15]. Alteon has released firmware source code for its card, enabling developers to download operations to the device firmware. The SPINE project focuses on downloading application-specific operations to the ACEnic intelligent network adapter [5]. In the SPINE model, a programmer examines an application to determine which operations are more suitable for an intelligent network adapter than the host system. Once candidate operations are identified, they are converted into extensions written in a typesafe language and downloaded onto the ACEnic adapter. The SPINE implementation has demonstrated excellent performance for applications that do not require computation on a host processor, such as a video client and an Internet Protocol router.

A Network Attached Storage (NAS) device, also called a storage appliance, is a new storage architecture integrating intelligent disks and network adapters[6, 2, 14]. A NAS device connects directly to a network and provides file server capabilities for several network protocols, such as NFS, FTP, and HTTP. In addition, NAS also has the ability to appear like an ordinary disk drive to clients. As a result, NAS simplifies data management while increasing I/O scalability.
The Intelligent I/O (I₂O) model incorporates specialized I/O processors with devices [17]. The goals of this architecture are to minimize performance penalties due to interrupts and to implement a device driver model that shares components across platforms.

5 Conclusion

For many years, people correctly viewed their a computer system as a processor connected to passive peripherals. However, peripheral devices now have processing capabilities, giving them “intelligence”. Several factors motivate the shift toward intelligent devices. These include preprocessing data to mitigate the I/O bottleneck, hard-coding functions onto a device for fast, easily supported applications, and supporting application-specific functionality. However, most of the justifications for intelligent devices can be summarized by a single principle: reduce cost.

This paper discusses the five principles required by an intelligent device programming model. Furthermore, the paper demonstrates that Magi, a programming model fitting these criteria, can reduce a system’s effective latency, increase its effective bandwidth, and exploit parallelism through pipelining. Magi allows intelligent devices to improve existing applications implicitly and permits future applications to leverage the potential of intelligent devices. We hope this discussion serves as a starting point for developing a uniform programming model for intelligent devices.

References


