1

Implementation of Transactional Mechanisms for Open SmartCard

Sylvain Lecomte[1], Gilles Grimaud[1], Didier Donsez[2]

[1]RD2P
Bat. M3, Cité Scientifique
59655 Villeneuve d'Ascq Cédex France
{lecomte, grimaud}@lifl.fr,

[2]LAMIH
Le Mont Houy BP 311
59304 Valenciennes Cedex France
donsez@univ-valenciennes.fr

Abstract. Smartcard is well adapted to store confidential data and to provide secure services in a mobile and distributed environment. But many cases of smartcard application failure can corrupt data in smartcard persistent memory. In this paper, we propose a recoverable persistent memory to maintain data consistency in a smartcard. Then, we adapt and compare two recovery algorithms used in Database Management Systems (shadow paging and before-image logging) to the smartcard memory features. At last, we present a prototype, which demonstrates the feasibility of these algorithms in a smartcard.

Problems of failure in SmartCard

Smartcard is well adapted to store confidential data and to provide secure services in a mobile and distributed environment [1]. So, the range of their applications is continuously growing (electronic commerce, medicine, phone card, etc). These applications are more and more designed as distributed applications [2][3]. The computation of the application is spread over several computers on a network. The smartcard is one of these computers when it is slotted into a terminal.

However, as in distributed systems, failures should have been considered. We distinguish the node failure in which a node (smartcard or other) crash and the communication failure between nodes. These failures can introduce incoherence in the smartcard (and in the other nodes) data. Smartcard can complete the computation (for example to credit an electronic purse) and valid updates of its data after the communication failure even if the other nodes decided to discard their part of the computation (for example to debit a bank account). The smartcard crash may cause also an incoherence in the data since a part of updates may be in RAM and lost by the crash.

The developer should have to handle these problems. The communication failure can be resolve by distributed validation protocols such as the two-phase commit protocol [4]. A simple way to manage the incoherence of the data during a smartcard crash is to restore the previous state of data, i.e. the state of the data at the beginning of the computation.

Transactional Model and Recovery Persistent Memory

A simple way to manage failure is to make atomic all updates done by an application (distributed or centralized). Atomicity of updates done by a group of code lines is a part of a larger concept, the concept of transaction [4]. A transaction is a group of actions (i.e. code lines). The transaction completes in two ways: the *Commit* in which all effects (i.e. updates) is durable either the *Abort* in which all effects are discarded. So, the transaction guaranties the ACID properties (Atomicity, Coherence, Isolation and Durability). Atomicity means that all or none of the modifications are validated. Consistency says that a transaction takes data in a coherent state and returns it in a coherent state. Isolation isolates the effect on the transaction until the commit so concurrent transactions can not seen the uncommitted updates. Durability enforces that effects of the transaction are accessible even in case of failure (media...).

In the smartcard context, the natural group of actions for which Atomicity and Durability must be guarantied is the APDU command (Application Protocol Data Unit) [5]. But the transactional model can be used to execute atomicaly a group of

actions over several APDU commands or several cards sessions (i.e. the card is plugged several times in different terminals before the commitment of a transaction) [2]. On the other hand, a command APDU may be composed of several transactions.

New models and platforms for application development hide the difference between volatile (RAM) and non-volatile memories (Hard Disks...) by supplying Recoverable Persistent Memory (RPM) [6][7]. RPM provides mechanisms to allow the atomicity, since it can undo updates or make them persistent at commit time. Many works have focused on RPM in Operating Systems (OS) and in Database Management Systems (DBMS) [8][9]. The main techniques are described by [10]. These techniques suppose that the non-volatile memories are magnetic media such as hard disks or tapes to guaranty the properties of Atomicity and Durability.

To store persistently data, DBMS and OS consider mainly block devices such as hard disks or tapes to make persistent data. These media can not be addressed directly by the CPU so data are loaded by bulk in the buffers in RAM. However some recovery algorithms [11][12] consider special hardware such as UPS-RAM (Uninterruptible Power Supply RAM) to improve the performance of logging operations by maintaining safe the logging buffers.

Recovery Manager in SmartCard

Smartcard persistent memories provide different features than magnetic media. Indeed, smartcard manufacturers use mainly the 3 following technologies to implant persistent memory in SmartCard: EEPROM, FeRAM, and FlashRAM. With these technologies, CPU can address (i.e. read and write) directly memory cells.

But in case of Flash-RAM, a quite long erasing operation (~100 ms) must precede writing operation. So the Memory Manager (MM) must emulate a paginated memory with a page size close to 8 to 16 words. MM imposes to write in non-volatile memory only through a RAM cache [13] so it translates logical addresses of data toward RAM or non-volatile memory addresses. But this needs a translation table residing in RAM and sometimes in non-volatile memory. Caching in RAM extends also smartcard's life duration since writing operations damage EEPROM media (manufacturers guaranty only 10000 writing operations in an EEPROM cells) [14].

Taking into account these features, we adapt two recovery algorithms designed for Database Management Systems to the smartcard persistent recoverable memory manager. The two algorithms are the "Shadow Pages" algorithm and the " Before-Image Logging " algorithm. The following sections present efficient and simple RPM over EEPROM and FlashRAM technologies. Nevertheless, we do not consider the After-Images Logging which is very equivalent to the Shadow page algorithm in the smartcard context.

Adaptation of the Shadow Pages Algorithm

The shadow page algorithm preserves the original state of data. Each write operation into a page is performed on a copy of this page, named "shadow page". In order of that, the algorithm uses a translation table. With FlashRAM, this table is loaded in RAM from non-volatile memory at the beginning of the transaction. All writing operations are performed in shadow pages allocated from the list of free pages. The translation table maps toward the shadow pages. To commit atomically all updates, the translation table is written in a shadow copy. So the previous state of the translation table is preserve. If the failure occurs before the commit phase of the transaction, the original table is reloaded so current updates are discarded. If the failure occurs during the commit phase of the transaction, the MM reloads the latest persistent copy of the table which is not corrupted (i.e. all write operations of the table are completed or not before the failure). With EEPROM, the translation table stays in persistent memory. To map the shadow page of an update page, the translation table is updated in a shadow copy.

This algorithm is easy to implant but it has major drawbacks. Firstly, only one transaction is running at a time (i.e. batch processing model) because the translation table can not be shared between several simultaneous transactions. Secondly, with FlashRAM, the translation table takes a great part of the RAM so this reduces the cache performance.

Adaptation of the Before-Image Logging Algorithm

This algorithm does not need a persistent translation table. It uses only a small in-RAM table (12 bytes) for the needs of the caching mechanism (if it is enabled). The Before-Image Logging algorithm keeps the original data values in a distinct area of the non-volatile memory called Before-Image Log. The log is a file that contains log records. Before performing a writing operation, the original value is saved in a new log record. The log record contains also an incremental sequence number, the value identifier (i.e. the address of a page if whole pages are logging or the offset and the size in case of sub-page logging). If a failure occurs before and during the commit phase, the MM plays forward the log to replace the new (uncommitted) values by the original (committed) values saved in the log. With EEPROM technology, sub-page logging is possible (if the caching mechanism is enable). Sub-page logging consist in saving only the updated parts of the page in a log record. Indeed before each cache replacement, the updated page (which is in the RAM cache) is compared with the non-volatile page to detect updated words and to log these original corresponding values. The operation called "Diffing" is used for Client-Server OO-DBMS Recovery [8][15] With Flash-RAM technology, the Diffing allows only whole page logging since erasing operation deletes the state of a whole page before writing.

The Before-Image Logging presents several advantages. Firstly, no huge translation table is required. Therefore, cache size is maximal and this improves

performances by reducing the number of cache replacements. This reduces also the number of log records if the Diffing is used. Secondly, several transactions can access concurrently to the persistent memory. Thirdly, the MM can easily rollback over multiple checkpoints and can provide recovery mechanisms to implement extended transaction models [16] such as nested-transactions [17]. However, the two former features require a complex (for the smartcard) recycling (compression) of the log.

Implementation of a prototype

We have developed a demonstrator to evaluate the performances of these recovery algorithms (Shadow Pages and Before-Image Logging without Diffing) for two non-volatile memory (EEPROM and FlashRAM). These algorithms are benchmarked with several configuration of memory (varying size of RAM and varying size of persistent memory) and several application workloads. These benchmarks and these results are presented in this section. A part of the demonstrator was developed with the GemXpresso, the GEMPLUS's JavaCard [18] compliant smartcard.

Experimental Results

This section present first the methodology to benchmark the two algorithm (the shadow page algorithm vs the before image logging). To perform the benchmarks we run common GemXpresso example programs, and we calculate access time to the data. Thus we have a pool of scenario for persistent memory access. So we can define typical persistent memory access figures (address, size, sequence, frequency, ...). With this figure we choose a typical persistent memory access scenario (from the pool of our scenario). This scenario run on the GemXpresso card in 0,7 ms. At last we perform a fine evaluation with the typical scenario play on our demonstrator.

We couldn't implement the algorithm in a same way with each physical memory features. So we study our algorithm on two memory models : Flash technologies and traditional E²PROM. Each model produce different experimental results.

Shadow Page and Before Image Logging : memory Usage

To manage shadow paging we require two data structures : a Shadow Table and a Flash Allocation Table (as shown in *Adaptation of the Shadow Pages Algorithm*). The figure 1 shows the persistent memory space needed to store these data structures in different recoverable memory configurations.

Shadow16 column show the space used by data structures to implement Shadow Page Algorithm in a 64Kb memory with shadow page size of 16 bytes. Shadow32

show space used in memory for page size of 32 bytes and Shadow64 show space needs for page size of 64 bytes. At last we show a tiny data structure used for Shadow Page of 64 Bytes in a 16Kb Persistent Memory.

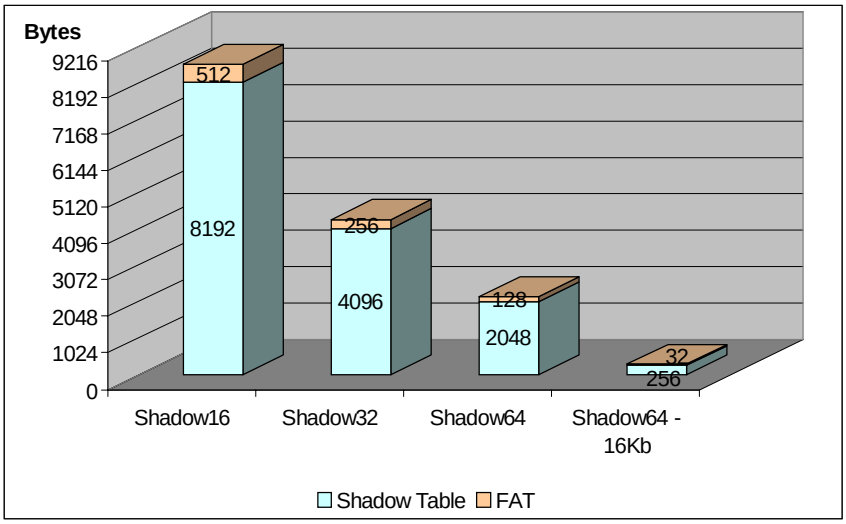

Figure 1 : Shadow Page algorithm initial needs in persistent memory.

In another hand, shadow page (an more precisely small shadow page) use less persistent memory at run time. The Figure 2 show the memory consumption when we play our preferred scenario (step by step).

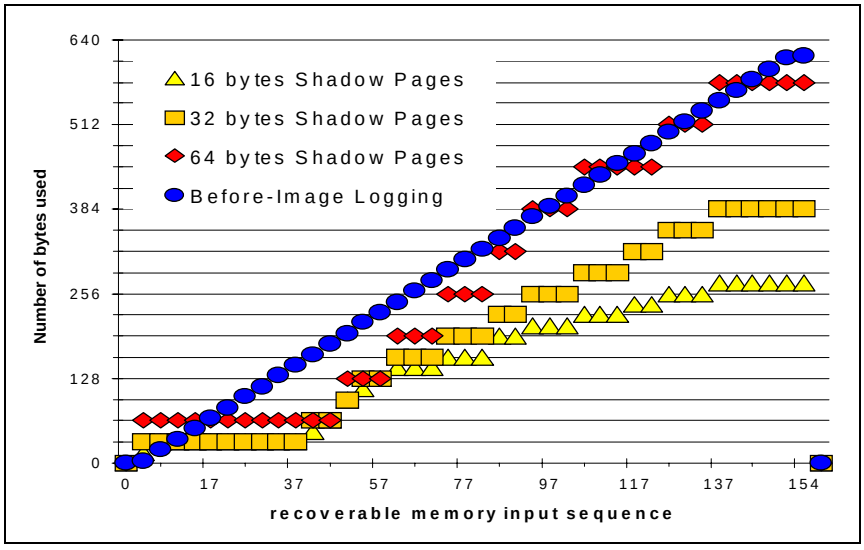

figure 2 : Memory used by recoverable memory algorithms

Theoretically shadow page algorithm needs less runtime memory than before image logging (logarithmic curve for the first, and linear for the second). but, in our preferred scenario, the number of writing operation is to limited to really make difference between shadow64 and BIL (before image logging).

Implementations on Flash Memory

Our implementation of the Shadow Page Algorithm is useful on Flash Memory because the physical writing operation are buffered in RAM. The RAM Buffer work as a "one page cache" for shadow page. Until a new page is needed or program exit it does not perform persistent writing. However, our demonstrator don't deal with general cache implementation. According cache management to JavaCard Virtual Machine Specification is another problem... Figure 3 shows the number of **`LineErase`** operation perform (in our demonstrator) by the shadow page algorithm in different pages size configurations. Due to the fact that a **`LineErase`** takes less time to be performed on a small physical page, this figure show the useful of small physical page and large shadow page (a factor 2 in number of LineErase for a factor 4 in memory page size, but **`lineErase`** time is not a linear function of page size).

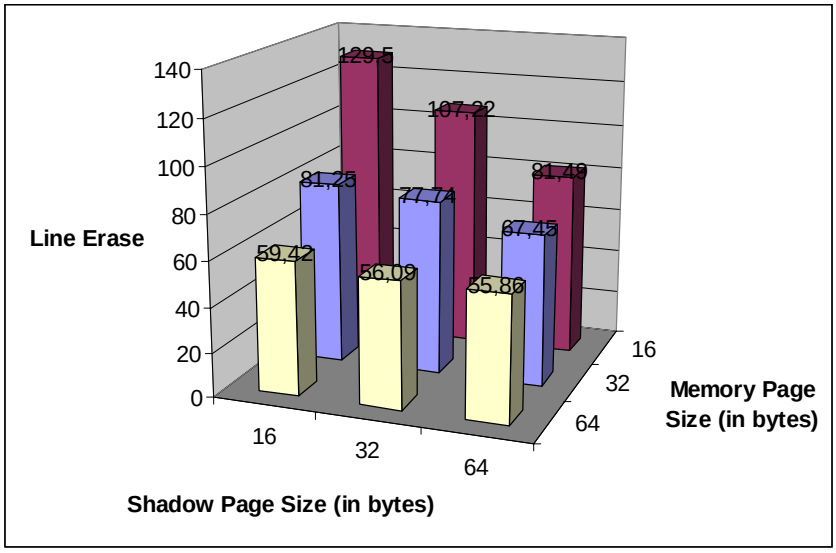

figure 3 : Number of LineErase operation for each memory configurations in our preferred scenario.

Before Image Logging breaks the pseudo-cache feature in Flash Memory Access because it write in two distinct area space the data and the BIL record. Furthermore,

according to the fact that Flash memory can be 'LineErased' and after a long time written, we implement the BIL without shadow page cache invalidation.

The figure 4 present time consumption in 3 different memories configuration of BIL algorithm and Shadow Page algorithm. BIL16, BIL32 and BIL64 is the same BIL algorithm implemented on Flash memory of 16 byte pages, BIL32 on 32 bytes pages and BIL64 on 64. This figure shows drawbacks of BIL algorithm in time consumption. BIL drawbacks are minimized with large pages size of the physical memory. But the main problem is the atomic physical page writing. Table 1,2 resume Experimental results.

		Shadow16	Shadow32	Shadow64	BIL
	16 bytes	114	61	48	265
	32 bytes	81	60	47	215
	64 bytes	52	57	46	196
Flash granularity	memory				

Table 1 : number of LineErase performed by the scenario ended by a commit.

		Shadow16	Shadow32	Shadow64	BIL
	16 bytes	107	66	51	369
	32 bytes	73	66	51	258
	64 bytes	64	56	51	230
Flash granularity	memory				

Table 2 : number of LineErase performed by the scenario ended by a rollback.

Implementations on E²PROM

the E²PROM doesn't need the use of a LineErase operation before writing data. Moreover, we the E²PROM, we can access to the date without using pages. In this case, the implementation of this algorithm can be easily performed.

However, the writing in a cellule can not be considered as an atomic action. In this case.

	Shadow16	Shadow32	Shadow64	BIL
WriteCell – end with commit	205	190	181	462
WriteCell end with rollback	188	178	172	772

Table 1 : number of LineErase performed by the scenario ended by a commit.

Conclusion and perspectives

In this paper, we first present the failures, which can make incoherent data stored in a smartcard. Then to recover these failures, we propose mechanisms based on the transactional model. We propose to implement Recoverable Persistent Memory in smartcards. RPM hide the difference between volatile (RAM) and non-volatile memories (FlashRAM, EEPROM, ...) and guaranty the atomicity and durability of a group of code lines. We adapt and implement the DBMS recovery algorithms to the smartcard memory context : the " Shadow Pages " one and the " Before-Image Logging ".

However, smartcards will be completely integrated when they will be able to participate to distributed transactions [3]. This assumes that ACID properties are guaranties by every node of the distributed transaction. The ACID properties in distributed transactions oblige the card to participate to a distributed validation protocol such as the two phases commit protocol specified by X/Open and OMG. In another hand, the smartcard can execute several transactions simultaneity. The Isolation property must be guaranty by the card. We work on concurrency control mechanisms such as the two-phase locking which provides a coherent sharing of data.

References

[1] V. Cordonnier, The future of SmartCards : Technology and Application , In proceedings IFIP World Conference on Mobile Communication, Camberra, September 1996.

[2] S. Lecomte, Didier Donsez, Intégration d'un Gestionnaire de Transaction dans les cartes à microprocesseur , Proceeding of NOTERE, Nouvelles Technologies de la Répartition, Pau, France, 4-6 Novembre 1997, pp. 347-362

[3] S. Lecomte, COST-STIC : Cartes Orientées Services Transactionnels et Systèmes Transactionnels Intégrant des Cartes, PhD Dissertation, University of Lille I, Villeneuve d'Ascq, France, November 1998.

[4] J. Gray, Andreas Reuter, Transaction Processing : Concepts and Technics, Morgan Kaufmann Publishers, 1993.

[5] International Standard Organisation (ISO), Identification cards, integrated circuit cards with contacts~: Part 4 inter-industry command for interchange, International Standard ISO/IEC 7816-4, 1995

[6] M. Satyanarayanan, H.H . Mashburn, P. Kumar, D.C. Steere, J.J. Kistler, Lightweight Recoverable Virtual Memory , Proc of 1993 Symposium on Operating System Principles, pp 146-160, December 1993.

[7] M. Atkinson, K. Chishlom, P. Cockshott, R. Marshall, Algorithms for a Persistent Heap , Software, Practice and Experience, 13(3) :259-271, March 1983.

[8] V. Singhal, S.V. Kaddak, P.R. Wilson, Texas: An Efficient, Portable Persistent Store, Proc. of the Fifth Intl Workshop on Persistent Object System, San Miniato, Italy, September 1992.

[9] S.J. White, D.J. DeWitt, QuickStore : A High Performance Mapped Object Store, Proc. of the 1994 ACM SIGMOD Conf. Minneapolis, Minnnesota, May 1994.

[10] P.A. Bernstein, V. Hadzilacos, N. Goodman, Concurrency Control and Recovery in Database Systems, Addison-Wesley, Reading , MA, 1987.

[11] G. P. Copeland, T. Keller, R. Krishnamurthy, M. Smith, The Case For Safe RAM, VLDB 1989, pp327-335

[12] W. Teck Ng, P. M. Chen, Integrating Reliable Memory in Databases, VLDB'97, Proceedings of 23th International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece. Morgan Kaufmann 1997, ISBN 1-55860-470-7, pp76-85.

[13] Belady, A study of Replacement Algorithms for Virtual Storage Computers, IBM Systems Journal 5, 1966, pp78-101.

[14] Esprit program EP8670, CASCADE: Operating System, European project ESP8670 , April 1995.

[15] S. J. White, D. J. DeWitt, Implementing Crash Recovery in QuickStore: A Performance Study, In proceedings of the 1995 ACM SIGMOD Conference, pp 187-198

[16] A.K. Elmagarmid, Database Transaction Models for Advanced Applications, Morgan Kaufmann Publishers, 1992

[17] J.E.B. Moss, Nested Transactions: An Approach to Reliable Distributed Computing, Boston, MIT Press, 1985.

[18] Sun Microsystems, JavaCard 2.0 Language subset and Virtual Machine Specification, Rev 1.0 final, October 1997.