Algebraic Signatures for Scalable Distributed Data Structures

Witold Litwin
CERIA, Université de Paris 9
Pl. du Mal. de Lattre, 75016 Paris, Fr.
WitoldLitwin@dauphine.fr

Thomas Schwarz
Comp. Engineering, Santa Clara Univ.
Santa Clara, CA 95053, USA
tjschwarz@scu.edu

Abstract

Signatures detect changes to data objects. Numerous schemes are used, e.g., the cryptographically secure standards SHA1 and MD5 used in Computer Forensics to distinguish file versions. We propose a novel signature descending from the Karp-Rabin signatures, which we call algebraic signatures because it uses Galois Field calculations. One major characteristic, new for any known signature scheme, is certain detection of small changes of parameterized size. More precisely, we detect for sure any change that does not exceed n-symbols for an n-symbol signature. For larger changes, the collision probability is typically negligible, as for the other known schemes. We apply the algebraic signatures to the Scalable Distributed Data Structures (SDDS). We filter at the SDDS client node the updates that do not actually change the records. We also manage the concurrent updates to data stored in the SDDS RAM buckets at the server nodes. We further use the scheme for the fast disk backup of these buckets. We sign our objects with 4-byte signatures, instead of 20-byte standard SHA-1 signatures that would be impractical for us. Our algebraic calculus is then also about twice as fast.

1. Introduction

A *signature* is a string of a few bytes intended to identify uniquely the contents of a data object (a record, a page, a file, etc.). Different signatures prove the inequality of the contents, while same signatures indicate their equality with overwhelmingly high probability. Signatures are a potentially useful tool to detect the updates or discrepancies among replicas (e.g. of files [Me83], [AA93], [BGMF88], [BL91], [FWA86], [Me91], [SBB90]). Their practical use requires further properties, since the updates often follow common patterns. For example, in a text document the cut-and-paste (switch) of a large string is a frequent operation. An update of a database record often changes only relatively few bytes.

Common updates should change the signatures, i.e., they should not lead to *collisions*. The collision probability, i.e. the probability that the old and the new value of a record have exactly the same signature, should be uniformly low for every possible update, although no schemes can guarantee that the signature changes for all possible updates.

In this paper, we propose algebraic signatures. An algebraic signature consists of n symbols, where a symbol is either a byte or a double byte, though theoretically symbols can be bit strings of any length f. There are other signatures, such as CRC signatures, the increasingly popular SHA1 standard of length 20B [N95], [KPS02], the 16B MD5 signature used to ascertain integrity of disk images in computer forensics, and Karp Rabin Fingerprints (KRF) [KR87], [B93], [S88], and its generalizations in [C97]. Other proposed signatures have additional "good" properties for specific applications e.g., for documents, against transmission errors, malicious alterations… [FC87], [KC96], [KC99].

We used algebraic signatures in the context of Scalable Distributed Data Structures (SDDS). A typical SDDS file implements a table of a relational database that consists of a large number of records with a unique key. A more generic SDDS file is a very large set of objects with a unique object identifier. Most known SDDS schemes implement a hash LH* or LH*RS file, or a range partitioned RP* file [LNS06], [LS01], [LNS94]. An implementation, the SDDS-2000 prototype system, is available for download [C02]. SDDS are intended for parallel or distributed databases on multicomputers, grid, or P2P systems [NDLR00], [NDLR01], [LRS02]. The SDDS-2000 files reside in distributed RAM buckets and its access times are currently up to a hundred times better than access to data on a disk.

We motivate our use of signatures in the SDDS context as follows. First, some applications of SDDS-2000 may need to back up the data in the buckets from time to time on disk. One needs then to find only the areas that changed in the bucket since the last backup. For reasons whose detail we give later, but essentially because we did not design SDDS-2000 for this need, we could not use the traditional "dirty bit" approach. Signatures were apparently the only workable approach.

Second, the signatures of record updates proved useful. On frequently observed occasions, an application requests a record update, but the new and the old record are in fact identical. We compare the signatures of the before and after image to detect this problem and avoid unnecessary traffic. If transactions follow the two-step model, we can prevent dirty reads by calculating the signatures of the read set between reading and just before committing the writes.

Our n-symbol algebraic signature is the concatenation of n power series of Galois Field (GF) symbols in GF(28) or GF(216). While our algebraic signatures are not cryptographically secure as is SHA1 or MD5, they exhibit a number of attractive properties. First, we may produce short signatures, sufficient for our goals, e.g., only 4B long. Next, the scheme is the first known, to the best of our knowledge, to guarantee that objects differing by n symbols have guaranteed different n-symbol signatures. Furthermore, the probability that a switch or any update leads to a collision is also sufficiently low, namely 2^{-s} with signature length in bits of s. We can also calculate the signature of an updated object from the signature of the update and from the previous signature. Finally, we may gather signatures in a tree of signatures, speeding up the localization of changes. Cryptographically secure signatures such as the 20B SHA-1 or the 16B MD5 do not fit our purpose well because they lack the last properties and because they are unnecessary large. Interestingly enough, they do not guarantee a change in signature for very small changes.

Our signatures are related to the Karp Rabin Fingerprints (KRF) [KR87] [B93] [S88] and its generalizations in [C97]. A 1-symbol algebraic signature corresponds to a KRF calculated in a Galois field and not through modular integer arithmetic. KRF and our signature share a nice property that allows us to calculate the signature of all substrings (of given length) in a larger string quickly. We can thus use both KRF and algebraic signatures for pattern matching in strings. However, KRF work with bits and ours with larger characters such as bytes. Concatenating KRF in order to obtain the equivalent of our n-symbol signature does not seem to lead to interesting algebraic properties such as freedom of collisions for almost identical pages.

Below, we first describe more in depth the needs of SDDS for a signature. Next, we recall basic properties of a GF. Afterwards, we present our approach, we overview the implementation, and we discuss the experimental results. We conclude with directions for the further work.

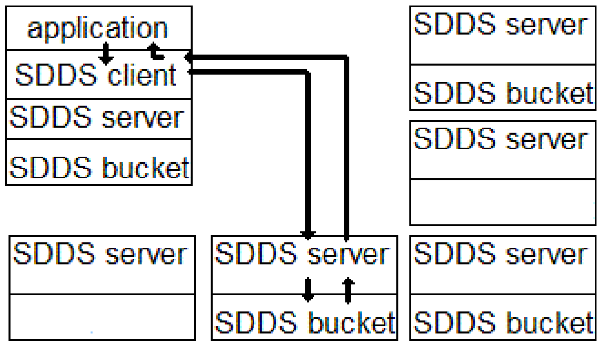

Figure 1: SDDS architecture with data flow for a simple key search

2. Signatures for an SDDS

A Scalable Distributed Data Structure (SDDS) uses the distributed memory of a multicomputer to store a file. The file itself consists of records or more generally, objects. Records or objects consist of a unique key and a non-key portion. They reside in *buckets* in the distributed RAM of the multicomputer. The data structure implements the key-based operations of inserts, deletes, and updates as well as scan queries. The application requests these operations from an *SDDS client* that is located at the same node of the multicomputer. The client manages the query delivery through the network to the appropriate server(s) and receives the replies if any. (Fig. 1). As the file grows with inserts, bucket splits generate more buckets. Each split sends about half of a bucket to a newly created bucket. Recall that buckets reside always in main memory of the nodes of the multicomputer. We originally designed our signatures to help with bucket back-up and to streamline record updates, but other applications emerged, which we present in the conclusion.

2.1. File back-up

We wish to backup an SDDS bucket B on disk. Since a typical bucket contains many MB worth of data, we only want to move those parts of the bucket that differ from the current disk copy. The traditional approach is to divide the buckets into pages of reasonably small granularity and maintain a dirty bit for each page. We reset the dirty bit when the page goes to disk and set it when we change the page in RAM. For a backup, we then only move the clean pages, i.e. those with reset dirty bit. The implementation of this approach in our running prototype SDDS-2000 [C02] would demand refitting a large part of the existing code that was not designed for this capacity. As is often the case, this appeared to be an impossible task. The existing code is a large piece of software that writes to the buckets in many places. Different students, who left the team since, produced over the years different related functional parts of the code.

Our alternative approach calculates a signature for each page when data should move to the disk. This computation is independent of the history of the bucket page and does not interfere with the existing maintenance of a data structure. This is the crucial advantage in our context.

More in detail, we provide the disk copy of the bucket with a *signature map*, which is simply the collection of all its page signatures. Before we move a page to disk, we recalculate its signature. If the signature is identical to the entry in the signature map, we do not write the page.

The slicing of the buckets into pages is somewhat arbitrary. The signature map should fit entirely into RAM (or even into the L2 cache into which we can load it with the *prefetch* macro). Smaller pages minimize transfer sizes, but increase the map and the overall signature calculus overhead. We expect the practical page size to be somewhere between 512B and 64KB. The best choice depends on the application. In any case, the speed of the signature calculus is *the* challenge, as it has to be small with respect to the disk write time. Another challenge is the practical absence of the collisions to avoid an update loss. The ideal case is a zero probability of a collision, but in practice, a probability at the order of magnitude of that of irrecoverable disk errors (e.g. writes to an adjacent track) or software failures is sufficient. The database community does not bother dealing with those unlikely failures anyway so that the equally small possibility of a collision does not change fundamentally the situation.

Currently, we implement the signature map simply as a table, since it fits into RAM. The algebraic signature scheme allows also structuring the map into a *signature tree*. In a tree, we compute the signature at the node from the signatures of all descendents of the node. This speeds up the identification of the portions of the map where the signatures have changed (similar to [Me83] et al.) More on it in Section 4.1.

2.2. Record Updates

An SDDS update operation only manipulates the non-key part of a record R. Let R_b denote the *before-image* of R and S_b its signature. The before-image is the content of record R, subject to the update by a client. The result of the update of R is the *after-image* which we call R_a and its signature S_a. The update is *normal* if R_a depends on R_b, e.g., `Salary := Salary + 0.01*Sales`. The update is *blind* if R_a is set independently of R_b, e.g., if we request `Salary := 1000` or if a house surveillance camera updates the stored image. The application needs R_b for a normal update but not always for a blind one. In both cases, it is often not aware whether the actual result is effectively $R_a \neq R_b$. As in the above examples for unlucky salespersons in these hard times, or as long as there is no burglar in the house.

Typically, the application nevertheless requests the update from the data management system that typically executes it. This "trustworthy" policy, i.e., if there is an update request, then there is a data change, characterizes all the DBMSs we are aware of. This is indeed surprising, since the policy can often cost a lot. Tough times can leave thousands of salespersons with no sales, leading to useless transfers between clients and servers and to the useless processing on both nodes of thousands of records. Likewise, a security camera image is often a clip or movie of several Mbytes, leading to an equally futile effort.

Furthermore, on the server side, several clients may attempt to read or update concurrently the same SDDS record R. It is best to let every client read any record without any wait. The subsequent updates should not however override each other. Our approach to this classical constraint is freely inspired by the *optimistic* option of the concurrency control of MS-Access, which is not the traditional one in the database literature, e.g. [LBK02].

The signatures for SDDS updates are useful in this context as follows. The application that needs R_b for a normal update, requests from the client a search operation for key R and eventually receives the record R_b. When the application has finished its update calculation, the application returns to the client R_b and R_a. The client computes S_a and S_b. If $S_a = S_b$, then the update actually did not change the record. Such updates terminate at the client. The client sends R_a and S_b to the server only if $S_a \neq S_b$. The server accesses R and computes its signature S. If $S_b = S$, then the server updates R to $R := R_a$. Otherwise, it abandons the update because of a conflict. A concurrent update must have happened to R in the time since the client read R_b and the server received its update R_a. If the new update proceeded, it would override the intervening one, violating serializability. The server notifies the client about the rollback, which in turn alerts the application. The application may read R again and redo the update.

For a blind update, the application provides only R_a to the client. The client computes S_a and sends the key of R_a to the server requesting S. The server computes S and sends it to the client as S_b. From this point on, the client and the server can proceed as in a normal update. Calculating and sending S alone as S_b already avoids the transfer of R_b to the client. It may also avoid the useless transfer of R_a to the server. These can be substantial savings, e.g., for the surveillance images.

The scheme does not need locks. In addition, as we have seen, the signature calculus saves useless record transfers. Besides, neither the key search, nor the insert or deletion operations need the signature calculus. Hence, none of these operations incurs the concurrency management overhead. All together, the degree of concurrency can be high. The scheme roughly corresponds to the R-Committed isolation level of the SQL3 standard. Its properties make it attractive to many applications that do not need transaction management, especially, if searching is the dominant operation, as is the case in general for an optimistic scheme.

The scheme does not store the signatures. Interestingly, the storage overhead can be zero. This is not possible for timestamps, probably the approach of MS Access, although that overhead is usually negligible, and hence perfectly acceptable in practice. In fact, it can still be advantageous to vary the signature scheme by storing the signature with the record. As we show later, the storage cost at the server is typically negligible, of only about 4B per signature. The client sends in this case also S_a to the server which stores it in the file with R_a if it accepts the update. When the client requests R it gets it with S. If the client requests S alone, the server simply extracts S from R, instead of dynamically calculating it. All together, one saves the S_b calculus at the client and that of S at the server. Also, and perhaps more significantly in practice, the signature calculus *happens only at the client*. Hence, it is entirely parallel among the concurrent clients. This can enhance the update throughput even further.

Whether we store the signature or not, the main challenge remains the speed of the signature. Since a SDDS key search or a SDDS reaches currently speeds of 0.1 ms, the time to calculate record signatures cannot be longer than dozens of microseconds. Another challenge is the zero or practically zero probability of collisions to prevent update losses.

2.3. Searches

The frequent SDDS scan operation looks for all records that contain a string somewhere in a non-key field. If the SDDS contains many server nodes, if the search string is long, and if there are few hits, then we can use the widely used distributed, Las Vegas pattern-matching algorithm based on Karp-Rabin fingerprints [KR87], but with variations stemming from the use of our signatures. In more detail, the client does not send on the search string, but rather its length and signature. The SDDS servers receive the signature and then calculate individually the signatures of all the substrings of the correct length in their collection of records. They send back all records with such a string. While our signature differs from the ones that Karp and Rabin use, the difference is small enough so that they retain the property to evaluate quickly all substrings of a given length in a larger string. The large number of substrings of a given length in an SDDS file virtually guarantees collisions, but these false positives are not dangerous since the client evaluates the strings returned by the servers.

3. Galois Fields

A Galois field (GF) is finite field. Addition and multiplication in a GF are associative, commutative, and distributive. There are neutral elements called zero and one for addition and multiplication respectively, and there exist inverse elements regarding addition and multiplication. We denote GF(2^f) a GF over the set of all binary strings of a certain length f. We only use GF (2^8) and GF (2^{16}). Their elements are respectively one-byte and two-byte strings.

We identify each binary string with a binary polynomial in one formal unknown x. For example, we identify the string 101001 with the polynomial x^5+x^3+1. We further associate with the GF the *generator polynomial g(x)*. This is a polynomial of degree f that cannot be written as a product of two other polynomials other than the trivial products $1\cdot f$ or $f\cdot 1$. The addition of two elements in our GF is that of their binary polynomials. This means that the sum of two strings is the XOR of the strings. When we use the "+" sign between two GF elements, it always refers to the exclusive or and never to an integer addition. The product of two elements is the binary polynomial obtained by multiplying the two operand polynomials and taking the remainder modulo *g(x)*.

There are several ways to implement this calculus. We use *logarithms,* based on *primitive* elements of a GF, all of which we define below. The *order* ord (α) of a non-zero element α of a Galois field is the smallest non-zero exponent i such that $\alpha^i = 1$. All non-zero elements in a GF have a finite order. An element $\alpha \neq 0$ of a GF of size s is *primitive*, if $\text{ord}(\alpha) = s$-1. It is well known that for any given primitive element α in a Galois field with s elements, all the non-zero elements in the field are different powers α^i, each with a uniquely determined exponent i, $0 \leq i \leq s$-1. All GF have primitive elements. In particular, any α^i is also a primitive element if i and s-1 are *coprime,* i.e., without non-trivial factors in common. Our GFs contain 2^f elements, hence the prime decomposition of 2^f-1 does not contain the prime 2. For our basic values of f = 8 or16, 2^f-1 has only few factors, hence there are relatively many primitive elements. For example, for f=8 we count 127 primitive elements or roughly half the elements in the GF.

We define logarithms with respect to a given primitive element α. Every non-zero element β is a power of α. If $\beta = \alpha^i$, we call i the logarithm of β with respect to α and write $i = \log_\alpha(\beta)$ and we call β the antilogarithm of i with respect to α and write $\beta = \text{antilog}_\alpha(i)$. The logarithms are uniquely determined if we choose i to be $0 \leq i \leq 2^f$-2. We set $\log_\alpha(0) = -\infty$.

We multiply two elements β and γ by the following formula which uses addition modulo 2^f-1:

$$\beta \cdot \gamma = \text{antilog}_\alpha(\log_\alpha(\beta) + \log_\alpha(\gamma)).$$

To implement this formula, we create one table for logarithms of size 2^f symbols. We also create another one for antilogarithms of size $2^f \cdot 2$. That table has two copies of the basic antilog table. It accommodates indices up to size $2^f \cdot 2$ and avoids the slower modulo calculus of the formula. For our choices of f, both tables should fit into the cache of most current processors (not all for f = 16). We also check for the special case of one of the operands being equal to 0. All together, we obtain the following simple C-pseudo-code:

```
GFElement mult(GFElement left,GFElement
right) {
        if(left==0 || right==0) return 0;
        return
antilog[log[left]+log[right]];
}
```

In terms of Assembly language instructions, the typical execution costs of a multiplication are two comparisons, four additions (three for table-look-up), three memory fetches and the return statement.

4. Algebraic Signatures

4.1. Basic properties

We call *page P* a string of l symbols p_i ; $i = 0..l$-1. In our case, the symbols p_i are bytes or 2-byte words. The symbols are elements of a Galois field, GF (2^f) (f = 8, 16). We assume that $l < 2^f$-1.

Let $\boldsymbol{\alpha} = (\alpha_1 \ldots \alpha_n)$ be a vector of different non-zero elements of the Galois field. We call $\boldsymbol{\alpha}$ the *n-symbol signature base*, or simply the *base*. The (*n-symbol*) *P signature* or, simply, *P signature*, based on $\boldsymbol{\alpha}$, is the vector

$$\text{sig}_{\boldsymbol{\alpha}}(P) = (\text{sig}_{\alpha_1}(P), \text{sig}_{\alpha_2}(P), \ldots, \text{sig}_{\alpha_n}(P))$$

where for each α we set

$$\text{sig}_\alpha(P) = \sum_{i=0}^{l-1} p_i \alpha^i \ .$$

We call each coordinate of $\text{sig}_{\boldsymbol{\alpha}}$ the *component signature*.

The n-symbol signature has some interesting properties that depend on the choice of the coordinates. We use primarily the following one where the coordinates are consecutive powers of the same primitive element α.

$\boldsymbol{\alpha} = (\alpha, \alpha^2, \alpha^3 \ldots \alpha^n)$ with $n \ll \text{ord}(a) = 2^f - 1$.

In this case, we denote $\text{sig}_{\boldsymbol{\alpha}}$ as $\text{sig}_{\alpha,n}$. Clearly, the collision probability of $\text{sig}_{\alpha,n}$ can be at best 2^{-nf}. If n = 1, this is probably insufficient. We also experimented with a different n-symbol signature $\text{sig}'_{\alpha,n}$ where <u>all</u> coordinates of $\boldsymbol{\alpha}$ are primitive:

$\boldsymbol{\alpha} = (\alpha, \alpha^2, \alpha^4, \alpha^8 \ldots \alpha^{2n})$.

Since α is primitive and since powers of 2 have no common factor with 2^f-1, all coordinates of $\text{sig}'_{\alpha,n}$ are primitive. As we will see when we discuss cut-and-paste operations, $\text{sig}'_{\alpha,n}$ has different properties. The basic new property of the $\text{sig}_{\alpha,n}$ signature is that any change of up to n symbols within P changes the signature **for sure**. This is our primary rationale in this scheme. More formally, we stay this property as follows.

Proposition 1: Provided the page length l is $l < \text{ord}(\alpha) = 2^f - 1$, $\text{sig}_{\alpha,n}$ signature discovers any change of up to n symbols per page.

Proof: As α is primitive and our GF is GF (2^f) we have $\text{ord}(\alpha) = 2^f - 1$. Assume that the file symbols at locations i_1, i_2, … i_n has been changed, but that the signatures of the original and the altered file are the same. Call d_ν the difference between the respective symbols in position i_ν. The difference of the component signatures is then:

$$\sum_{\nu=1}^{n} \alpha^{i_\nu} d_\nu = 0 \ , \ \sum_{\nu=1}^{n} \alpha^{2i_\nu} d_\nu = 0 \ , \quad ... \sum_{\nu=1}^{n} \alpha^{ni_\nu} d_\nu = 0 \cdot$$

The d_ν values are the solutions of a homogeneous linear system:

$$\begin{pmatrix} \alpha^{i_1} & \alpha^{i_2} & \alpha^{i_3} & \alpha^{i_4} & \cdots & \alpha^{i_n} \\ (\alpha^{i_1})^2 & (\alpha^{i_2})^2 & (\alpha^{i_3})^2 & (\alpha^{i_4})^2 & \cdots & (\alpha^{i_n})^2 \\ (\alpha^{i_1})^3 & (\alpha^{i_2})^3 & (\alpha^{i_3})^3 & (\alpha^{i_4})^3 & \cdots & (\alpha^{i_n})^3 \\ (\alpha^{i_1})^4 & (\alpha^{i_2})^4 & (\alpha^{i_3})^4 & (\alpha^{i_4})^4 & \cdots & (\alpha^{i_n})^4 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ (\alpha^{i_1})^n & (\alpha^{i_2})^n & (\alpha^{i_3})^n & (\alpha^{i_4})^n & \cdots & (\alpha^{i_n})^n \end{pmatrix} \cdot \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ \vdots \\ d_n \end{pmatrix} = 0.$$

The coefficients in the first row are all different, since the exponents i_ν < ord(α). The matrix is of Vandermonde type, hence invertible. The vector of differences $(d_1,d_2...d_n)^t$ is thus the zero vector. This contradicts our assumption. Thus $\text{sig}_{\alpha,n}$ signature detects any up to n-symbol change. **qed**

Notice that Proposition 1 trivially holds for $\text{sig}'_{\alpha,n}$ with $n \le 2$. More generally, it proves best possible behavior of $\text{sig}_{\alpha,n}$ scheme for changes limited to n symbols. An application can however possibly change up to $l > n$ symbols. We now prove that the $\text{sig}_{\alpha,n}$ scheme still exhibits the low collision probability that we need in a signature scheme.

Proposition 2: Assume page length l < ord(α) and that every possible page content is equally likely. Then the signatures $\text{sig}_{\alpha,n}$ of two different pages collide (coincide) with probability of 2^{-nf}.

Proof: The n-symbol signature is a linear mapping between the vector spaces $GF(2^f)^l$ and $GF(2^f)^n$. This mapping is an epimorphism, i.e., every element in $GF(2^f)^n$ is the signature of some page, an element of $GF(2^f)^l$. Consider the map ϕ, which maps every page with all but the first n elements equal to zero to its signature. Thus, ϕ: $GF(2^f)^n \rightarrow GF(2^f)^l$, $(x_1,\ldots,x_n) \rightarrow \text{sig}_{\alpha,n}((x_1,\ldots,x_n,0,\ldots0))$, and:

$$\varphi((x_1,x_2,...,x_n)) = \begin{pmatrix} \alpha & \alpha^2 & \alpha^3 & \alpha^4 & \cdots & \alpha^n \\ \alpha^2 & (\alpha^2)^2 & (\alpha^3)^2 & (\alpha^4)^2 & \cdots & (\alpha^n)^2 \\ \alpha^3 & (\alpha^2)^3 & (\alpha^3)^3 & (\alpha^4)^3 & \cdots & (\alpha^n)^3 \\ \alpha^4 & (\alpha^2)^4 & (\alpha^3)^4 & (\alpha^4)^4 & \cdots & (\alpha^n)^4 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \alpha^n & (\alpha^2)^n & (\alpha^3)^n & (\alpha^4)^n & \cdots & (\alpha^n)^n \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_n \end{pmatrix}.$$

The matrix is again of Vandermonde type, hence is invertible. This implies that every possible vector in $GF(2^f)^n$ is the signature of a page with all but the first n symbols equal to zero, and of only one such page. Consider now an arbitrary vector ***s*** in $GF(2^f)^n$. Each page of form $(0,\ldots,0,x_{n+1},x_{n+2},\ldots, x_l)$ has some vector ***t*** in $GF(2^f)^n$ as its signature. For any ***s, t*** there is then exactly one page $(x_1,\ldots,x_n,0,\ldots0)$ that has ***s-t*** as the signature. The page $(x_1,\ldots,x_n, x_{n+1}, x_{n+2},\ldots, x_l)$ has therefore signature ***s***. Thus, the number of pages that have signature ***s*** is that of all pages of form $(0,\ldots,0,x_{n+1},x_{n+2},\ldots, x_l)$. There are $2^{f(l-n)}$ such pages. There are furthermore 2^{fl} pages in total. A random choice of two pages leads thus to the same signature ***s*** with probability $2^{f(l-n)} / 2^{fl}$, which is 2^{-fn}. Assuming the selections of all possible pages to be equally likely, our proposition follows. **qed**

Notice that Proposition 2 also characterizes $\text{sig}^2_{\alpha,n}$ for $n \le 2$. We called our scheme "algebraic" because it has interesting algebraic properties. We now turn to state and prove the more important ones, starting with one that shows how our algebraic signature behaves when we change the page slightly. In short, the signature of the changed page is the old signature changed by the signature of the change. Slight changes are common in databases, where a typical attribute consists only of a few symbols. Proposition 3 below then allows us to calculate quickly the new signature of the record without having to scan in the complete record. Another application, which we use in [XMLBLS03], checks whether a bunch of updates to a record actually took place. We do this by calculating the signature of the record before the update. We then calculate the signature of the record after the changes first based on Proposition 3 and then by rescanning the record. If the two signatures coincide, then we trust that all updates have been performed correctly. For example, we see the blocks in a RAID Level 5 scheme as records (of length 512B or a multiple of that value, depending on the file system.) We maintain a log of all block changes. A daemon removes old entries in the log when they are no longer needed for recovery. This daemon uses Proposition 3 to check that all updates in the log – whether about to be removed or not – have been performed. This scheme amounts to a hybrid between a journaling file system and a classical system.

Proposition 3: Let us change $P = (p_0, p_1, \ldots p_{l-1})$ to page P' where we replace the symbols starting in position r and ending with position s-1 with the string $q_r, q_{r+1}, \cdots, q_{s-1}$. We define *Δ-string* as $\Delta = (\delta_0, \delta_1, \ldots,$

δ_{s-r-1}) with $\delta_i = p_{r+i} - q_{r+i}$. Then for each α in our base **α** we have:

$$\mathrm{sig}_\alpha(P') = \mathrm{sig}_\alpha(P) + \alpha^r \mathrm{sig}_\alpha(\Delta).$$

Proof: The difference between the signatures is:

$$\begin{aligned}\mathrm{sig}_\alpha(P') - \mathrm{sig}_\alpha(P) &= \sum_{i=r}^{s-1}(q_i - p_i)\alpha^i \\ &= \alpha^r(\sum_{i=r}^{s-1}(q_i - p_i)\alpha^{i-r}) \\ &= \alpha^r(\sum_{i=r}^{s-1}\delta_{i-r}\alpha^{i-r}) \\ &= \alpha^r(\sum_{i=0}^{s-r-1}\delta_i\alpha^i) \\ &= \alpha^r\mathrm{sig}_\alpha(\Delta).\end{aligned}$$

qed

As the final property, we present the behavior of algebraic signatures when a cut-and-paste operation changes the page. Proposition 1 states that the algebraic signature detects the change it if we move a string of length up to $\lfloor n/2 \rfloor$. This is of course very limiting. Proposition 2 only gives an error probability. The base $\boldsymbol{\alpha} = \alpha_0, \alpha_1, \ldots \alpha_{n-1}$, $0 \le i \le n-1$, where every α_i is primitive, has the largest possible ord (α_i) for each α_i. The $\mathrm{sig}'_{\alpha,n}$ scheme appears intuitively preferable in this context to $\mathrm{sig}_{\alpha,n}$ and the following proposition confirms the conjecture formally.

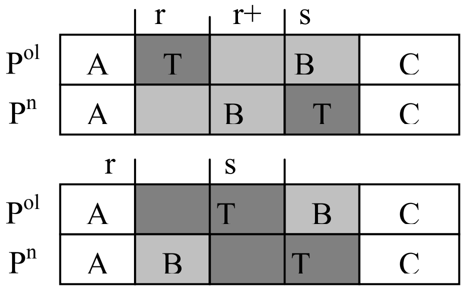

Figure 2: Illustration of the cut and paste operation.

Proposition 4: Assume an arbitrary page P and three indices r, s, t of appropriate sizes, Figure 2. We use a signature $\mathrm{sig}_{\boldsymbol{\alpha}}$ where base $\boldsymbol{\alpha} = \alpha_0, \alpha_1, \ldots \alpha_{n-1}$ has every ord(α_i) above the length of P, $0 \le i \le n-1$. Cut a string T of length t beginning with position r and move it into position s in P. If all T are equally likely, the probability that $\mathrm{sig}_{\boldsymbol{\alpha}}(P)$ changes is 2^{-nf}.

Proof: We denote B the reminder $B = P\text{-}T$. B contains at least n symbols or T contains at least n symbols, Figure 2. We only treat the case of length(B) $\ge n$, the other one is analogue. Without loss of generality, we assume a forward move of T within the file from position r to position s. A backward move just undoes this operation and thus has the same effect on the signature. Figure 2 defines names for the regions of the block and makes a spurious case distinction depending on whether $r+t < s$ or not. For any $\alpha \in \{\alpha_0, \alpha_1 \ldots \alpha_{n-1}\}$, the α signature of the "before" page (the top scheme for both situations) is

$$\begin{aligned}&\mathrm{sig}_\alpha(P^{\text{old}}) = \\ &\mathrm{sig}_\alpha(A) + \alpha^r\mathrm{sig}_\alpha(T) + \alpha^{r+t}\mathrm{sig}_\alpha(B) + \alpha^{s+t}\mathrm{sig}_\alpha(C)\end{aligned}.$$

The after page signature is

$$\begin{aligned}&\mathrm{sig}_\alpha(P^{\text{new}}) = \\ &\mathrm{sig}_\alpha(A) + \alpha^r\mathrm{sig}_\alpha(B) + \alpha^s\mathrm{sig}_\alpha(T) + \alpha^{s+t}\mathrm{sig}_\alpha(C).\end{aligned}$$

The difference of the two signatures is:

$$\begin{aligned}&\mathrm{sig}_\alpha(P^{\text{new}}) - \mathrm{sig}_\alpha(P^{\text{old}}) = \\ &\alpha^r\mathrm{sig}_\alpha(T) + \alpha^{r+t}\mathrm{sig}_\alpha(B) + \alpha^r\mathrm{sig}_\alpha(B) + \alpha^s\mathrm{sig}_\alpha(T) = \\ &(\alpha^r + \alpha^s)\mathrm{sig}_\alpha(T) + (\alpha^r + \alpha^{r+t})\mathrm{sig}_\alpha(B) = \\ &\alpha^r\left((1+\alpha^s)\mathrm{sig}_\alpha(T) + (1+\alpha^t)\mathrm{sig}_\alpha(B)\right).\end{aligned}$$

Since our addition is the bitwise XOR, the negative is the same as the positive and no negative sign is missing in this expression. The expression is zero only if the right hand side, or the following expression, where we use γ_i as an abbreviation, is zero:

$$\begin{aligned}&(1+\alpha_i^s)(1+\alpha_i^t)^{-1}\mathrm{sig}_{\alpha_i}(T) + \mathrm{sig}_{\alpha_i}(B) \\ &= (1+\alpha_i^s)(1+\alpha_i^t)^{-1}\mathrm{sig}_{\alpha_i}(T) + \sum_{\nu=n}^{\text{size}(B)\text{-}1}\alpha_i^\nu b_\nu + \sum_{\nu=0}^{n-1}\alpha_i^\nu b_\nu \\ &= \gamma_i + \sum_{\nu=0}^{n-1}\alpha_i^\nu b_\nu.\end{aligned}$$

We now fix the whole situation with the exception of the first n symbols in B. The change in signature is:

$$\begin{aligned}&\left(\gamma_0 + \sum_{\nu=0}^{n-1}\alpha_0^\nu b_\nu, \gamma_1 + \sum_{\nu=0}^{n-1}\alpha_1^\nu b_\nu, \ldots, \gamma_{n-1} + \sum_{\nu=0}^{n-1}\alpha_{n-1}^\nu b_\nu\right) = \\ &(\gamma_0, \gamma_1, \ldots, \gamma_{n-1}) + \left(\sum_{\nu=0}^{n-1}\alpha_0^\nu b_\nu, \sum_{\nu=0}^{n-1}\alpha_1^\nu b_\nu, \ldots, \sum_{\nu=0}^{n-1}\alpha_{n-1}^\nu b_\nu\right).\end{aligned}$$

which is zero if and only if:

$$\begin{aligned}&\left(\sum_{\nu=0}^{n-1}\alpha_0^\nu b_\nu, \sum_{\nu=0}^{n-1}\alpha_1^\nu b_\nu, \ldots, \sum_{\nu=0}^{n-1}\alpha_{n-1}^\nu b_\nu\right) \\ &\qquad = (\gamma_0, \gamma_1, \ldots, \gamma_{n-1}).\end{aligned}$$

The left hand side is a linear mapping in the (b_0, b_1, … b_{n-1}), which has a matrix that is invertible, because it has a Vandermonde type determinant. Therefore, there exists only one combination (b_0, b_1, …, b_{n-1}) that is mapped by the mapping onto the right hand vector. This combination will be attained for a randomly picked B with probability 2^{-nf}. **qed**

To obtain the strongest property of a $\text{sig}_{\alpha,n}$ signature schema, one should thus use $\boldsymbol{\alpha}$ whose α_i have the largest order. The natural choice are primitive α_i. Assuming the need for $n > 2$, this is precisely the rationale in $\text{sig}^2_{\alpha,n}$. Notice that for GF (2^{16}), the collision probability is already small enough in practice, we discuss this in detail in Section 5.2.

At this stage of our research, the choice of $\text{sig}'_{\alpha,n}$ appears only as a trade-off between smaller probability of collision for possibly frequent updates (switches here), and the zero probability of collision for updates up to any n symbols. We are able only to conjecture that there is α in GF(2^8) or GF(2^8) for which Proposition 1 and 2 holds for $\text{sig}'_{\alpha,n}$ with $n > 2$. We did not pursue the investigation further. For our needs, $n = 2$ for GF (2^{16}) was sufficient (Section 5.2). As $\text{sig}'_{\alpha,2} = \text{sig}_{\alpha,2}$, the properties of both schemes coincide anyway.

4.2. Compound Algebraic Signatures

Our signature schemes keep the property of sure detection of n-symbol change as long as the page size in symbols is at most $2^f - 2$. For $f = 16$, the limit on the page size is almost 128 KB. Such granularity suffices for our purpose. There might be many pages in an SDDS bucket that can reach easily 256 MB for SDDS-2000. We can view the collection of all the signatures in a bucket as a vector. We call this vector the *compound* signature (of the bucket). More generally, we qualify a compound signature of m pages, as m-fold. The signature map of Section 2.1 implements a compound signature.

The practical interest of the compound signatures stretches beyond our motivating cases. Assume that we have a large string A. Proposition 1 guarantees only that we find small changes through our signature if A does not contain more than 2^f-1 characters. To apply Proposition 1 nevertheless, we break A into m pages of length smaller than ord(α)–1, $\alpha \in \boldsymbol{\alpha}$. The resulting m-fold compound signature then allows us to find any change involving up to n characters in any of the pages **for sure.**

If we have many pages (m>>1), we can use the following Proposition 5 to speed up the comparison of compound signatures by calculating the signature of a group of contiguous pages. We can do the same to a number of these "super-signatures" to obtain higher-level signatures. We can then organize these higher signatures in a tree structure. We always calculate algebraically the higher-level signature in a parent node from all the lower-level signatures in the children nodes. If a signature of a page changes, then all signatures on a path from a leaf to the node change (Fig. 3). This capability of compound signatures can be of obvious interest to our SDDS file backup application.

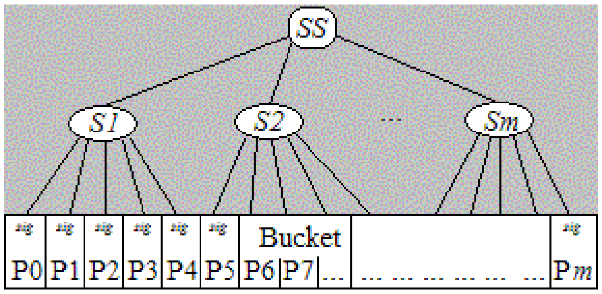

Figure 3: Signature tree with 3 levels of signatures.

The following proposition states the algebraic properties for only two pages of possibly different sizes. It can be easily generalized to more pages.

Proposition 5: Consider that we concatenate two pages P_1 and P_2 of length l and m, $l + m \leq 2^f$-1, into page (area) denoted $P_1|P_2$. Then, the signature $\text{sig}_{\alpha,n}$ ($P_1|P_2$) is as follows, where sig denotes $\text{sig}_{\alpha,n}$:

$$\text{sig}_{\alpha,n}(P_1|P_2) = (\text{sig}(P_1)+a^l\cdot \text{sig}(P_2),\ \text{sig}(P_1)+a^{2l}\cdot \text{sig}(P_2),\ldots,\ \text{sig}(P_1)+a^{nl}\cdot \text{sig}(P_2)).$$

Proof: The proof consists in applying the following lemma to each coordinate α of **α**.

qed

Lemma. We note $\text{sig}_{\alpha,1}$ as sig_α. Then, the 1-symbol $\text{sig}_\alpha(P_1|P_2)$ signature is

$$\text{sig}_\alpha(P_1|P_2) = \text{sig}_\alpha(P_1)+a^l\,\text{sig}_\alpha(P_2).$$

Proof: Assume that $P_1 = \{s_1,s_2,\ldots,s_l\}$ and $P_2 = \{s_{l+1}, s_{l+2}, \ldots, s_{l+m}\}$. Then

$$\begin{aligned}
&\text{sig}_\alpha(P_1|P_2)\\
&=\sum_{\nu=1}^{l+m} s_\nu\alpha^\nu\\
&=\sum_{\nu=1}^{l} s_\nu\alpha^\nu + \sum_{\nu=l+1}^{l+m} s_\nu\alpha^\nu\\
&=\sum_{\nu=1}^{l} s_\nu\alpha^\nu + \alpha^l\sum_{\nu=1}^{m} s_{\nu+l}\alpha^\nu\\
&=\text{sig}_\alpha(P_1)+\alpha^l\text{sig}_\alpha(P_2).
\end{aligned}$$

qed

Proposition 5 holds analogously for $\text{sig}^2_{\alpha,n}$. Together, all the propositions we have formulated prove the potential of our two schemes. We are currently investigating further algebraic properties.

4.3. Reinterpretation of Symbols

The following *meta-proposition* turns out to be important when we tune the speed of the signature calculation.

Proposition 6: Let φ be a function that maps the space of all symbols into itself and is one-to-one. (With other words, φ is a bijection of GF(2^f).) Define the *twisted signature*

$$\mathrm{sig}_{\varphi,\alpha}(P) = \sum_{i=0}^{l-1} \varphi(p_i)\alpha^i$$

and similarly

$$\mathrm{sig}_{\phi,\boldsymbol{\alpha}}(P) = (\mathrm{sig}_{\varphi,\alpha_1}(P), \mathrm{sig}_{\varphi,\alpha_2}(P), ..., \mathrm{sig}_{\varphi,\alpha_n}(P))$$

Then Propositions 1 to 5 also apply *mutatis mutandis* to the twisted signatures.

The proof is simply by inspection.

5. Experimental Implementation

5.1. Speeding up Galois Field arithmetic

We can tune the signature calculation. First, according to Proposition 6, we can interpret the page symbols directly as logarithms. This saves a table look-up. The logarithms range from 0 to 2^f-2 (inclusively) with an additional value for log(0). We set this one to 2^f-1. Next, the signature calculations forms a product with α^i, which has i as the logarithm. Thus, we do not need to look up this value. The following pseudo-code for $\mathrm{sig}_{\alpha,1}$ applies these properties. It uses as parameters the address of an array representing the bucket and the size of the bucket. The constant TWO_TO_THE_F is 2^f. The type GFElement is an alias for the appropriate integer or character type. The application to the calculation of $\mathrm{sig}_{\alpha,n}$ is easy.

```
GFElement signature( GFElement *page, int
pageLength)  {
 GFElement returnValue = 0;
 for(int i=0; i< pageLength; i++) {
  if(page[i]!=TWO_TO_THE_F-1)
    returnValue ^= antilog[i+page[i]];
  }
  return returnValue;
}
```

In our file backup application, the bucket usually contains several pages so that we typically calculate the compound signature. To tune the calculus, one should consider the best use of the processor caches, i.e., L1 and L2 caches on our Pentium machines. To increase locality of memory accesses, we first loop on the calculation of $\mathrm{sig}_{\alpha,1}$ for all the pages, then move to $\mathrm{sig}_{\alpha^2,n}$ and so on. Our experiments confirmed that this is better.

5.2. Experimental Performance

We analyzed the performance of the signature scheme based on $\mathrm{sig}_{\alpha,1}$. Our test bed consisted of 1.8 GHz Pentium P4 nodes and from 700 Mhz Pentium P3 nodes over a 100 Mbs Ethernet. We used simulated data and varied the way we calculate the signature. We did the same for $\mathrm{sig}'_{\alpha,n}$. Not surprisingly, both schemes needed about the same calculation times.

We tested our implementation of the signature calculus with simulated data. We varied the details of the $\mathrm{sig}_{\alpha,n}$ and the $\mathrm{sig}'_{\alpha,n}$ calculus implementation and experimented with various ways of compounding. As was to be expected, the calculation times of $\mathrm{sig}_{\alpha,n}$ and the $\mathrm{sig}'_{\alpha,n}$ where the same. Finally, we have ported the fastest algorithm of $\mathrm{sig}_{\alpha,n}$ calculus to SDDS-2000.

Our sample SDDS had records of about 100 B and a 4B key. For both n-symbol signature schemes, we divided the bucket into pages of 16KB and with a 4B signature per page. We selected this page size as a compromise between the signature size (and hence its calculation time) and the overall collision probability of order 2^{-32}, i.e. better than 1 in $4*10^9$. At one back-up operation a second, we can expect a collision every 135 years.

Internally, the bucket in SDDS-2000 has a RAM index because it is structured into a RAM B-tree. The index is small, a few KB at largest. To break up the index into pages of the same size as for bucket pages does not make sense there. We picked a page size of 128 B for the index.

For record updates, we use the scheme where we calculate signatures on the fly only. Alternatively, we could have stored a signature with each record. Recall that the signature calculation is only done for updates and not for inserts.

We present the experiments and their analysis in full in [LMS03], but summarize the main results here:

When we calculated signatures not in the context of SDDS, then the calculation time depended to a large degree on the type of data used. The longest calculation was for strings that consisted of completely random characters in the full range of ASCII and the shortest for highly structured data such as a spelled out number repeated several times. We attribute this behavior to the influence of the various caches.

For a given page size, the calculation times for $\mathrm{sig}_{\alpha,n}$ were linear in n. The actual calculation times of the 4B long signature $\mathrm{sig}_{\alpha,2}$ (calculated in GF(2^{16}), see below) as finally integrated into SDDS-2000 was 20-30 ms per 1 MB of RAM bucket, manipulated as a mapped file. For SHA-1, our tests showed about 50-60 ms. As was desired, it took in the order of dozens of microseconds to calculate $\mathrm{sig}_{\alpha,2}$ for an index page or for a record. The time grew linear with the bucket or record size, and – somewhat surprisingly – turned out to be independent of the algebraic signature scheme tested. Probably due to a better use of the cache, calculating the signature of a 64KB page is relatively faster than the one of a 16KB page. We contrast these times with the actual transfer time of 1 MB from RAM to disk, which is about 300 msec.

Bytes are the smallest usable chunks in a modern computer because of the need to process text efficiently. For this reason, we should choose a Galois field whose elements are bytes or multiple of bytes. Since the logarithm and antilogarithm tables of a Galois field GF(2^f) have sizes in the order of 2^f, we really only have the choice between GF(2^{16}) and GF(2^8). Using the first taxes the cache more because of the larger size of these tables, but the number of Galois field operations is half of that when we make the latter choice. Our experiments showed that the calculation of signatures using GF(2^{16}) turned out to be slightly faster. This justified our final choice to use $\mathrm{sig}_{\alpha,2}$ based on GF(2^{16}) in SDDS-2000.

We have experimented with using signatures to distinguish between updates that change and those that in fact do not change a datum (a pseudo-update). The experimental results are detailed in [H03] and show the expected savings for pseudo-updates (e.g. an almost four-fold gain of pseudo-updates over actual updates for updates of 1KB and a double speed for updates of 100B).

We also ran experiments on a modified SDDS-2000 implementation that uses signatures to distinguish between updates that in fact change the record and those that do not. The latter is a "pseudo-update". We did this for blind updates, which – as we recall change the value of the record absolutely – and for normal updates, which set the new value of the record based on the old value. (See Section 2.2 for the scheme.) The complete results of the experiments are in [H03]. In overview, their results confirm that signature calculation and record update is fast and that savings for pseudo-updates are substantial.

In detail, we measured the time to calculate a signature to be under 5 μsec / KB of data on a 1.8 GHz Pentium 4 machine. A 700MHz Pentium 3 was surprisingly about thirty times slower yielding a rate of 158 μsec / KB. This again shows the sensitivity of the signature scheme to caching. A normal update on the P4-workstation took 0.614 msec per 1KB record, but a normal pseudo-update took only 0.043 msec per 1KB record. The savings amounts to about 90% of the normal update time. The numbers do not include the time it takes to access the record over the net, which is 0.237 msec. If we add this time, then the savings for normal pseudo-updates is only 70%. As expected, processing times for blind updates are faster, namely 0.8372 msec and 0.2707 msec for a true and for a blind pseudo-update, respectively. The times include the key search, the update processing, and the transfer of the record signature. The savings for pseudo-updates are again about 70%.

When we experimented with 100B records, the times were naturally faster. For normal updates we measured 0.419 msec (true update) and 0.03 msec (pseudo-update). Including a search time of 0.22 msec the numbers become 0.63 msec and 0.25 msec, respectively. The total times for blind updates were 0,51 msec and 0,24 msec. The savings were now about 50%. [H03] gives the numbers for the P3 machines.

Finally, we tested our signature scheme for the search in the non-key portion of a record, as laid out in Section 2.3. Since we use GF(2^{16}), that is, since we use symbols of length 2B, and since our records consists of 1B ASCII characters, our code has to take care of an alignment problem, that arises when e.g. the second, third, and fourth byte of a record make up the string for which we are searching. The searches stretched over all 8000 records with a 60B non-key field. We manipulated the bucket so that the third-last record contained the 3B string for which we were searching. The total search time was 1.516 sec, but traversing the bucket already took 0.5 sec. Without this time, we search at a speed of 2 sec per MB. We compared this time with a Karp Rabin type search where instead of our signature we use the byte-wise XOR. This took 1.504 sec (instead of 1.516 sec). Thus, most of the calculation time is spend on memory transfers and very little on Galois field arithmetic.

6. Conclusion and Future Work

Our signature scheme has the new (to our knowledge) property of guaranteed detection of small changes and also allows algebraic operations over the signatures themselves. (Cohen [C97] describes general extension of Karp-Rabin fingerprinting to obtain similar algebraic properties for use in searching. These properties are also used in finding similarities amongst files, e.g. [ABFLS00]. Together with the high probability of detection of any change, including cut-and-paste operations, small overhead, and fast signature calculation, our approach proved to be useful for the SDDS bucket backups and SDDS record updates that motivated our investigation. Our experiments allowed us to successfully add the sig$_{\alpha,n}$ scheme to SDDS-2000 system.

6.1. Signature Properties and Implementation

In the future, we should determine additional algebraic properties of the scheme. We know that sig$^2_{\alpha,n}$ for some α does not have the property of Proposition 1, i.e. that it detects small changes with certainty, but whether this is true for all α remains an open question. In greater generality, is it possible to find a base α consisting of primitive elements that satisfies Proposition 1? It appears that an answer depends on deep properties of finite fields.

We are currently exploring methods to increase the speed of the signature calculation. In a nutshell, the current multiplication operation evaluates an anti-logarithm for every symbol in the page. By using a technique adapted from Broder [B93], we can implement an alternate way to calculate signatures. Preliminary results suggest that it is two to three times faster than the method reported here.

We did not explore the *Prefetch* macro that allows us to save additional calculation time by preloading the L1 and L2 caches in an Intel x86 architecture.

The use of signature trees for computing the compound signatures and explore the signature maps is an open research area.

6.2. Additional Applications

In the context of SDDS, we can apply our scheme to the automatic eviction of SDDS files when several files share an SDDS server whose RAM became insufficient for all the files simultaneously, [LSS02]. Our signature scheme appears to be a useful tool to manage the cache at the SDDS client and to keep the cache and server data synchronized.

There is also an interesting relationship between the algebraic signatures and the Reed-Salomon parity calculus we use for the high-availability SDDS LH*$_{RS}$ scheme, [LS00]. In this scheme, we use GF calculations to generate the parity symbols of a Reed Solomon code ECC (error-control-code). LH*$_{RS}$ combines a small number m of servers into a reliability group and adds k parity servers to the ensemble. The parity servers store parity records whose non-key data consists of parity symbols. We can reconstruct contents of lost servers as long as we can access the data in m out of the m+k total servers in a reliability group. LH*$_{RS}$ needs to guarantee consistency between parity and data servers, i.e., parity and data servers need to have seen the same updates to records. We have shown the existence of an algebraic relation between the signatures of data and parity records which can be used to confirm this consistency between parity and data buckets. The importance of maintaining consistency between parity records and client data records is not unique to SDDS. In a RAID Level 5 disk array, the same problem exists between the parity blocks and the client data blocks. [XMLBLS03] proposes the same scheme in the context of a very large disk farm that uses mirroring and / or parity calculus to secure data against disk failure.

Our techniques should also help other database needs. Especially, they should prove beneficial for a RAM-based database system that typically needs to image the data in RAM to disk as well. RAM sizes have reached GB size and operating system and hardware support is becoming available for even larger RAM. This should add to the attraction of RAM-based database system at the costs of traditional disk-based database systems, [NDLR01].

Our signature-based scheme for updating records at the SDDS client should prove its advantages in client-server based database systems in general. It holds the promise of interesting possibilities for transactional concurrency control, beyond the mere avoidance of lost updates.

Acknowledgments

We thank Lionel Delafosse, Jim Gray, and Peter Scheuermann for fruitful discussions. We gratefully acknowledge the support research grants from Microsoft Research, from the European Commission project ICONS (project no. IST-2001-32429), and the SCU IBM grant 41102-COEN-RSCH-IG-IG09.

Bibliography

[AA93Abdel-Ghaffar,] K. A. S., El-Abbadi, A. *Efficient Detection of Corrupted Pages in a Replicated File.* ACM Symp. Distributed Computing, 1993, p. 219-227.

[ABFLS00] Ajtai, M., Burns, R., Fagin, R., Long, D. Stockmeyer, L.
Compactly encoding unstructured input with differential compression.
www.almaden.ibm.com/cs/people/stock/diff7.ps. IBM Research Report RJ 10187, April 2000.

[BGMF88] Barbara, D., Garcia-Molina, H. , Feijoo, B. *Exploiting Symmetries for Low-Cost Comparison of File Copies.* Proc. Int. Conf. Distributed Computing Systems, 1988, p. 471-479.

[BL91] Barbara, D., Lipton, R. J.: *A class of Randomized Strategies for Low-Cost Comparison of File Copies*. IEEE Trans. Parallel and Distributed Systems, vol. 2(2), 1991, p. 160-170.

[B93] Broder, A. *Some applications of Rabin's fingerprinting method*. In Capocelli, De Santis, and Vaccaro, (ed.), Sequences II: Methods in Communications, Security, and Computer Science, pages 143--152. Springer-Verlag, 1993.

[C02] Ceria Web site. http://ceria.dauphine.fr/.

[C97] Cohen, J. *Recursive Hashing Functions for n-Grams.* ACM Trans. Information Systems, Vol 15(3), 1997, p. 291-320.

[FC87] Faloutsos, C., Christodoulakis, S.: *Optimal Signature Extraction and Information Loss.* ACM Trans. Database Systems, Vol. 12(5), p. 395-428.

[FWA86] Fuchs, W. Wu, K. L., Abraham, J. *A. Low-Cost Comparison and Diagnosis of Large, Remotely Located Files.* Proc. Symp. Reliability Distributed Software and Database Systems, p. 67-73, 1986.

[Go99] D. Gollmann: Computer Security, Wiley, 1999.

[H03] Hammadi, B. *Suppresions et mises a jour dans le système SDDS-2000*. CERIA research report CERIA-BD-DEA127.

[KR87] Karp, R. M., Rabin, M. O. *Efficient randomized pattern-matching algorithms.* IBM Journal of Research and Development, Vol. 31, No. 2, March 1987.

[KC 95] Jeong-Ki Kim, Jae-Woo Chang: "A new parallel Signature File Method for Efficient Information Retrieval", CIKM 95, p. 66-73.

[KC99] S. Kocberber, F. Can: "Compressed Multi-Framed Signature Files: An Index Structure for Fast Information Retrieval", SAS 99, p. 221-226.

[KPS02] Kaufman, C., Perlman, R., Speciner, M. *Network Security: Private Communication in a Public World*, 2nd Ed., Prentice-Hall 2002.

[LBK02] Lewis, Ph., M., Bernstein. A. Kifer, M. Database and transaction Processing. Addison & Wesley, 2002.

[LNS94] Litwin, W., Neimat, M-A., Schneider, D. RP* : A Family of Order-Preserving Scalable Distributed Data Structures. 20th Intl. Conf on Very Large Data Bases (VLDB), 1994.

[LNS96] Litwin, W., Neimat, M-A., Schneider, D. LH*: A Scalable Distributed Data Structure. ACM Transactions on Database Systems ACM-TODS, (Dec. 1996).

[LMS03] Litwin, W., Mokadem, R., Schwarz, T.: *Disk Backup through algebraic signatures in scalable and distributed data structures. Proc.* 5th Workshop on Distributed Data and Structures, Thessaloniki, June 2003 *(WDAS 2003).*

[LS00] Litwin, W., Schwarz, Th. LH*$_{RS}$: A High-Availability Scalable Distributed Data Structure using Reed Solomon Codes. ACM-SIGMOD 2000.

[LRS02] Litwin, W., Risch, T., Schwarz, Th. An Architecture for a Scalable Distributed DBS :Application to SQL Server 2000. 2nd Intl. Workshop on Cooperative Internet Computing (CIC 2002), August, 2002, Hong Kong.

[LSS02] Litwin, W., Scheuermann, P., Schwarz Th. Evicting SDDS-2000 Buckets in RAM to the Disk. CERIA Res. Rep. 2002-07-24, U. Paris 9, 2002.

[Me83] Metzner, J. *A Parity Structure for Large Remotely Located Data Files*. IEEE Transactions on Computers, Vol. C – 32, No. 8, 1983.

[MA00] Michener, J., Acar, T. *Managing System and Active-Content Integrity*. Computer, July 2000, p. 108-110.

[N95] Natl. Inst. of Standards and Techn. *Secure Hash Standards.* FIPS PUB 180-1, (Apr. 1995).

[NDLR00] Ndiaye, Y., Diène A., W., Litwin, W., Risch, T. *Scalable Distributed Data Structures for High-Performance Databases.* Intl. Workshop on Distr. Data Structures, (WDAS-2000). Carleton Scientific (publ.).

[NDLR01] Ndiaye, Y., Diène A., W., Litwin, W., Risch, T. AMOS-SDDS: A Scalable Distributed Data Manager for Windows Multicomputers. 14th Intl. Conference on Parallel and Distributed Computing Systems- PDCS 2001.

[S88] Sedgewick, R: *Algorithms*. Addison-Wesley Publishing Company. 1988.

[SBB90] Schwarz, Th., Bowdidge, B., Burkhard, W., Low Cost Comparison of Files, Int. Conf. on Distr. Comp. Syst., (ICDCS 90) , 196-201.

[XMLBLS03] Xin, Q., Miller, E, Long, D., Brandt, S., Litwin, W., Schwarz, T. *Selecting Reliability Mechanisms for a Large Object-Based Storage System.* 20th Symposium on Mass Storage Systems and Technology. San Diego. 2003.