An Application Framework for Efficient, Reliable and Secure Access to Memory Spot

Jun Li, Riddhiman Ghosh, Elsa Durante
Digital Printing and Imaging Laboratory
HP Laboratories Palo Alto
HPL-2006-83(R.1)
April 18, 2007*

efficient data synchronization, automatic data mapping, data access control, event-based application, near-field communication, flash-memory file system

Memory Spot is a microchip developed by HP Labs at Bristol. Compared to Radio Frequency Identification (RFID) devices, Smart Cards and other Near-Field Communication (NFC) devices, this chip provides much larger storage capacity, faster data transfer rate, and smaller physical dimensions. The chip is normally passive and is energized by an external reader/writer over the Radio Frequency (RF) channel. The same RF channel enables data communication between the reader/writer and the powered chip. Memory spot potentially opens up a wide range of new consumer-oriented and enterprise service-oriented applications. This paper presents an application framework to help develop these memory spot related applications, by providing common functionalities across different applications, such as data synchronization, application management and application-device interaction. In general, these applications are involved with arbitrarily complex application data types, including database and data schema. Further, in a disconnected and cross-organization environment, these applications require efficient, reliable and secure access and update of data to memory spot. Our application framework addresses these requirements, different from other frameworks that focus on RFID, Smart Cards and other NFC devices. To demonstrate the innovative applications enabled by memory spot and the application framework, we have prototyped the warranty spot application, which aims to significantly reduce computer-related warranty fraud, by having a memory spot permanently attached to each computer.

* Internal Accession Date Only
Approved for External Publication
© Copyright 2007 Hewlett-Packard Development Company, L.P.

An Application Framework for Efficient, Reliable and Secure Access to Memory Spot

Jun Li, Riddhiman Ghosh, and Elsa Durante
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304
{jun.li, riddhiman.ghosh, elsa.durante}@hp.com

ABSTRACT

Memory Spot is a microchip developed by HP Labs at Bristol. Compared to Radio Frequency Identification (RFID) devices, Smart Cards and other Near-Field Communication (NFC) devices, this chip provides much larger storage capacity, faster data transfer rate, and smaller physical dimensions. The chip is normally passive and is energized by an external reader/writer over the Radio Frequency (RF) channel. The same RF channel enables data communication between the reader/writer and the powered chip. Memory spot potentially opens up a wide range of new consumer-oriented and enterprise service-oriented applications. This paper presents an application framework to help develop these memory spot related applications, by providing common functionalities across different applications, such as data synchronization, application management and application-device interaction. In general, these applications are involved with arbitrarily complex application data types, including database and data schema. Further, in a disconnected and cross-organization environment, these applications require efficient, reliable and secure access and update of data to memory spot. Our application framework addresses these requirements, different from other frameworks that focus on RFID, Smart Cards and other NFC devices. To demonstrate the innovative applications enabled by memory spot and the application framework, we have prototyped the warranty spot application, which aims to significantly reduce computer-related warranty fraud, by having a memory spot permanently attached to each computer.

1. Introduction

Memory spot is a microchip invented at HPL Bristol [25, 18, 2, 8]. The physical size of the chip is only about 2 mm^2. Through the Radio Frequency (RF) power coupling, the on-chip customized processor responds to the reader/writer over the RF channel on its embedded flash memory access. From a technology perspective, memory spot can be viewed as a Near Field Communication (NFC) device [17]. However, compared to Radio Frequency Identification (RFID) and other NFC devices, memory spot not only occupies a much smaller physical dimension, but also provides a much larger storage capacity (up to 512KB) and a faster data transfer rate (~10Mb/s). Due to the limited processing power on the chip, applications have to be deployed on a host device, which can be a PC or a mobile device like a Personal Digital Assistant (PDA) or cellular phone. The reader/writer is attached to the host as an accessory device. Typically the application reads the data to the host cache through the reader/writer by positioning the reader/writer close to the spot (~ 1 mm away), manipulates the cached data at the host, and finally updates the data back to the spot. Because of its sufficient storage, applications can access the data on memory spot locally, anytime and anywhere, in contrast to other reference-only systems like RFID that have to rely on an online infrastructure for actual data access.

Memory spot can potentially become a significant enabler for a large collection of applications, including voice-annotated documents, electronic passport, medical history tracking, computer warranty tracking, etc. These applications span across both consumer and enterprise service domains.

We envisage memory spot as an integrated part of a much larger information processing system, which involves both the people that access the spot and the online infrastructure that the application occasionally

interacts with. In enterprise service-oriented applications, such as to keep track of a computer's configuration, service and warranty entitlement over its lifetime, or to record medical history information [4], memory spot allows data access such that different people from different organizations can read and update the data. Consequently, memory spot becomes a cross-organization data fusion point. Further, data access cannot rely on an always-on infrastructure, as otherwise the key value of self-contained local storage would be largely diminished.

For memory spot and its applications to be operated in a disconnected and cross-organization environment, our goal is to figure out how to provide a flexible programming environment to rapidly develop memory spot applications that can efficiently, reliably, and securely access and update the data stored in a memory spot. In particular, we have identified the following major technical challenges on data management:

- Flexible application data types: Data type definitions differ among applications, e.g. only an integer counter is required for a spot-enabled public transit pass, but complete computer service history requires an in-memory database (with different database tables) for efficient data inspection via SQL. A unified data modeling scheme is required to flexibly express arbitrarily complex data types;
- Reliable and efficient data access and update: The communication channel between the reader/writer and the chip is inherently unreliable, since the distance to the chip is often not well-controlled by the user. Sufficient feedback on the status of data access and update is required in order to assist the user to recover the communication channel promptly. Furthermore, flash-memory's write access is relatively slow due to memory cells' re-programming, which demands an efficient data update scheme different from a traditional file system;
- Scalable storage: For the applications involved with multimedia data, such as document scanning and voice annotation, the contents could exceed a single memory spot's storage capacity;
- Data Security: data integrity, data confidentiality and data access control are crucial to high-value application data. However, one should not expect support from an always-on infrastructure to help ensure data security at the time of data access.

Our research objective is twofold. The first one is to develop a software infrastructure to provide common functionalities, including data management, to enable rapid development of various memory spot applications in different application domains. The second one is to investigate and develop compelling applications that can bring new service and revenue opportunities to the IT sector including HP. We believe that the technical challenges identified above are also common to the family of NFC microchip storage devices that provide on-chip storage access through power coupling over the wireless communication channel.

The rest of this paper is structured as follows. Section 2 describes two application examples, voice spot on voice annotation and warranty spot on computer service and warranty history tracking. Voice spot will be used in other sections for example illustration and warranty spot will be detailed in Section 8. Warranty spot is the most comprehensive application prototype that we have developed so far. Section 3 provides the overview of the application framework, which is located between the hardware and the end-user applications. Section 4 details various techniques in data management, including automatic data mapping, reliable and efficient data caching and updating. Section 5 describes major memory spot application features and management of these applications. Section 6 details data security features offered in the application framework with the on-chip hardware support, in particular, digital signature and data access control. Section 7 uses voice spot to demonstrate how a memory spot application can be developed. Section 8 focuses on warranty spot application and the prototype. Section 9 contrasts this work to related work in RFID, Smart Cards and other NFC devices. Finally, Section 10 offers some conclusions and future directions.

2. Application Examples

Two application examples, voice spot and warranty spot, are provided to explain how memory spot can be attached to host devices or physical media and become an information carrier to convey additional information in a unique way, because of its sufficient storage capacity and miniature size. In particular, a voice spot enables digital multi-media insertion onto a piece of physical paper, and a warranty spot facilitates lifetime tracking of the warranty/service information for a computer.

2.1 Voice Spot

In this example, a project proposal is printed on paper and a memory spot is attached to the front page of this document. A hand-held device (such as a PDA or a cellular phone) equipped with voice recording capability hosts the voice spot application. The owner of the document makes an initial voice record to the spot to state that the spot is associated with this particular document. The physical document (along with the spot) is passed on to the proposal reviewers. Each reviewer reviews the paper document, and voice annotates the spot with his or her comments. Each review from a reviewer can contain multiple voice annotation sections, and the reviewer can specify which pages are under which annotation section.

The document is circled between the reviewers. At the end, the document is returned back to the document owner. The document owner uses the handheld host device to retrieve all the voice comments from the memory spot. The owner can listen through all the comments, sorted by the reviewers or by page numbers. By incorporating the comments recorded on the memory spot, the owner can create a new version of the document.

If necessary, this new version of the document is printed out. A new memory spot is attached to this new paper document. A new round of proposal review may be started over again.

2.2 Warranty Spot

This application and the related solution architecture will be described in more detail in Section 8. We provide only a brief explanation here.

A memory spot is permanently attached to a computer and over its lifetime, machine configuration, service records and part warranty entitlements are all stored to the memory spot, by technicians or end-user customers or other authorized personals. Such computer information is ready for retrieval right next to the computer, without resorting to the centralized database, which does not exist today. Currently, such computer-related information is only available at the disjoint databases scattered across different organizations. To acquire comprehensive information about a computer under warranty is time consuming, or very often impossible as cross-organizational access is unavailable due to the issues in access authorization or internet connectivity, among others. As a result, the memory spot becomes a data fusion point of data provided from different organizations, regarding the computer that the spot is attached to.

The spot is called warranty spot, because it is originally designed to reduce warranty fraud for the computer under service contract. It can also be leveraged to improve service efficiency, as all the historical information on the computer is available, right next to the computer, independent of whether the computer is still functional or not.

The initial memory spot is created at the end of the manufacturing line, when all the major hardware components are configured. Once the computer is installed to the customer site and becomes operational, every time a technician from a different service provider is dispatched to provide the service, the technician can locally determine the most recent system configuration and whether the failure part is under warranty, by retrieving the information stored on the warranty spot, without relying on the online databases that might not have access rights granted to the external service providers, or at the time of the service, either an online connection is not available or the online service is temporally down. After the technician finishes the computer service, the service record is updated to the memory spot, along with the information about the hardware add-on parts and the warranty associated with the newly introduced parts. All data input will bear the provider's digital signature to prevent data tampering and facilitate future data traceability.

Once the technician is back to the online environment (such as the office), the fully cached information from the memory spot can be used to provide data back up, in case the memory spot attached to the computer is maliciously destroyed or accidentally corrupted. The same information can be used to automatically fill a filed report. Furthermore, once this cached information is sent to the warranty administration department, the claimed part's warranty entitlement can be verified, in contrast to the current practice that only relies on the part serial number to determine the warranty entitlement.

3. Overview

We have developed a multi-layered application framework as shown in Figure 1. At the bottom, the framework relies on the hardware abstraction called the Reader/Writer APIs, which expose the memory spot as a file system to the application framework. The reader/writer APIs cover the file-level read or write access to the memory spot from the reader/writer via the RF channel. With this hardware abstraction, the

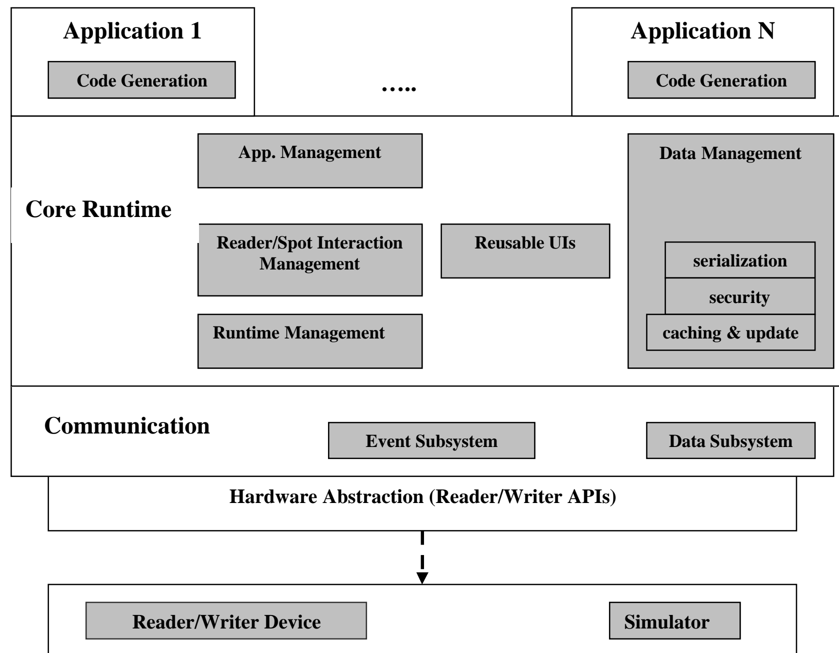

Figure 1: Memory Spot Application Framework Overview

application framework and thus the application become unaware of the physical realization of the reader/writer (and the memory spot as well). As a result, the actual hardware can be substituted by the pure software simulator counterpart, as long as the simulator follows exactly the same reader/writer APIs. In practice, the simulator allows application developers to develop memory spot applications, without the real memory spot hardware devices in place. Only at the late software-hardware integration stage, will the real hardware devices be required.

The communication layer is comprised of two subsystems: event subsystem and data subsystem. The data subsystem creates a wrapper to synchronize file-level data access exposed from the reader/writer APIs. The event subsystem allocates a monitor thread to actively monitor the RF communication channel to detect whether a memory spot is available and in-range for reading or writing. The event subsystem also intercepts the calls to the reader/writer APIs, discovers and filters various abnormal hardware access events derived from the return of these calls. The events on either data availability or hardware abnormal access are then translated into high-level events, which can be propagated in the application framework and eventually reach the application.

The core runtime layer incorporates a collection of software modules. In particular, the *Runtime Management* module is responsible for the start and shutdown of the application framework and the occupied resources (e.g., threads, communication channel). The *Application Management* module is responsible for locating a collection of applications installed on the host, determining whether an application has sufficient capability required by the target memory spot, and starting an appropriate application. The *Reader/Spot Interaction Management* module is to keep track of the memory spots that are being cached, to subscribe to the events raised from the communication layer and to promptly translate the captured events into visual and sound cues to the user. The *Data Management* module is to handle file system caching, data object serialization, reliable and efficient data caching and updating, and data security that primarily deals with data integrity and data access right control. The *Reusable UIs* packages different UI components that can be incorporated into different applications' UI front-ends, to display the status of data processing, or to render the events raised by the framework into text or sound, in order to actively engage the user into this human-in-the-loop system.

At the top of the framework is a collection of memory spot applications, which are event driven. A generic memory spot application is provided with concrete implementation, to handle data retrieval and

data updating, to respond to the events raised from the lower application framework layers, etc. To further simplify user-level programming, hierarchical data objects' access and their marshaling/unmarshaling are provided automatically via code generation, based on the user-defined data model.

4. Data Management

A memory spot application starts with data retrieval from memory spot, and then performs data inspection on the cached data. If the data is updated by the application, the data needs to be synchronized back to the spot, before the application is terminated. In this data life cycle, data management is responsible for data object serialization between the memory spot and the application in a flexible way (described in Section 4.1), file system caching and synchronization (described in Section 4.2), and data updating in a reliable and efficient manner (described in Section 4.3). Data security, certainly an important aspect in data management, is separately addressed in Section 5.

4.1 Automatic Data Mapping

Application data is stored as a linear sequence of bytes in memory spot. For each memory spot, the reader/writer APIs expose application data as the files in a file system. In general, the following application data types have been identified:

- **Data Structures**, ranging from a simple integer (such as the counter in the public transit pass), to hierarchical data structures (such as the data in voice spot and warranty spot).
- **Files**, including document files in *.pdf and *.doc, media files in *.wav and *.JPEG, etc. If the application files have application-specific meta-data to encode further information on these files, e.g., the voice annotation's starting time and end time, such meta-data can be expressed in a hierarchical data structure, with one particular data element in the data hierarchy being designated to hold the fully scoped file name of the file to establish the linking between the file and its meta-data.
- **Database**, in particular, the in-memory database hosted in the address space of the application. Database and SQL provide a platform that greatly simplifies data inspection and data manipulation.

To unify data modeling of all these different application data types, we exploited the Interface Definition Language (IDL), the specification language commonly adopted for the middleware platforms like CORBA or COM. In general, the IDL supports arbitrary and complex hierarchical data type specification. An example of the IDL specification for voice spot is shown in Figure 2, in which all the data elements are structured hierarchically to form a data tree, with the root of the tree being with the type of `VoiceSpotData`.

From the user-defined IDL data specification, the IDL compiler automatically produces code for hierarchical data tree's access and update, and serialization/de-serialization. In addition to the root node, the serialization/de-serialization can be performed at a particular sub-tree as well. The data serialization engine commonly exists in a middleware infrastructure. The particular one that we have incorporated into our application framework is the IIOP serialization engine in a CORBA-like infrastructure called ORBLite [21]. The IIOP serialization output is a binary stream.

With this hierarchical data mapping that simplifies the application's data access, the data flow between the application and the memory spot regarding data access and update is shown in Figure 3. To the application, data access can be at the file level or at the finer data record level.

Furthermore, mapping between hierarchical data structure and database is supported. To database related applications, data modeling starts with IDL. For a table, each data row is expressed as a data structure in IDL, called the *row data type*. The entire database table is expressed as a sequence of the data elements sharing the same row data type. In a data structure, for the SQL data types like `SQL_INT`, `SQL_FLOAT` and `SQL_STRING`, the IDL already provides the counterparts like `long`, `float` and `string`. For other SQL data types such as `SQL_DATE`, we have to define a row data structure called `Date` in IDL to express the corresponding fields of `SQL_DATE`.

Once the database schema is modeled in IDL, after the application retrieves the memory spot data (a binary file), the data is first constructed as a hierarchical data tree. A database table is then constructed and populated on-the-fly in the address space of the application. The data schema for the table can be automatically constructed by extracting the type definition of the row data structure via reflection. That is, the name of the column matches the filed name in the row data structure, and the data type of the column

```
module VoiceSpot {
 struct VoiceRecord {
       long PageNumber; /*the page number associated with the document*/
       long StartTime; /*the start time of the voice annotation*/
       long EndTime;  /*the end time of the voice annotation*/
       string StoredFileName; /*where the actual voice media is recorded*/
 };
 /*each person can have multiple voice records*/
 typedef sequence<VoiceRecord> VoiceRecords;

 struct PersonRecord {
   VoiceRecords Records; /*all the voice records from the same person*/
   string Owner;   /*the name of the person that contributes the voice records*/
 };
 /* multiple reviewers to annotate the document*/
 typedef sequence <PersonRecord> PersonRecords;

 /*all these reviews are associated with a particular version of the document*/
 struct DocumentVersionAnnotation {
       string DocumentVersion; /*the version of the document being annotated*/
       string DocumentName;  /*the name of the document*/
       PersonRecords VoiceRecords; /*all the voice records from multiple people*/
 };
 /*the root node of the single data tree*/
 typedef DocumentVersionAnnotation VoiceSpotData;
};
```

Figure 2: IDL Application Data Specification

matches (or converted, in the case of SQL_DATE and the like, as the matching is implicitly defined) the type of the corresponding data field. Right before the application is closed, if the database is changed, the entire database is de-populated into a new instance of data tree, which is further de-serialized into a binary file and eventually this binary file is updated to memory spot, to reflect the changes conducted by the application.

The technique of transforming hierarchical data structure from/to binary data sequence described above, is called *automatic data mapping*. It provides a universal data modeling mechanism and the corresponding runtime support to greatly simplify application development, when application data access and update needs to be performed at the record level.

4.2 File System Reliable Caching and Updating

With automatic data mapping, the application still requires a reliable file system to support reading, writing and updating of the file (or files) to this file system. This reliability requirement stems from the

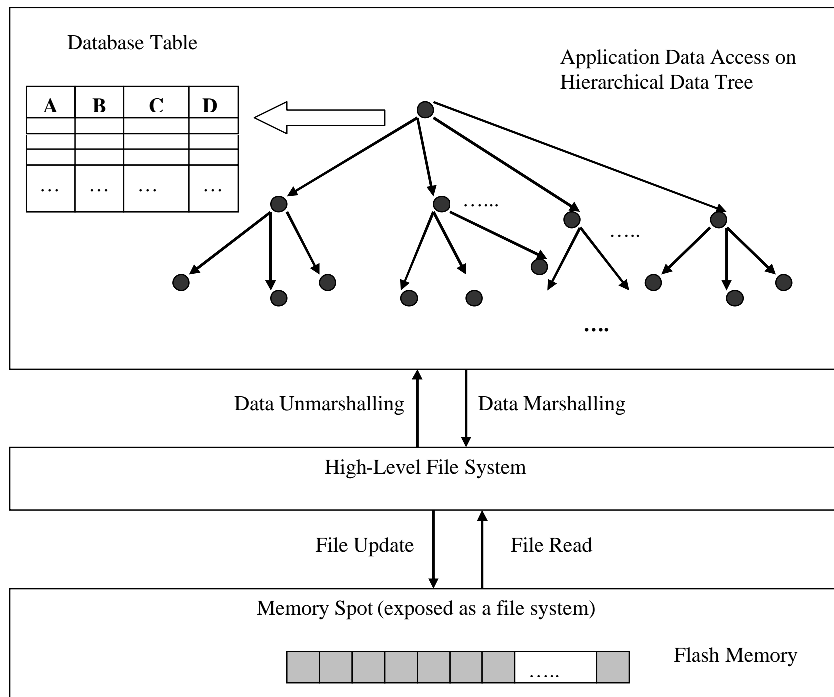

Figure 3: Automatic Data Mapping for Hierarchical Data Access and Update

physical characteristics of the underlying hardware and the communication link between the reader/writer and the memory spot. The communication channel facilitates not only data transfer but also power coupling from the reader/writer. The power coupled to the spot drives a highly customized processor to read/write the spot's flash memory [18]. The distance between the reader/writer and the spot has to be within the range of 1~2 mm, and these two devices have to be well aligned. In a typical operational environment, during a complete read/write operation issued by the application, it is difficult to guarantee that the user can steadily hold the reader/writer to the memory spot without interrupting the communication in the middle, especially when writing a large data file. To the current memory spot with 32KB storage capacity, writing 32KB currently takes about 4 seconds.

Even more troublesome, if the application has to deal with reading/updating of files multiple times, each time a file operation is issued, before the operation can proceed, the user has to be notified by the application to approach the reader/writer close to the spot to establish the communication link.

To spare the application the burden of actively being responsible of reliable data communication, a file system cache is provided by the application framework. Each memory spot has a unique file system cache. With this cache, the user-to-spot interaction follows the *two-sensing* model. At each sensing, the user has to approach the reader/writer to the spot within a sufficiently close range with a good alignment. The first sensing happens when the application has the spot data to be fully cached into a local file system at the host computer. The second sensing happens when the application decides to synchronize the local cache with the spot, if the cache has been modified by the application. Once the cache is synchronized with the spot, the local cached file system is deleted. In between these two sensing instances, the application only interacts with the cached file system on the host, which is a reliable file system, as it is located either in memory or on a hard drive.

To each memory spot, the duration between the commencement of the spot data caching from the spot, and the finishing of the data synchronization back to the spot, is defined as a *data communication session*. We also call this local cached file system that we just introduced the *high-level file system* (HLFS). Correspondingly, the flash-memory oriented file system that resides below the reader/writer APIs is called the *native flash file system* (NFFS).

A state machine is created for each memory spot's caching. The data caching state machine is shown in

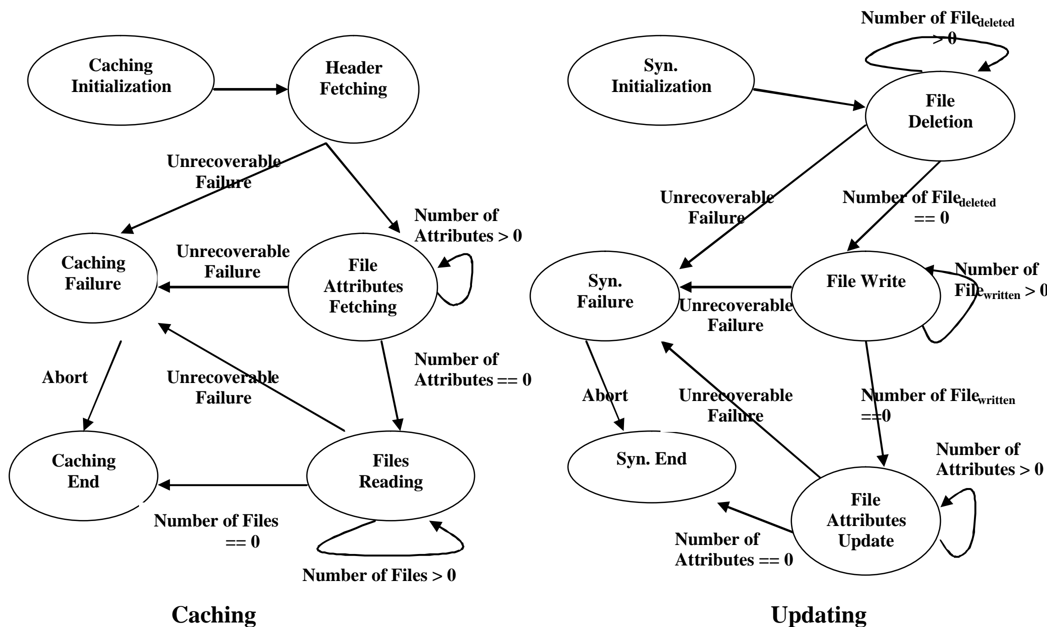

Figure 4: State Transition Diagram for Reliable File System Caching and Updating

Figure 4. Each state is associated with an action that encapsulates the invocations to the reader/writer APIs. The state machine ensures that caching is always continued from the last healthy state, should a transient failure occur. Transient failures are due to the communication link that can be temporarily broken. In Figure 4, an unrecoverable failure can happen, when an inherent failure happens to the NFFS located on the memory spot, such as being not formatted or being corrupted. Such unrecoverable failure can be detected by examining the runtime exception thrown from the call to the reader/writer APIs. The unrecoverable failure can also happen, when the total number of the exceptions (due to the transient failures) captured exceeds the predefined limit. Once an unrecoverable failure incurs, the file system caching is aborted and the incomplete file system instance is deleted. In turn, an HLFS related abortion event is raised. Should the spot caching be reissued at a second time, the caching is started from the `initialization` state, with the associated action to clean and initialize a new HLFS file system instance.

If without unrecoverable failures, the HLFS will be established eventually. The application can then start to invoke file system related operations onto the HLFS. In our design, each operation has an interceptor deployed at the beginning of the operation, to keep track of the state of the HLFS and store the historical state information into a state monitor. Each HLFS instance is attached with a state monitor instance. The state information can include, for example, which files have been read/deleted/modified, which file attributes (in particular, the read-only attribute) have been modified, and when is the last time the read/write operation was issued.

Through the interceptor, the actual overall size of the file system is monitored as well, and compared against the total affordable space provided by the NFFS. If based on the model of the NFFS, the overall size exceeds the physical capacity of the NFFS, an exception on “file system is full” is thrown. This proactive measure allows the application to promptly handle the file system full exception, instead of being unaware of such failure situation until the late synchronization stage between the NFFS and HLFS, as at that time it is too late for the application to recover the failure by returning to a proper historical state.

When the HLFS is required to perform data synchronization between the local cache and the memory spot, the second state machine, i.e., the data synchronization state machine, is formed, as shown in Figure 4. By examining the state monitor attached to the HLFS, an update scheduler can determine at each state, what are the reader/writer APIs that need to be performed to the NFFS. For instance, at the “file deletion”

state, the files that need to be deleted are determined by the scheduler and then file deletion operations are carried out in sequence.

Similar to the caching, data synchronization will continue until it finishes successfully, in which some transient errors might be encountered, but with no unrecoverable failures. If an unrecoverable failure happens, data synchronization is aborted. Unlike the caching, which does not modify data on the spot, the premature abortion during data synchronization does introduce data inconsistency. However, because the local HLFS is always available and keeps the most recent and consistent files, as long as no timing limitation is imposed on how long the data communication session should be, data synchronization can be repeated, if the most recent synchronization abortion is due to the situation that the number of the transient failures exceeds the predefined limit. For the situation in which the inherent hardware failures happens (such as file system corruption), the recovery action can be taken by the user to reformat the spot (e.g., when the NFFS is corrupted) or even replace the spot (e.g., when the spot is totally destroyed), once such a failure is communicated to the user. As a result, because of the HLFS, data synchronization to the NFFS is always able to be completed eventually.

4.3 File System Efficient Updating

A popular application category for memory spot is data journaling, which holds a collection of data records to keep track of the history of a logical entity over time. Voice spot and warranty spot described in Section 2 indeed fall into this specific category. Once a record is introduced, it should stay permanently. When the content of a record needs to be updated, in order to preserve the history, a new data record is created, along with additional information to denote the "update" relationship between the two data records. Such a requirement is similar to version control system, in which a versioned object should never be destroyed.

In each data communication session, the application might introduce several new data records. The binary file, that is, the serialization result of the data tree, does not always have the newly introduced records to be located at the end of the file. This can happen, if there exist multiple row data types in the application, each of which corresponds to a different table, and all the tables are serialized into one single binary file. The insertion of a new record into a table that is not the last one in the serialization sequence, will certainly lead to the insertion of a byte sequence in the middle of the binary file.

Following a conventional file system's updating mechanism, the old file would have to be removed from the memory spot, before the updated file is copied to the spot. This happens even when only one byte in the file is modified. To a flash memory device like memory spot, this update approach becomes inevitably slow because of flash memory's electronic characteristics. In fact, in flash memory, when a block is written once, further data update onto the block needs to be erased first. This process is called re-programming. Correspondingly, the first-time write to a flash memory block is called programming. Re-programming is a much slow process compared to read and programming [26, 6, 15]. To the particular memory spot with 32KB storage capacity that we have, the erase of a block takes 20 ms, whereas byte-writing takes only 20 us (and therefore for a 128B block, it takes 2.56 ms). In contrast, reading is extremely fast. In fact, reading a byte only takes 60ns in this particular memory spot chip.

As a result, when no free memory block is left, data updating to memory spot requires block re-programming. In addition, such updating is always over the unreliable communication channel. A scheme that can reduce the data that needs to be actually synchronized over the RF communication channel between the HLFS and the NFFS, becomes very important for data synchronization.

We exploited the data chunking scheme used for file synchronization over the Internet [7], to divide a file into a collection of chunks. The chunk boundaries are uniquely determined by the file's content. Furthermore, localized data modification leads to only localized modification of chunks and thus we only need to update these chunks back to the spot. The mechanism is schematically shown in Figure 5.

The rest of this section shows how to realize the chunking mechanism for each individual file that has been designated for intelligent data updating in the HLFS. This approach does not require the modification of the reader/writer APIs and therefore no change to the NFFS. In the HLFS, the files that are subject to only small modification in each run of the application, are marked as the *fast update* files, and become eligible for efficient data updating. Correspondingly, we introduce a special file system operation in HLFS called *FileUpdate*, which takes the inputs that include the file name, and the binary sequence that represents the entire file content.

Furthermore, we introduce a special binary file called the *super chunk file* into the HFLS, which records the file names of all the fast update files, and the sequence of the hash values of the chunks that belong to

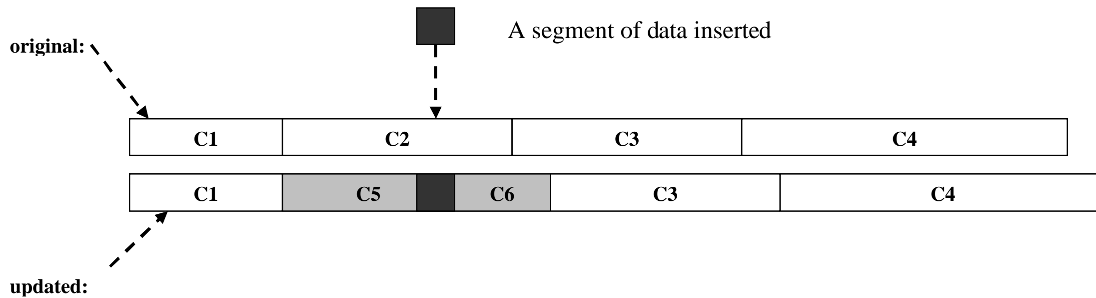

Figure 5: Chunking Modification for Data File Updating

each of the fast update file. This super chunk file is a binary file, whose data semantics is expressed in IDL, as shown in Figure 6.

Following the chunking algorithm, a fast update file is partitioned into multiple chunks. Each chunk is recorded as a file into the native memory spot file system (NFFS), with the file name being the hash value of the entire chunk's byte sequence. The chunks are invisible to the application, since the application only interacts with the HLFS. The HLFS provides chunk aggregation, according to the relationship specified in Figure 6, and exposes only the fast update files to the application but no chunks.

In the NFFS, to distinguish other regular files in the NFFS, a chunk file is assigned with a file attribute, e.g., 0x40000000, and a super chunk file is assigned with a different file attribute, e.g., 0x80000000. The same file attributes are then picked up by the HLFS for data aggregation.

When the application decides to perform data synchronization between the HLFS and the NFFS, the reverse transformation happens. The normal files are treated identically as before, only the fast update files need to undergo the chunking. After the chunking, the new hash value sequence is compared with the previous version of the chunking result recorded in the super chunk file. The update scheduler introduced in Section 4.2 is slightly adjusted to accommodate the fast update files. The adjustment happens to the chunk files, for adding a chunk file (if the chunk is newly introduced), or deleting a chunk file (if the chunk disappears). The super chunk file is updated as well, to reflect the new version of the fast update files.

```
typedef octect[20] HashOfChunk; /*a SHA1 hash value is 160 bits*/

typedef struct FileAttribute {

    string FileName;   /*the file name of the fast update file in HFLS*/

    long size;    /*the size of the file*/

    long creationTime; /*the date when the file is created*/

    long modificationTime; /*the date when the file is created*/

}

typedef struct SuperChunk {

   FileAttribute fileAttributes; /*a fast update file's attributes*/

   sequence <HashOfChunk> correspondingChunks;/*a sequence of chunks*/

};

/*the binary file that contains all the super chunks*/

typedef sequence <SuperChunk> SuperChunks;
```

Figure 6: IDL specification of Super Chunk

To examine the overhead introduced by the chunking algorithm, assume that there is only one fast update file in the HLFS. There are two overheads involved. The first one is due to the update of the super chunk file. The second one is due to the change of the chunk boundaries, when one byte change occurs to a particular fast update file. To update of the supper chunk file, the entire file content needs to be transferred over and written to the spot. The file has a total size of:

$$Size_{File-Attribute} + N_{Chunks} \bullet Size_{Hash}$$

where $Size_{File-Attribute}$ denotes the total meta data size for a file, N_{Chunks} denotes the total number of chunks in the fast update file, and $Size_{Hash}$ denotes the size of the hash value for a chunk.

The second overhead can be found from [7], which can be represented statistically as follows:

$$\alpha \bullet Size_{Average}$$

where α denotes the chunking update overhead with a value less than 2, and $Size_{Average}$ is the average size of the chunk in the fast update file.

Note that the total fast update file can be statistically represented as N_{Chunks}*$Size_{Average}$. As a result, the overhead, compared to the conventional file system file updating that writes the entire file content, can be described statistically as:

$$\frac{(Size_{File-Attribute} + N_{Chunks} \bullet Size_{Hash}) + (\alpha \bullet Size_{Average})}{N_{Chunks} \bullet Size_{Average}}$$

In the implementation, $S_{File-Attribute}$ is 16, $Size_{Hash}$ is 20. The $Size_{Average}$ is controllable. Suppose we choose it to be much larger than $Size_{Hash}$, such as 2K. The overhead can be simplified to become:

$$\frac{\alpha}{N_{Chunks}}$$

For a memory spot with a 32KB capacity, with all the parameters above, and assume the spot holds a single fast update file, the overhead is less than 12.5%. For a 512KB, if we choose 8KB chunks, the overhead is less than 3.1%.

The overhead about is calculated only for one modification occurs to the file. If an update leads to multiple modifications, each one will likely introduce its own overhead. But as long as the chunking yields the chunk number that significantly exceeds α , we always have a win in data updating through chunking.

5. Application and Application Management

This section presents the common functionalities provided by the application framework, regarding memory spot application and application management.

5.1 Event Based Programming

Memory spot applications are designed to be event-based, to respond to the events raised from the application framework. Different layers within the application framework introduce their own events. These events are propagated up to the application level eventually. The application can choose to respond to the specific events that it is interested in and ignore the others. Currently, there are two types of events: *error events* that represent the incurrence of abnormal system behaviors, and *status update* events that represent the status of normal system behaviors. Events are about:

- The proximity and interaction to the physical spot. Examples of events are “spot in range”, and “spot out of range”, “channel not available”, “read progress on channel” and “write progress on channel”. These error or status events are raised from the communication layer;
- The file system and its caching to the host computer. Examples of events are “file system cached”, “file system not formatted”, “file system corrupted”, “file system aborted”, “file permission violation”, etc. These error or status events are raised from the data management unit in the core runtime layer;
- The reader/spot interaction management. Examples of events are “read/write state change” and “cache cleaning failed”. These error or status events are raised from the data management unit as well.

These well-defined error or status events in the framework can be reported to the user with either visual or sound effects, to actively engage the user in this human-in-the-loop system.

Notice that some of the abnormal execution related events, such as “file permission violation”, can have their exception counterparts defined in the application framework. Therefore, right after an abnormal event is raised, the corresponding exception can be thrown. The difference between error events and their paired exceptions, lie in the fact that the exceptions are always propagated in only a single execution thread, whereas to the events, there can be potentially two threads involved: one thread to raise the event, and the other thread to respond to the raised event.

Under this event-based programming paradigm, in a normal data communication session, an application typically experiences the following five phases:

- The application is started and registered to the application framework. The application then subscribes to a collection of events that it is interested in, including the event of “file system cached”;
- The “file system cached” event is raised from the application framework, right after the reader/writer approaches to the spot and the framework successfully caches the data to the HLFS. To respond to this event, the application starts to perform data processing on the cached data;
- If the cache is modified during the application’s data processing, data synchronization between the HLFS and the memory spot is required. This action is performed after receiving the command from the user;
- Finally, the application is closed.

5.2 Memory Spot Application

A generic memory spot application provides common functionalities shared by memory spot applications. Each memory spot application is always inherited from this generic application. The generic application provides the implementation on:

- **Application Start and Shutdown:** the application automatically registers itself to the framework when the application is started, and un-registers itself from the framework when the application is shutdown.
- **Event Registration/Un-Registration and Default Event Handlers:** All the pre-defined runtime events are registered by the application when the application starts, and un-registered when the application is shut down;
- **Data Access and Data Update:** A file can be read from the HLFS, or be written/updated to the HLFS;
- **Data Commit:** the entire HLFS has its data synchronization back to the memory spot, before the application is shutdown.
- **Query on Cached Spots**: More than one memory spot can be cached at the host. Each cached spot can be selected to become an active spot. Once the memory spot becomes active, the application can perform access to its HLFS;
- **Automatic Data Marshaling**: if the application deals with hierarchical data structures, the data tree can be populated and depopulated automatically, by providing the binary file name and the data type of the root node in the tree;
- **Capability Matching:** It ensures that each time the spot data is accessed or updated, the application has the right capability to process the spot data, based on the capability matching specified by the memory spot (detailed in Section 5.3).

With the support from this generic memory spot application, each user-defined application provides both the customized event handlers to override the default implementation, and the capability matching criteria to proactively determine whether the application is capable of processing the data stored on the memory spot.

To distinguish different applications, each user-defined application is assigned with a UUID. Different versions of the same application become distinctive, once each application version is also with a different UUID.

5.3 Dynamic Application Activation

There are two mechanisms to activate the application. The first mechanism, called *static application activation*, is to designate a single application for launching at the time the application framework is started. The second mechanism, called *dynamic application activation*, is to have the application to be launched by the framework automatically, after the spot data is cached into a HLFS and the spot is selected

to become the active spot.

To support dynamic application activation, all dynamic application candidates are located in a predefined directory (or directories). In .NET, these applications are packaged in DLL or EXE format. To determine which application is eligible for activation among a collection of application candidates, capability matching is required. The information on matching criteria comes from the active memory spot. The criteria can be specified in XML in a configuration file that is a read-only (created at spot manufacturing, and no user modification is allowed). The matching criteria can be as simple as to have the application's UUID specified in the configuration. The application framework uses code reflection to retrieve the UUID from the candidate application and determines whether the candidate application has the correct UUID. The XML configuration could additionally provide an Internet location to download the corresponding application code, should the application is not available at the host. A more sophisticated capability matching can be provided via the configuration file that follows an application-specific data schema definition.

Since memory spot can provide sufficient storage capacity, the entire application code can be potentially stored in a spot as well. After the spot data is fully cached into a HLFS, the corresponding DLL or EXE file is copied onto a predefined application loading directory. Consequently, the application is chosen for dynamic launching.

If the application code is downloaded from the memory spot or the Internet, the application management needs to make sure that the digital signature of the downloaded code does come from a trusted software publisher and no code tampering incurs.

6. Data Integrity and Access Control

Data security is critical to memory spot applications in both consumer and enterprise domains. The data stored in the spot can be potentially personal and private, and therefore disallows public access. Unauthorized data modification can potentially have negative legal and financial impacts. Memory spot has built-in hardware support to allow data to be accessed and updated only when the reader/writer is authenticated by the spot [18, 1]. The authentication mechanism follows the challenge/response protocol, similar to the symmetric key authentication mechanism adopted in Smart Card devices [14]. Once the user is authenticated, data access is granted for all data stored on the chip. In the current hardware support, there is no finer application data access enforcement, both for files and hierarchical data structures.

In general, instead of limiting the data on the spot to be privately owned by a particular person, we assume that data on memory spot is accessible and updatable, by different people from different organizations, selectively. Therefore, in the application framework, we were focused on the following two data security features:

- **Data Integrity**: for the data stored on the memory spot, how to verify that data records are really provided by a particular user, and whether they have not been tampered ever since. One particular example is the service records in warranty spot, each of which is stored to memory spot right after the technician completes a computer service;
- **Data Access Control**: how to determine that a particular data record, or a particular data access or update operation, is granted to a particular user. One example in voice spot is that a reviewer can only append the voice records to the spot; only the document owner can format and erase the entire voice spot.

The primary challenge herein is how to provide such data security features, in an environment where we do not have the online Internet access and enterprise-level authentication and authorization security services, at the time the data is accessed or updated to the spot. Consequently, the online access should be postponed by an indefinite period of time, from the time when the data is accessed or updated to the spot. This operational constraint is critical, as the primary advantage offered by memory spot over other referential devices such as RFID is its self-contained local storage.

In addition to the hardware authentication feature mentioned above, we also take advantage of other hardware support. In particular, each spot has a write-once memory segment on the chip. Once the data is written onto this special segment, the data becomes read-only and can not be erased any more. This segment stores the following information:

- A Universally Unique Identifier (UUID) provided by the hardware manufacturer to the spot, as the identity of the memory spot. Currently, the UUID has the size of 128 bytes;
- A memory spot header to record spot-specific and application-specific information.

6.1 Data Integrity

We use digital signature to ensure data integrity. The digital signature is performed on the data records stored on the spot. A data record is referred to a data node located at any arbitrary hierarchy (except at the leaf node level) in a hierarchical data tree constructed from the byte sequence stored on memory spot. Which hierarchy is at the appropriate level to evaluate and then assign digital signature is dependent upon individual applications. Digital signature **S** of a data record **R** is actually embedded in **R** as one of the data fields, denoted as **R**.**signature**. Given **R**, **S** is generated in the following operation sequences:

(1) Assigning the corresponding data field **R**.**signature** to be null;
(2) Applying the data marshaller of **R** (via automatic data mapping described in Section 4.1) to **R**, and obtain byte sequence, B[];
(3) Applying digital signature generation algorithm on B[], to obtain **S**;
(4) Assign **S** to **R**.**signature**.

These steps can be combined and described as:

$$\text{R.signature} = \text{Signature}(\text{Marshal}(\text{R}, \text{R.signature} = \text{null}))$$

Correspondingly, to perform digital signature verification on **R** requires the byte sequence from **Marshal**(**R**, **R.signature** = null). Furthermore, we need to have a public key embedded along with the digital signature. In our implementation, we choose X.509 digital certificate. The X.509 certificate allows us to check other aspects of the certificate, including the holder's name, the organization issuer, and the expiration time. All of this information can be further verified by walking through the Certificate Authority (CA) chain [5] embedded in the digital certificate. For example, the certificate is signed by HP's CA, which can be further signed by VeriSign's CA. To verify the involved CA chain, we need to have all the involved CAs' public key certificates to be pre-installed onto the host computer. To a particular memory spot application, at the time the application is installed, the digital certificates of the involved CAs have to be installed. Alternatively, the certificate verification can be deferred until the time when the online connection is available. We then have all the required public key certificates to be installed, as a one-time effort.

One other issue that can not be ensured without the online connection is digital certificate revocation, if the digital certificate is revoked before the certificate is expired. If such checking is required, similarly, we can have it deferred until the online service infrastructure becomes available.

Digital signature can be applied to each individual file. What is required is to create a data record that contains a fully qualified file name to the file, and a byte sequence to hold the file's digital signature and public certificate. Similarly, Digital signature can be applied to the entire memory spot's application data. This can be useful for memory spot backup, in case the memory spot is either accidentally corrupted or maliciously destroyed. From the digital signature and the public certificate, we can determine whether the backup data is corrupted, and who is the person that performed the data backup.

6.2 Data Access Control

Recall from Section 4, application data represents itself in three different ways. At the application level, data access is viewed as different data records in a hierarchical data tree. At the file system level (either in a HLFS or a NFFS), data is represented in a binary file. In the physical memory spot chip, data is represented as a collection of byte segments stored in different flash-memory blocks. In the application framework, our focus on data access control is at both application data records, and the file system in HLFS. We consider the file system as a whole as a particular resource, which supports the operations such as formatting.

Therefore, application data records and files are two different types of resources under our software data access control. Each resource is with a fully qualified name. In terms of data records, the qualified name is derived, by starting from the root node with name of "/", and descending down to the particular data element under the current interest, with the type names of the nested data structure elements along the path to be concatenated in sequence. For example, the data element: page number, of the voice record described in Figure 2, is "/VoiceSpotData/VoiceRecords/PersonRecord/Records/VoiceRecord/PageNumber". To a file, since the current NFFS and thus the HLFS only support a flat file system with only one directory level, the name of the file is the file's fully qualified name.

Regarding data records, the right can be the following:

(1) create/update a composite tree node;

(2) add/remove/update a child element node in the tree node that represents an array or sequence of data elements.

With respect to the NFFS and HLFS file system, the rights involved are read, write and update of the files. Note that update is distinguished from write, as we have a particular update operation introduced in Section 4.3.

We augmented the IDL to allow access control on hierarchical data structures, by having access control to a data structure become an attribute associated with the data structure. Correspondingly, the IDL compiler is modified for code generation to accommodate such language extension. An example of access control specification is shown in Figure 7.

```
[owner (create, modify)]
struct VoiceRecord {
        long PageNumber; /*the page number associated with the document*/
        long StartTime;  /*the start time of the voice annotation*/
        long EndTime;   /*the end time of the voice annotation*/
        string StoredFileName; /*where the actual voice media is recorded*/
};
[owner (create, add, remove), reviewer(read, add)]
typedef sequence<VoiceRecord> VoiceRecords; /*each person can have multiple voice records*/
```

Figure 7: Role-Based Access Right Specification in IDL

So far, we assume that the read right is always granted to users. To enforce read access, we can resort to the hardware authentication that we mentioned early. That is, if a user is able to pass the spot's authentication, the user is automatically granted the right to read, for all the data stored on the spot.

We adopt the certificate-based authorization approach [22] to enforce correct resource access in memory spot applications. The enforcement is only performed for data write and update. With this certificate-based approach, the access control enforcement can be done in the offline environment, once all the necessary certificates have been downloaded in advance from the online trusted environment. The entire access control subsystem is shown schematically in Figure 8. Every time the resource access request is issued from either the application data manager regarding hierarchical data access, or the HLFS regarding file system access, the policy engine will interpret the rules and determine whether the resource access should be granted.

Access Control Related Attributes and Certificates

To support policy verification, the following information is stored permanently in the memory spot header:

(1) The application policy certificate, along with the digital signature of the policy certificate and the public key certificate of the signing authority. Alternatively, what needs to be stored includes only the digital signature and the URL location to download the policy certificate and the public key certificate. In an online environment, the certificates are downloaded. By digital signature comparison, the policy engine can determine whether the policy certificate downloaded really is the one specified by the target memory spot;

(2) the application UUID, so that the policy engine can tie the current application being enforced to the target memory spot;

(3) Other attributes that are related to memory spot and the application. e.g., the specification of the organization (s) in which the spot will be owned. For voice spot, we can specify the organization is "hp.com", for example. For the memory spot that is attached to a physical passport, we can have the name of home country associated with the passport holder to be specified.

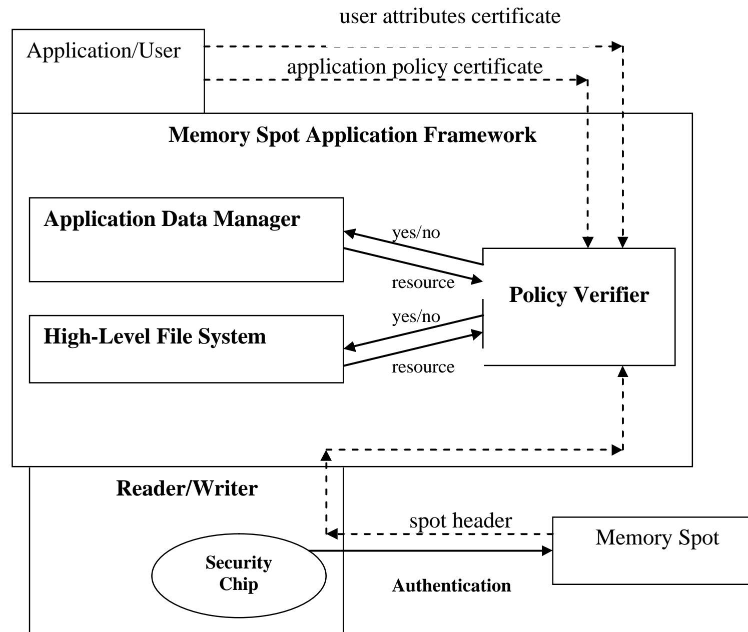

Figure 8: Certificate-Based Access Control on Memory Spot

Aside from the policy certificate which is either stored in the spot or downloaded from the Internet, the user attributes certificate is required by the policy engine. This user attributes certificate specifies a unique role that the current user belongs to. The user needs to install this certificate in a user-specific environment, before the application is started. The role can be explicitly specified in the attributes certificate. Alternatively, the role can be dynamically evaluated, based on some declarative specification on spot header information and even some application data information. For example, the role of the voice spot's owner, can be dynamically evaluated by having the current spot's UUID to match the spot identifier specified in the user attributes certificates. If not matched, then the role is defaulted to be reviewer. Another example is to have the spot to record the electronic passport, in which the role of the passport of "visitor", can be determined by comparing the current user's organization attribute on the attributes certificate with the organization attribute specified on the spot. In the user attributes certificate, an expiration date can be issued, if we intend to have the right granted to only within a finite period.

With all the three pieces of information: the application policy certificate, the user attributes certificate, and memory spot header, the policy engine is able to perform policy enforcement to a particular application that is accessing the spot data on behalf of the current user.

Policy Engine

Data access is enforced at different data access points. To the application hierarchical data structures, the enforcement is performed when application data synchronization to memory spot is called. By comparing the newly updated tree with the existing tree on the cache, the policy engine can go over the rules specified in the policy certificate one by one and determine whether some rules about data hierarchy are violated, before the corresponding updated binary file (containing the serialized result) is written or updated to the HLFS.

The entire hierarchical data tree constructed in the address space of the application is always visible to the application. It is the application's responsibility to enforce data write/update protection proactively, by following the rules specified in the policy certificate, rather than wait until the application data is serialized and ready for data synchronization. If the policy is violated, the application framework will refuse to perform data synchronization.

With respect to the file system access, it is enforced when the HLFS file system's write/update is called from the application. An internal call from other modules within the application framework is immune

from such enforcement.

In the actual implementation for policy verifier, data access enforcement can be achieved by having the call interceptor deployed at the beginning of each operation involved with resource access. The interceptor delegates the resource request to the policy verifier. The policy verifier can be implemented as a general rule interpretation engine. Alternatively, we can come up with an application-dependent verifier. The policy verifier can be developed at the time the application is developed and packaged as a DLL (called the Verifier DLL). This DLL is digitally signed by the trusted authority to prevent code tampering. To support this code checking, the verifier DLL has a strong name to be associated with, and is forced to link to the application to ensure its association to the correct version of the application. The association can be achieved at the compilation time, by having the application implementation to create a policy verifier instance and register this instance to the application framework runtime.

7. A Programming Example

We use voice spot application to illustrate how a memory spot application is developed in the application framework. The application data schema has already been shown in Figure 2.

7.1 Application

Figure 9 shows the overall application class layout. It is inherited from the generic memory spot application `MemorySpotApplication`. It has a UUID to be assigned as a class level custom attribute. The default constructor provides the application-specific initialization. For this particular application, only two methods defined in `MemorySpotApplication` are required to provide the overridden implementation: the event handler `OnFileSystemCached` for "file system cached", and the application-defined capability matching `CapabilitiesMatch`.

The event "file system cached" is raised after the HLFS is successfully constructed from the cached spot data. In the handler `OnFileSystemCached`, the most recent memory spot that has been cached is retrieved, via the method called `LatestMEmorySpot`. The associated memory spot is designated as the active memory spot, via the assignment of the property called `ActiveMemorySpot`. The root node of the application data is created, and the method on data retrieval is performed via the method called `RetrieveData`. Once the data is populated into the in-memory data tree, if no exceptions are encountered, the event `ApplicationDataReadyEvent` is to signal that application data is ready to use. All the three methods mentioned above, namely `LatestMEmorySpot`, `ActiveMemorySpot` and `RetrieveData`, are all declared and implemented in `MemorySpotApplication`. However, the event, `ApplicationDataReadyEvent`, is specific to this application, and thus is only introduced in this particular application. Other high-level application processing can subscribe to this application-defined event in order to perform further data processing.

As we do not provide our own user-specific capability matching, the method `CapabilityMatching` simply returns true.

7.2 Application Start and Shutdown

The programming pattern on how to start and shutdown application is shown in Figure 10. The application is registered to the application runtime by identifying itself with its own UUID, and then the runtime is launched, following static application activation. Once the application is finished with possible data synchronization back to the spot, we can shutdown the entire application runtime.

8. Warranty Spot

This section describes how to exploit memory spot to reduce warranty fraud in computers. We explain computer-related warranty fraud, and why a memory spot attached physically to the computer helps reduce warranty fraud. We also present the service architecture on warranty spot application, which is an enterprise service application that spans across multiple organizations. We also present the application prototype to demonstrate our application framework.

```
[GuidAttribute("CA761232-ED42-11CE-BACD-00AA0057B223")]
public class VoiceSpotApp: MemorySpotApplication
{
   public VoiceSpotApp() {
     …..
   }
   public override void OnFileSystemCached (object sender, MemorySpotFileSystemEventArgs args) {
    try
    {
       //(1) Retrieve the most recent memory spot as the active spot
       IMemorySpot spot = this.LatestMemorySpot();
       this.ActiveMemorySpot = spot;
       //(2) Retrieve the data from the spot
       IMemorySpotApplicationData data =
           new MemorySpotApplicationData
                    (ApplicationDataTransformation.Unmarshal,
                     new PersonRecordsHolder());
       RetrieveData (data, "VoiceAnnotate.dat");
       AppDataReadyEventArgs args = new AppDataReadyEventArgs(data);
       OnApplicationDataReadyEvent (args);
    }
    catch (ApplicationDataAccessException)
    {
          ….
    }
  }
  public override bool CapabilitiesMatch (IMemorySpotApplicationManifest  capabilities){
    return true;
  }
}
```

Figure 9: The Overall Voice Spot Application Class Definition

8.1 Computer Warranty Fraud

HP has recently confirmed that 6% to 8% of its warranty claims are fraudulent, an estimated $142 to $189 million in warranty fraud per year [23]. The root cause is that there is no integrated information system to systematically keep track of computer hardware configuration, computer service records, and warranty entitlements associated with a computer system and its major hardware components, over the lifetime of the computer. Its lifetime is from the time the computer is at the end of the computer assembly line to the time when the computer's parts are being upgraded, repaired or replaced, and ultimately when the computer retires. Currently, such computer-related information is only available at the disjoint databases scattered across different organizations. To acquire comprehensive information about a computer under warranty is time consuming, or very often impossible because of cross-organizational access availability (authorization, internet connectivity, etc). In fact, the whole warranty entitlement creation, update and enforcement process involves other organizations external to HP, including distribution channels, authorized service providers (ASP), etc. In particular, we see the following problems that contribute to frequent warranty fraud:

- Inability to tie constituent subassemblies and components to the computer as a whole. Without such system/part relationships recorded, damaged parts from out-of-warranty computers can be swapped into the computers under warranty and therefore become eligible for warranty service;

```
//start the application, with the GUID being the application's identifier
Guid appID = new Guid("CA761232-ED42-11CE-BACD-00AA0057B223");
//the runtime is a singleton
MemorySpotRuntime runtime = MemorySpotRuntime.Instance;
runtime.ApplicationID = appID;
runtime.Start();
…..
//to shut down the application
runtime.Shutdown();
```

Figure 10: Start and Shutdown the Memory Spot Application

- Inability to aggregate all information about a computer from the disjoint databases to make an accurate assessment for a service request regarding this computer. In practice, to ensure customer satisfaction, if no evidence is found to invalidate the service request in 4 minutes, the request has to be granted;
- Inability to access centralized warranty databases by field technicians from an external ASP, as they usually do not have connectivity or access rights to the HP-owned databases. However, they are authorized to initiate the service based on limited computer information;
- Inability to determine at the time of field service whether the malfunctioned part is installed by an authorized personnel. Should such a situation be confirmed, the warranty is voided.

The Warranty Fraud Monitoring (WFM) database in HP [24] was built for such backend processing. It records computer-level warranty entitlements, service logging (with reported symptoms) at the call center, services being dispatched, and part-to-customer shipping history. With the WFM and other databases, a data mining tool has been developed to identify warranty fraud. Although the tool has shown its effectiveness, it is not designed to proactively stop warranty fraud at the first place.

8.2 Information Fusion via Memory Spot

To proactively reduce warranty fraud, instead of consolidating the disjointed databases that would require a tremendous effort, we adopt an approach that facilitates computer history information to be available right next to the computer, and enables proactive fraud prevention. It is based on memory spot, with the spot being permanently attached to the computer. The spot records the computer's hardware configuration (mother board, memory modules, etc), service records (part repair/upgrade/replacement), and warranty entitlements of the computer and its parts over its lifetime [16]. We called this particular memory spot Warranty Spot.

The spot is first created and populated at the end of the manufacturing assembly line. System configuration and warranty information – details about all the constituent components of the computer – is written onto the spot. The spot is further updated at the distributors/value-added resellers to reflect warranty purchase or hardware customization. Once the computer is in service at the customer site, the spot always contains an up-to-date version of the warranty and configuration of the computer, which can be accessed locally, independent of whether the computer is operative or not. Storing such information to the hard drive of the computer does not work if the computer can not be booted up, such as encountering hard disk crash. The service technicians can immediately determine whether the part under service request has valid warranty, whether the part was originally a part of the computer under service, and whether the part was swapped in. The service technician updates the service details to the spot after finishing the service. Thus the spot stores a living history of the computer. For end-user replaceable parts (EURP), the customer should manage to obtain the reader/writer device to update service records.

Besides decision-making assistance right next to computer, this comprehensive computer-related information can be serialized as a flat file and transported via file upload or email attachment to remote

parties, such as the call center, claims administration, return part processing center, etc. At the receiver, complete structural information from the spot is reconstructed to grant warranty claims, to determine replacement parts, to close service cases, etc.

Overall, the warranty spot is a data fusion point for different people across different organizations. The comprehensive computer-related information is self-contained, and locally accessible and updateable, without relying on a centralized system. Further, the spot and reader/writer device are designed to be low-cost (less than $1/spot). Thus this solution is viable, especially for higher-end computer server systems.

8.3 Solution Architecture

Figure 11 shows the architecture for the warranty spot application. In the application, built upon the application framework is the data processing module, which contains two sub-modules. The first one is the In-Memory Database (IMDB), which is created on-the-fly from the data stored on the spot. It has a relational database that handles computer part/service/warranty entitlement (3 tables) input, update and SQL query-based inspection. Information navigation between different tables becomes available via the defined table relationships. The process of serializing all the computer-related information into a flat file, called *imaging*, can be performed via the data management services exposed by the application framework. The spot image is used for the purposes like warranty entitlement enforcement and data backup. Instead of real-time synchronization with the centralized databases, a backup service that prevents data loss requires minimum changes to the current IT infrastructure with simple and cost-effective implementation. Once the spot images are backed up, they become valuable for data mining tools beause they embody the updated history of the associated computers.

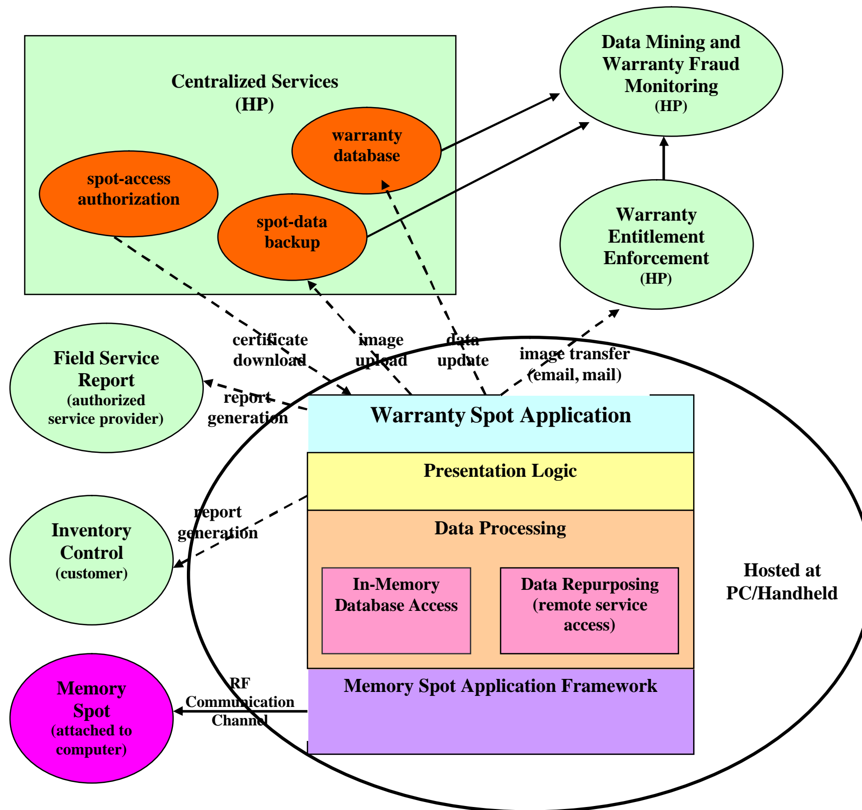

Figure 11: Overall Warranty Spot Solution Architecture

The second sub-module is on data repurposing, which transforms local data information on the spot for other purposes, e.g., to update the customer's data center about the most recent computer configuration, to

populate field reports on the service just performed to the ASP's data center, to update the centralized database with warranty entitlements just entered to the spot, etc. Note that all these actions can be deferred until an online connection is available.

8.4 Application Prototype

The prototyping application has been implemented and successfully demonstrated. The application prototype provides the implementation on the portion that is hosted on the PC/handheld. Shown in Figure 11, the implemented portion covers all the modules within the largest circle. In our implementation, we use the Tablet PC running on Windows XP as the hosting platform. In addition to the general reading and updating a memory spot offered by the application framework, the application prototype is capable of viewing and inserting records for hardware parts, the associated warranty, and the service records that deal with repairing/upgrading/diagnosing the old or new hardware parts. Furthermore, each time a successful update of the spot is finished, a copy of the application data is created into a binary file and stored in a pre-defined directory (implemented with .NET isolated storage) on the host, ready to be uploaded to the online service provider for spot backup. The host also keeps track of all the service records done on this host by the technician. A XML-based render is able to transform each service record into a nice field report in an Internet browser. To support digital signature on each data record, the user needs to have a X.509 certificate installed to the host. The screen shot of the application in action is shown in Figure 12.

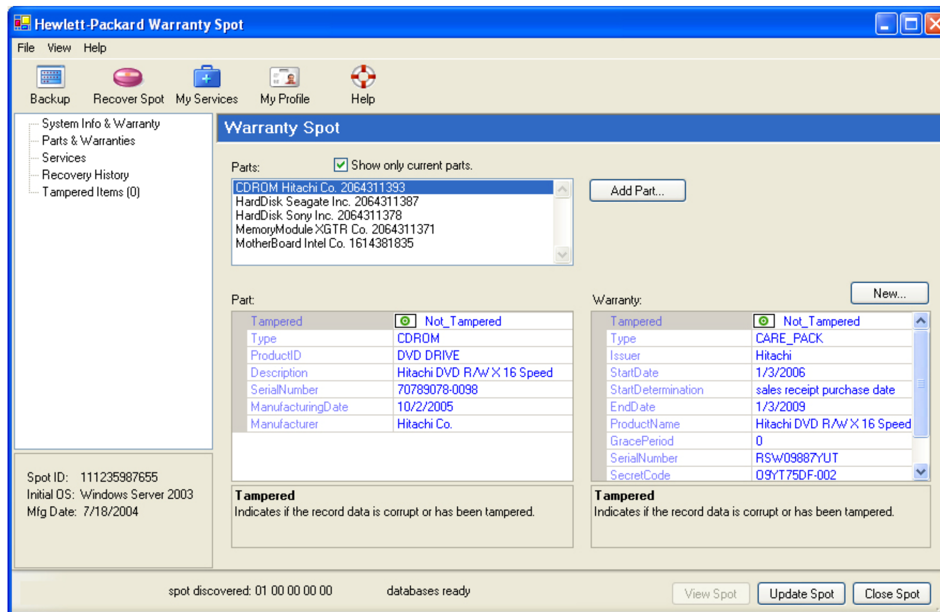

Figure 12: Screen Shot from Warranty Spot Application

9. Related Work

In this section, we identify the physical devices that share similar physical properties with memory spot. These devices include the RFID, Smart Card devices and the NFC devices. We focus on the comparison of the involved application development frameworks associated with these devices, to the one that we developed for memory spot.

RFID chips only provide referential information (e.g., the EPC code about the product that the chip is attached to, with 96-bit in storage). The RFID chips typically are read-only. Regarding the interaction with hardware chips, the RFID middleware [19] is concerned with how to collect the data from multiple chips simultaneously in a wireless broadcast channel. In terms of the high-level functionalities that span from the low-level device interaction to the high-level enterprise business applications, our memory spot application

framework shares certain functionalities with the Ovum's model [19] at the bottom three layers: data collection, data cache, and device management. Once the RFID data is successfully retrieved at the data collection layer, an event then propagates to the high-level middleware layer that deals with business information processing, such as database access, product tracking, inventory control, etc. Because the RFID chips only store the referential identifier information, unlike our memory spot application framework, there is no data management capability like data caching (which has different semantic definition in Ovum's software stack) and synchronization, data serialization and data integrity checking, for individual chips.

A smart card device provides an on-chip processor running an operating system to support on-chip downloadable applications [14]. An on-chip flash-memory based file system offers non-volatile and re-programmable storage, with limited storage capacity (4 KB or less). A smart card is often personally owned for applications like Secure-ID for identity authentication, or secure financial transaction with on-chip crypto-coprocessor [11]. The Java Card [13] enables application development with restricted Java features supported by a trimmed down version of the Java VM. When a smart card device is in operation, the reader and the card are in a contact mode. As a result, no eventing on the unreliable communication channel, as well as efficient and unreliable data caching and synchronization over this communication channel, is supported. Furthermore, no similar techniques are provided on smart card devices for automatic data mapping, data integrity and data access control, at the finer granularity than the file system support. A service framework called the Java Card Web Servlet (JCWS), which runs on a host computer (PC), is presented in [4]. This framework receives a remote service request from a local (remote) browser, and interacts with the service proxy installed on the smart card to access the on-card file-based resources, such as files and applets. In terms of data security, hardware-level authentication on smart card has been addressed [14]. On-card file-level role-based access control is also addressed in some Smart Card Operating Systems, such as in Microsoft Windows Card [20]. [3] exploits the use of role-based access control on files, each of which holds a database table or a view.

NFC devices exploit the similar microchip storage technology as smart cards have, except that NFC reader devices interact with the microchip through the wireless connectivity [17]. Therefore, the interaction is in a contact-less mode. In fact, memory spot can be viewed as one member of the NFC device family. Sony's FeliCa [9] is a popular NFC device widely used in Japan and other Asian countries to enable online payments via mobile phones, for the applications like ticketing, public transport access and retail store purchase. The device provides 5Kb and operates at 13.56MHz. MicroPass is another NFC chip that provides up to 64KB [12] storage capacity on the same carrier frequency of 13.56MHZ. Because of the carrier frequency difference, compared with memory spot, Felica card has much larger chip size (26.5mm in diameter, for the smallest variant of the Felica cards). In addition to limited chip storage, most of the applications currently available for these NFC microchip devices have very restricted data types, such as an integer counter for a transport pass, and a bank card that only contains user identifier, card number and expiration date. We are not aware of other application frameworks that provide the data management capabilities as comprehensive as our framework.

The common objective of flash-memory based file systems [26, 6, 15] is to increase the I/O throughput of data reading/writing to the storage device. Such a file system is an integrated part of the operating system and needs to be always available, independent of the status of application execution. Instead, memory spot applications mostly perform data processing on the cached data at the host computer. Data synchronization between the application and the storage device are only required at the beginning and the end of the application. In such a disconnected computing environment, one primary objective for our application framework is to minimize the complete data update from the host computer to the spot when the application is terminated (not the other way around, as reading from the spots is very fast). This objective, along with the unique operational environment (e.g., the RF link to the host computer, very primitive read/write memory cell operations and very limited processing power on the spot, and the powerful processing unit at the host), requires the data synchronization module being developed with a completely different device architecture and internal data structures. In contrast, on-board SRAM storage buffers are required in [26] and [6], and multiple memory chips to form multiple banks, rather than just one single chip, are assumed in [26] and [15].

10. Conclusions

We have developed an application framework to enable rapid application development for a microchip device called memory spot, a member of the NFC storage device family. The application framework has

been implemented, with the first release of a .NET SDK 1.0 made available in February 2006. All the techniques described in Sections 4-6 regarding data management, application management and data security, have been implemented, except for the chunk-based efficient data updating described in Section 4 and the data access control scheme described in Section 6, which are still pending.

To measure data access related performance, we chose a memory spot chip with 32KB storage capacity, and a host machine which is a HP Tablet PC Tx1100 (Pentium M Processor with 1.1GHz, and 768MB RAM). The baseline measurement was performed on the reader/writer APIs. We found that reading a file of 27KB (with randomized content) takes 99 ms, and writing a file of 27KB (with randomized content) takes 3.5 seconds. With our application framework, we found that writing a file of 27KB (with randomized content) to the HLFS and subsequently updating it to the memory spot, totally takes 3.6 seconds.

There is a lot of room for future framework improvement, which includes:

- Using a transactional file system [10] in either the NFFS under the reader/writer APIs (with possible hardware support), or the HLFS above the reader/writer APIs (no hardware support required), or having the NFFS and HLFS work together, to prevent data loss in data synchronization.
- Having multiple spots to be chained together to form a software-enabled jumbo spot that offers scalable storage to meet the application needs.
- Designing a software/hardware platform to secure the entire application framework to counter malicious attack, beyond the code security mechanism provided by the current Microsoft .NET framework.

We have demonstrated the warranty spot application to various people within HP, in particular to the HP Warranty Fraud Investigation team, to seek the opportunity of a field trial on adding memory spot to HP computer products to demonstrate its effectiveness in reducing warranty fraud. Recently, to continue to explore the applications and service opportunities for memory spot, we have started to work on a device-centric service platform. This platform will incorporate both the memory spot application framework and the memory spot reader/writer hardware, by viewing the reader/writer as a particular accessory device.

11. Acknowledgements

We are grateful to the memory spot team at HPL Bristol for closely working with us to help develop this application framework. In particular, the bi-weekly technical meetings with Fraser Dickin and Tom Rathbone allowed us to understand the hardware behavior of the reader/writer and memory spot, to evaluate memory spot application development ideas based on the underlying hardware features, to communicate high-level software requirements on the reader/writer APIs, and finally to integrate the application framework with the real hardware device. Helen Balinsky and Weng-Wah Loh provided valuable information and discussion on exploiting memory spot hardware security features. Joseph Ku from HP Global IT offered us his thoughts on warranty spot and deep domain knowledge on computer service and computer warranty fraud. Jeff Soesbe from HP IPG helped propose supporting the application development framework in the ChaiVM embedded platform for high-end LaserJets and in an EIO or EX accessory product (like JetDirect or WinCompanion). Kave Eshghi helped us understand the chunking algorithm and provided suggestions on applying his chunking algorithm to this special micro-chip storage domain. Ismail Ari helped review the paper and offered his insight on file systems. Finally, Mohamed Dekhil, Henry Sang, Ed McDonnell, John Waters, Keith Moore, and Neerja Raman in the Imaging Systems Lab (ISL) provided management support and sponsorship to enable the successful development of this work.

12. References

[1] H. Balinsky, M. Bresciani, W.W. Loh, L. Chen, K. Harrison, R. Castle, E. McDonnell, “A Secure Authentication Solution Using Memory Spot,” HP TechCon 2006.

[2] BBC Online: Tiny Wireless Memory Chip Debuts, July 2006, http://news.bbc.co.uk/1/hi/technology/5186650.stm.

[3] C. Bolchini, F. A. Schreiber, “Smart Card Embedded Information Systems: A Methodology for Privacy Oriented Architectural Design,” Data & Knowledge Engineering 41, pp. 159-182, 2002.

[4] A. Chan, J. Cao, H. Chan, and G. Young, "A Web-Enabled Framework for Smart Card Applications in Health Services," Communications of the ACM, Sept. 2001, Vol. 44, No.9, pp. 77-82.

[5] J. D. Clercq, *Windows Server 2003 Security Infrastructure*, Elsevier Digital Press, 2004.

[6] H. Dai, M. Neufeld, and R. Han, "ELF: An Efficient Log-Structured Flash File System for Wireless Micro Sensor Nodes," In Proceedings of the 2nd ACM Conference on Embedded Networked Sensor Systems, Nov. 2004, pp. 176-187.

[7] K. Eshghi, M. Lillibridge, J. Suermondt, "The Elephant Store: Efficient Synchronization and Management of Large Data," HP Tech Con 2005.

[8] K. Farkas, E. de Lara, "New Products," IEEE Pervasive Computing vol.5, no.4 : 12-14, Oct.-Dec. 2006.

[9] Felica Product Information, accessible at http://www.sony.net/Products/felica/.

[10] E. Gal and S. Toledo, "A Transactional Flash File System for Microcontrollers," In *Proceedings of the USENIX Annual Technical Meeting*, 2005.

[11] E. M. Hamann, H. Henn, T. Schack, and F. Seliger, "Securing E-Business Applications Using Smart Cards," IBM Systems Journal, Vol. 40, Number 3, 2001.

[12] Inside, http://www.insidecontactless.com/

[13] Java Card, http://java.sun.com/products/javacard/.

[14] T. M. Jurgensen, S. B. Guthery, *Smart Cards: The Developer's Toolkit*, Prentice Hall, 2002.

[15] A. Kawaguchi, S. Nishioka, H. Motoda, "A Flash-Memory Based File system," Proceedings of the 1995 USENIX Technical Conference. USENIX Assoc, Berkeley, CA, USA. pp. 155-64, 1995.

[16] J. Ku, "System Serviceability Enhancement by RF Powered Storage Device," Internal Presentation.

[17] D. Linsalata, "Improving the Consumer Mobile Experience through Near-Field Communication," IDC Report, Dec. 2004.

[18] W. W. Loh, F. Dickin, J. Waters, "High-Performance Protocol Processor for Memory-Spot," HP TechCon 2006.

[19] N. Macehiter and A. Woodward, "RFID and Enterprise IT," Market Research Report by Ovum, Jan. 2005.

[20] Microsoft Corporation. Windows for Smart Cards Toolkit for Visual Basic 6.0. www.microsoft.com/windowsce/smartcard/, 2000.

[21] K. Moore, and E. Kirshenbaum, "Building Evolvable Systems: the ORBlite Project," *Hewlett-Packard Journal*, pp. 62-72, Vol. 48, No.1, Feb. 1997.

[22] M. Thompson, A. Essiari, and S. Mudumbai, "Certificate-Based Authorization Policy in a PKI Environment", *ACM Transactions on Information and System Security*, Vol. 6, No. 4, Nov. 2003, pp. 566-588.

[23] Warranty Chain Management Conference 2005, accessible at: http://www.warrantyweek.com/archive/ww20050419.html.

[24] Warranty Fraud Monitoring, accessible at http://iss-tce.cca.hp.com/eps/epma/search/.

[25] J. Waters and R. Castle, "Power & Data Transfer for A New Class of Contact-Less Storage Device," HP Tech Con 2003.

[26] M. Wu and W. Zwaenepoel, "Envy: A Non-Volatile, Main Memory Storage System," SIGPLAN Notices. Vol. 29, Issue. 11, pp. 86-97; Nov. 1994.