Performance Improvement of State Space Exploration by Regular & Differential Hashing Functions

Bernard Cousin
Jean-michel Hélary

IRISA - Université de Rennes-I
Campus universitaire de Beaulieu
35042 - Rennes cédex
FRANCE

phone : (33) 99.84.73.33
fax : (33) 99.38.38.32
E-mail : bcousin@irisa.fr

Abstract
This paper presents regular hashing functions. Used in conjunction with differential computation process, regular hashing functions enable searching time of global state to be optimized. After a formal definition of the regular property for hashing functions, we propose a characterization of this property. Then the formal definition of differential hashing function is given. Next, we show the performance acceleration produced by the precomputed differential computation process applied to three hashing functions commonly used. The observed accelerations can be significant because the complexity of proposed implementation is independent of key length or respectively of item difference contrary to the usual or respectively differential implementations. Last we study the performances of precomputed differential hashing computation process on reachability graph exploration of distributed systems specified by Petri net using the Bouster tool, and on state space exploration of protocols specified by Lotos using the Open/Caesar environment.

1. Introduction

Hashing method is a well suited method to achieve state-space exploration for verification of distributed systems. Hashing method is used as searching method to accelerate the retrieval process of a particular state among large state space under exploration. This method enables searching, insertion and suppression operations to be done on average at a constant cost in number of comparisons. But the usual computation process of the hashing value has the first following drawback : its complexity is at least proportional to the key length, and unfortunately, the state descriptor, from which the keys are based, is in general very large (several hundreds of bytes [Doldi 92, Holzmann 91, Wolper 93]), if accurate modelling is considered.

Moreover, some of the recent works on state-space exploration made an intensive use of hashing functions (bitstate method [Holzmann 88], multihash method [Wolper 93]) : two hashing function calls in Holzmann's Spin validation environment for each newly created state, Wolper recommends 20 calls for each newly created state to achieve large coverage of the state-space. These two methods reduce the amount of space needed to store the explored state-space but, as Wolper writes, due to the intensive function calls, they have the second following drawbacks : "computing 20 hash functions is quite expensive and will substantially slow down the search".

In previous work on improvement of state-space exploration we have introduced *differential* hashing functions [Cousin 93]. These hashing functions use differential

computation process of the hashing value which replaces the usual computation process, and which optimizes their processing time. The proposed optimization is based on the following observation: key structure can be viewed as record of items. The computation process is called differential, if we can infer the hashing value of a key from the hashing value of another key which differs from the previous key in only few items. It can be more efficient to deduce the new hashing value knowing the value of some few new items rather than to apply the usual computation process on every item of the new key.

In opposition to the usual one, the differential computation process complexity is proportional to the difference number, which is the number of modified item between two successive keys. In general, studied systems have inherently successive states whose keys have few differing items (less than 10%) : the subsystems of a distributed system do not evolve simultaneously and at every moment. And our performance evaluation of differential computation process for hashing function has showed that it enables a substantial processing time improvement for typical distributed systems analysis.

However when the difference ratio is high the improvement can vanish because each difference in the key needs to be computed by a so-called mono-differential function. Even if the complexity of the mono-differential computation is far less than the complexity of the usual computation, the time spent into numerous function calls can exceed the gain in complexity.

This is why in this current work we propose an improvement of the differential computation process. This improvement is based on the knowledge of the mathematical operations which produce the new states. This knowledge enables the differential hashing value associated to each transition of the model to be computed during the initial loading phase of the model (or during the first firing of each transition). Similarly to the transition function associated to each transition which enables a new state to be produced from a previous state, the differential hashing value enables the hashing value of a new state to be computed from the hashing value of the previous state. The precomputed value is obtained from the same items as the original differential method : the items which differ. So we called it the differential hashing value.

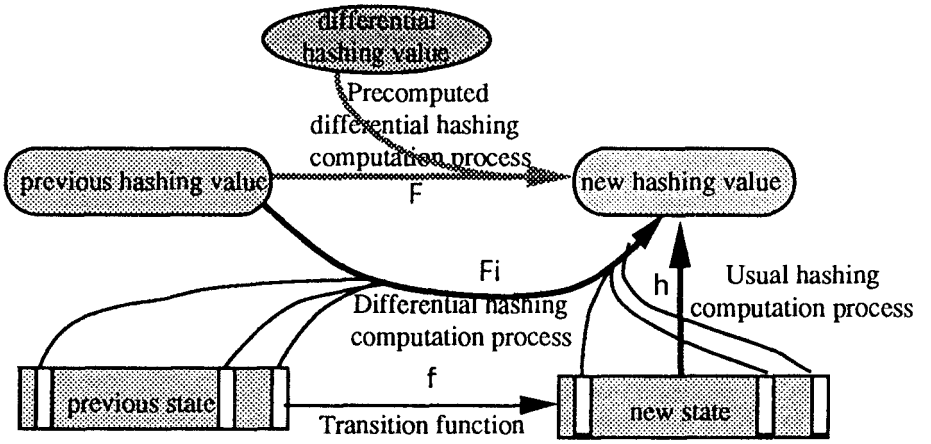

Figure 1 - Hashing value computation

From data point of view, there can be three different manners to implement the same hashing function (Figure 1): the usual hashing computation process which is

computed from all the items of the new state; the differential hashing computation process which uses the previous hashing value and all the items of the previous state and of the new state which differ [Cousin 93]; and the precomputed differential hashing computation process which uses the previous hashing value and the differential hashing value associated with the transition which has produced the new state.

At storage and computing cost of one differential hashing value for each transition of the model, the precomputed and differential method enables the complexity of hashing value to be completely independent of the length of the key and the number of differences. Unfortunately, a precomputed differential computation process can not be associated with every existing hashing function : first, because the hashing function has to be differential and it has been shown in our previous work that all hashing functions are not differential; second, because the operations used by the hashing function have to be compatible with the operations used by the transition function - this property is called *regularity*.

In the second section, after a formal description of regular functions and of differential hashing functions, we produce a characterization of the regular function. This characterization can be useful to find a precomputed differential hashing computation process because all hashing functions as well as all differential hashing functions are not necessarily regular.

The third section addresses the following question: Are the precomputed differential computation processes always more efficient than the differential or usual computation processes? We show that this is always the case, and furthermore, the precomputed differential implementation, enabling very efficient coding, cuts down the processing time significantly.

The fourth section gives the performances achieved by the precomputed differential hashing function used in the Bouster verification tool [Bonafos 90], based on formal description of distributed systems by Petri net, and the Open/Caesar verification tool [Garavel 90] based on Lotos. These state-space exploration examples allow us to exhibit the advantages and the limits of our precomputed differential computation process.

2. Regular functions

2.1 Presentation

Regular functions are defined in relation to other applications which represent the transition functions used to produce the new state. As stated above, the hashing functions have to be compatible with the transition functions, to enable the precomputation of the differential hashing value. We call this compatibility the regular property.

Definition: regular property

An application h from A to B is regular for the application f from A to A (h is said f-regular) if and only if there exists an application F from B to B such as :

$$f \circ h = h \circ F. \qquad (1)$$

An example of regular function is given by functions who have a right-inverse i.e. there exists h^r from B to A such as $h \circ h^r = Id$ where Id is the identity function on A.

In fact, $\forall f$, taking $F = h^r \circ f \circ h$ we have $h \circ F = h \circ h^r \circ f \circ h = f \circ h$ (q.e.d.).

2.2 Characterization

We characterize the applications h from A to B which are f-regular. First we restrict ourself to surjections, that is to say to applications h such as h(A) = B.

Theorem :

A surjection h from A to B is f-regular if, and only if, it satisfies the following property: $\forall x \in A, \forall y \in A, h(x) = h(y) \Rightarrow h(f(x)) = h(f(y))$. (2)

Proof :

a) the condition (2) is necessary.

Suppose, by hypothesis, that h is f-regular ; let us assume that $\forall x \in A, \forall y \in A, h(x) = h(y)$.

According to the definition (1) the f-regular application h respects the following property : $\exists F, f \circ h = h \circ F$.

Therefore, since F is an application : $\forall x \in A, \forall y \in A, h(x) = h(y) \Rightarrow F(h(x)) = F(h(y))$.

According to the definition (1) : $\forall x \in A, \forall y \in A, h(x) = h(y) \Rightarrow h(f(x)) = h(f(y))$, thus (2) is satisfied.

b) The condition is sufficient.

Let $\mathcal{R}$ be the equivalence relation on A : $x \mathcal{R} y \Leftrightarrow h(x) = h(y)$.

Let us denote the quotient set $A/\mathcal{R}$ by $\tilde{A}$, and, for any $x \in A$, let $\tilde{x}$ denote the class of x. With the surjection h can be associated the canonical injection $\tilde{h}$ from $\tilde{A}$ to B defined by $\forall \tilde{x} \in \tilde{A}, \tilde{h}(\tilde{x}) = h(x)$, where x is a particular element of $\tilde{x}$.

Clearly, $\tilde{h}$ is an application since if x' is another element of $\tilde{x}$, we have h(x') = h(x).

Similarly $\tilde{h}$ is an injection since $\tilde{h}(\tilde{x}) = \tilde{h}(\tilde{y}) \Rightarrow h(x) = h(y)$ where $x \in \tilde{x}$ and $y \in \tilde{y}$,

$$\Rightarrow x \mathcal{R} y$$

$$\Rightarrow \tilde{x} = \tilde{y} .$$

Finally, $\tilde{h}$ is an surjection since $\tilde{h}(\tilde{A}) = h(A) = B$. Thus $\tilde{h}$ is a bijection from $\tilde{A}$ onto B. With the application f from A onto A, we associate the relation $\tilde{f}$ from $\tilde{A}$ to $\mathcal{P}(\tilde{A})$, where $\mathcal{P}(\tilde{A})$ denotes the set of subsets of $\tilde{A}$, defined by : $\forall \tilde{x} \in \tilde{A}, \tilde{f}(\tilde{x}) = \{\widetilde{f(x)}$ such as $x \in \tilde{x}\}$. In others words : $\tilde{y} \in \tilde{f}(\tilde{x}) \Leftrightarrow \exists x \in \tilde{x}, \exists y \in \tilde{y}$, such as f(x)=y. (3)

Let us show that $\tilde{f}$ is an application from $\tilde{A}$ on to $\tilde{A}$.

Suppose that $\tilde{f}$ has two images $\tilde{y}$ and $\tilde{z}$ for the same element $\tilde{x}$:

$\tilde{y} \in \tilde{f}(\tilde{x})$ and $\tilde{z} \in \tilde{f}(\tilde{x})$.

By definition (3) :

$\exists x \in \tilde{x}, \exists y \in \tilde{y}$, such that f(x)=y and $\exists x' \in \tilde{x}, \exists z \in \tilde{z}$, such that f(x')=z.

But $x \mathcal{R} x'$ and thus h(x) = h(x').

By the hypothesis (2) : $h(x) = h(x') \Rightarrow h(f(x)) = h(f(x'))$; hence $f(x) \mathcal{R} f(x')$ i.e. $y \mathcal{R} z$, and thus $\tilde{y} = \tilde{z}$ (q.e.d.).

Two images by $\tilde{f}$ of the same element are not distinct, so $\tilde{f}$ is an application. Consider the application F from B to B by composition of the previous applications via $\tilde{A}$: $F = \tilde{h}^{-1} \circ \tilde{f} \circ \tilde{h}$ (Figure 2). Finally we show that $h \circ F = f \circ h$.

Let φ from A to $\tilde{A}$ be the application defined by $\forall x \in A, \varphi(x) = \tilde{x}$. We have $\forall x \in A$,

$(f \circ \varphi)(x) = \varphi(f(x)) = \widetilde{f(x)} = \tilde{f}(\tilde{x}) = \tilde{f}(\varphi(x)) = (\varphi \circ \tilde{f})(x)$, and thus $f \circ \varphi = \varphi \circ \tilde{f}$. (4)

Composing on the right with $\tilde{h}$: $f \circ \varphi \circ \tilde{h} = \varphi \circ \tilde{f} \circ \tilde{h}$,

which can be written : $f \circ \varphi \circ \tilde{h} = \varphi \circ \tilde{h} \circ \tilde{h}^{-1} \circ \tilde{f} \circ \tilde{h}$. (5)

But , $\forall x \in A$, $(\varphi \circ \tilde{h})(x) = \tilde{h}(\varphi(x)) = \tilde{h}(\tilde{x}) = h(x)$, and thus : $\varphi \circ \tilde{h} = h$. (6)

Putting this relation in (5) gives : $f \circ h = h \circ \tilde{h}^{-1} \circ \tilde{f} \circ \tilde{h}$, that is to say

$f \circ h = h \circ F$ by definition of F. (q.e.d.)

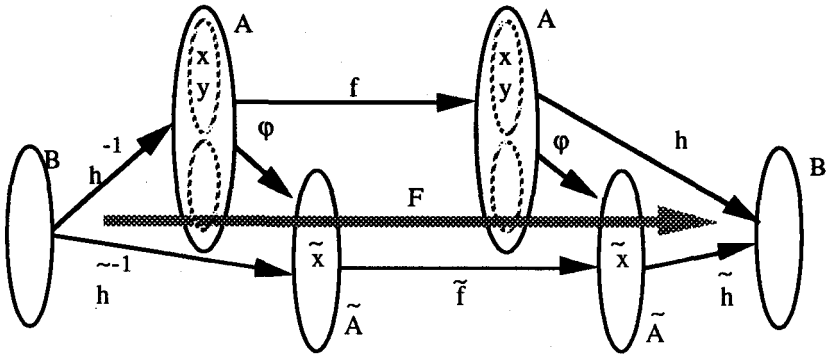

Figure 2 - F construction

The case where h is not a surjection is straightforward. Consider first F' from h(A) to h(A) such that $f \circ h = h \circ F'$, then extend arbitrarily F' to F from B to B.

2.3 Examples

For instance, let $\bigoplus$ be the application defined as the "exclusive or" between all key items. Each item belongs to the same set B. Formally :

$$\forall \langle x_1,\ldots,x_i,\ldots,x_N \rangle \in \prod_{k=1}^{N} B, \bigoplus(\langle x_1,\ldots,x_i,\ldots,x_N \rangle) = x_1 \oplus \ldots \oplus x_i \oplus \ldots \oplus x_N$$ with $\oplus$ the usual "exclusive Or" operator from B to B.

The application $\bigoplus$ is regular towards the usual "exclusive Or" operator ($\oplus$).

Proof :

As the composition of the operators $\oplus$ is commutative and associative, similarly the composition of the application $\bigoplus$ and the operator $\oplus$ is commutative and associative, and therefore, $\bigoplus$ is $\oplus$-regular: $\oplus \circ \bigoplus = \bigoplus \circ \oplus$ (q.e.d.).

So the precomputed differential hashing function of the hashing function $\bigoplus$ for the transition function $\oplus$ is $\oplus$ itself.

But the application $\bigoplus$ is not regular towards the usual addition (+).

Proof :
Counter-example of the characterization property (2):
Let be N =3, $x_1 = 1$, $x_2 = 2$, $x_3 = 3$.

$\bigoplus(<1,2,3>) = \bigoplus(<1,3,2>) = 0$ but $\bigoplus(<1,3,2+1>) \neq \bigoplus(<1,2,3+1>)$ because

$\bigoplus(<1,3,2+1>)=1\oplus3\oplus(2+1)=1\oplus3\oplus3=1$ and $\bigoplus(<1,2,3+1>)=1\oplus2\oplus(3+1)=1\oplus2\oplus4=7$ (q.e.d.).

This example explains the important of the choice of good hashing functions which are regular towards the transition function, all the more the hashing functions has to be differential.

2.4 Differential functions

Hashing function is an application with N variables from a product of sets $\prod_{k=1}^{N} A_k$ to a set B. We denote $h(<x_1,...,x_i,...,x_N>)$ the hashing value of the key $<x_1,...,x_i,...,x_N>$.

Definition : mono-differential computation function

The hashing function h has a mono-differential computation function of the hashing value for its i^{th} item if and only if it exists a function F_i from $B \times A_i \times A_i$ to B such that :

$$\forall <x_1,...,x_i,...,x_N> \in \prod_{k=1}^{N} A_k,\ \forall y_i \in A_i,\ F_i(h(<x_1,...,y_i,...,x_N>),\ y_i,\ x_i) = h(<x_1,...,x_i,...,x_N>). \quad (7)$$

The mono-differential function (F_i) enables the computation of the hashing value of a key ($<x_1,...,x_i,...,x_N>$: called son_key) from the hashing value of another key ($<x_1,...,y_i,...,x_N>$: called father_key) having in common with the previous key every items but one (i: called modified item). Knowing the old value of the modified item of the father_key (y_i), the new value that this modified item must have in the son_key (x_i), and the hashing value computed from the father_key ($h(<x_1,...,y_i,...,x_N>)$), the mono-differential function associated with the modified item enables the computation of the hashing value associated with the son_key ($h(<x_1,...,x_i,...,x_N>)$).

Definition : differential hashing function

If for every item of the keys of the hashing function h studied there exists one mono-differential computation function, then the set of these mono-differential functions constitutes a complete differential computation function family. If a hashing function has a complete differential computation function family, we shall say that it is differential.

2.5 Characterization

We can characterize the hashing functions which are differential, that is, which admit a complete differential computation function family.

Not every hashing function admits differential computation functions. For instance, the hashing function "⊗" built over a binary set product and defined as the binary operator "logical And" is not differential. In fact, no mono-differential

computation function can be established: $F_1(0,0,1)$ can be equal either to $F_1(\otimes<0,1>,0,1)=\otimes(<1,1>)=1$, or to $F_1(\otimes<0,0>,0,1)=\otimes(<1,0>)=0$, which contradicts the image unicity property.

Theorem :

The hashing functions which admit a mono-differential computation function for their i^{th} item are characterized by the following property:

$$\forall <x_1,...,x_i,...,x_N> \in \prod_{k=1}^{N} A_k,\ \forall <y_1,...,y_i,...,y_N> \in \prod_{k=1}^{N} A_k,\ h(<x_1,...,x_i,...,x_N>) = h(<y_1,...,x_i,...,y_N>) \Leftrightarrow h(<x_1,...,y_i,...,x_N>) = h(<y_1,...,y_i,...,y_N>). \quad (8)$$

The proof that the property (8) is a necessary and sufficient condition in order that the hashing functions admit a complete differential computation function family can be found in [Cousin 93b].

2.6 Example

Consider the same hashing function $\oplus$, defined previously as the "exclusive or" between all the key items, each item belonging to the same set B.

The mono-differential hashing computation functions F_i can be defined as

$$F_i(\oplus(<x_1,...,y_i,...,x_N>),\ y_i,\ x_i) = \oplus(<x_1,...,y_i,...,x_N>) \oplus y_i \oplus x_i$$

The proof that the hashing function $\oplus$ has a complete mono-differential computation functions ($\forall i \in [1,N],\ \exists\ F_i$) can be found in [Cousin 93].

3. Performance study

3.1 Presentation

In this section, we address the issue of performance improvement achieved by the precomputed differential computation process. For that purpose, we compare the performances obtained by hashing functions using precomputed differential computation process to usual or differential ones.

Due to the reduction of the image definition domain, hashing functions can associate several keys with one unique hashing value. Hashing methods associate specific functions with hashing functions to resolve collisions. As the precomputed and differential computation processes return exactly the same value than the usual computation process, the observed performance improvements are independent of the performance of the collision resolution function, and thus, are entirely preserved. So our study focuses on performances of hashing functions without studying the collision resolution functions. Nevertheless, previous studies have shown that the distribution of the three following hashing functions are good to very good. So, the preconceived idea according to which regular and differential properties induce inefficient hashing functions, is not well founded.

Three hashing functions have been chosen among those found in the literature. This sample does not cover all existing hashing functions, but these three functions have the advantage to be regular and differential. Besides this advantage, the fact that these functions are widely used and the good performance provided by their coding simplicity have also been taken in consideration as a criterion of choice. A simple algorithm with short code is often faster than a complex and long algorithm. Further

studies are undertaken to search differential algorithms of other hashing functions, in order to increase the study spectrum.

The proposed hashing functions use the following basic operators: division/product, modulo, addition/subtraction, folding [Knott 75]. Other operators can been used: power/root, radix transformation, polynomial computation, etc [Lum 71] : since they have long computation time, we choose not to use them.

Let F_i^X denotes the differential hashing function for the i^{th} item of the hashing function h^X, F^X denotes the precomputed differential function over the transition function f, and δ_f denotes the set of different items for the transition function f : δ_f = {x_i such that $x_i \neq f(x_i)$}.

The first function (h^1) uses the folding method based on the logical operator "exclusive or", as previously described :

$$h^1(<x_1,...,x_i,...,x_N>) = \bigoplus_{i=1}^{N} x_i ,$$

$$F_i^1 (h^1(<x_1,...,x_i,...,x_N>), x_i, y_i) = h^1(<x_1,...,x_i,...,x_N>) \oplus y_i \oplus x_i ,$$

$$F^1(h^1<x_1,...,x_i,...,x_N>, f) = h^1(<x_1,...,x_i,...,x_N>) \oplus (\bigoplus_{x_i \in \delta f} (f(x_i) \oplus x_i)).$$

The second function (h^2) is a sum of the key items weighted by the power of the prime constant p:

$$h^2(<x_1,...,x_i,...,x_N>) = \sum_{i=1}^{N} x_i.p^{(i-1)},$$

$$F_i^2 (h^2(<x_1,...,x_i,...,x_N>), x_i, y_i) = h^2(<x_1,...,x_i,...,x_N>) + (y_i - x_i).p^{(i-1)} ,$$

$$F^2(h^2<x_1,...,x_i,...,x_N>, f) = h^2(<x_1,...,x_i,...,x_N>) + \sum_{x_i \in \delta f} (f(x_i) - x_i).p^{(i-1)} .$$

The last function (h^3), which can be found in [Knuth 73], combines multiple precision integer arithmetic and modulo operation. b is equal to the number of bits needed to code every item. K is a prime number. In short, the whole key is interpreted as multiple precision integer.

$$h^3(<x_1,...,x_i,...,x_N>) = \left(\sum_{i=1}^{N} x_i.2^{(i-1).b} \right) \bmod K ,$$

$$F_i^3 (h^3(<x_1,...,x_i,...,x_N>), x_i, y_i) = (h^3(<x_1,...,x_i,...,x_N>) + (y_i - x_i).2^{(i-1).b}) \bmod K,$$

$$F^3(h^3<x_1,...,x_i,...,x_N>, f) = (h^3(<x_1,...,x_i,...,x_N>) + \sum_{x_i \in \delta f} (f(x_i) - x_i).2^{(i-1).b}) \bmod K.$$

The performances of the three hashing functions are displayed in two graphs. In both graphs, the time unit is $1/60.10^{-6}$ second.

The first graph shows the performance results of the usual algorithm and differential algorithm for various key lengths [Figure 3]. The codes of the mono-differential and precomputed processes are similar so their processing times are almost equal : to facilitate the graph reading, only the differential values are drawn. As the

processing times can depend on the modified item, we use the following process to obtain meaningful results : we compute the mean of several processing time based on random drawings of the modified item.

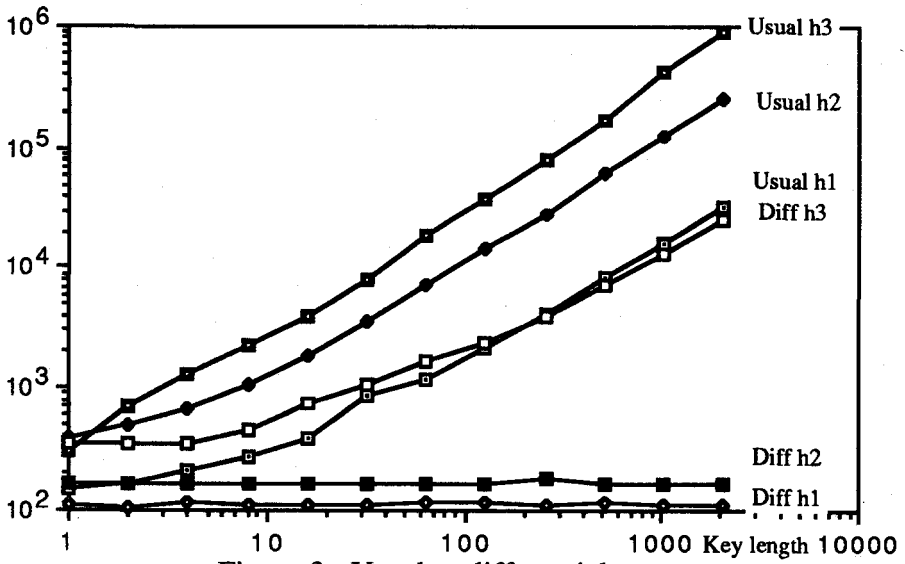

Figure 3 - Usual or differential processes

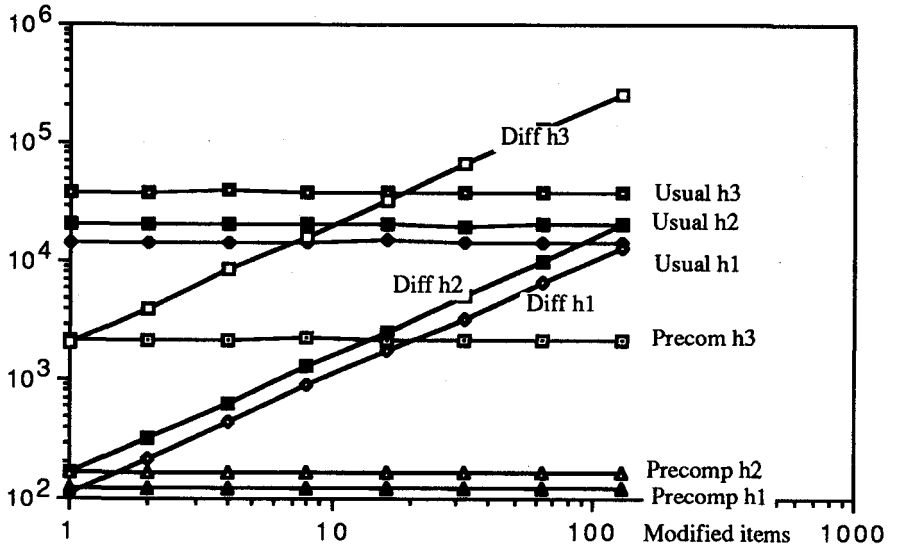

Figure 4 - Usual, differential or precomputed processes

The second graph shows the processing time obtained by the usual algorithm, the differential algorithm and the precomputed differential algorithm [Figure 4]. In this case the key length is fixed and equal to 128. We compute the processing time for various numbers of simultaneous modified items.

3.2 Preliminary discussion

Before going inside the comparison of hashing function computation processes (usual, differential or precomputed), it is worth to discuss some facts concerning the functions themselves. First, we observe that all the functions have an usual computation process which increases in time depending on the length of the key (Figure 3). In fact, to be efficient, hashing functions must use all the key items, if they are significant, to compute a hashing value [Knuth 73]. So the usual computation processes have similar behavior : their computation time increases according to the key length.

Recall that the displayed times do not show the overall performance times of hashing methods, because hashing methods are based on hashing/collision resolution function pair. We emphasize that an inadequate hashing function can generate numerous collisions which will degrade considerably the overall performance of the hashing method. However, we should not conclude that the obtained results on hashing computation time are not significant. In fact, the time due to resolution collision is independent of key length, so it becomes negligible for long keys compared to the computation time of the hashing functions [Deudon 92]. Furthermore experiments show that the three previous hashing functions are rather good scattering functions.

We have made these performance tests over numerous versions of the hashing functions: word length variation (1, 2, 4 bytes item), modification of the overflow treatment, table extraction or computation of the computational factor, etc. The previous and following described behaviors have been maintained, even if some local optimizations have been noted.

3.3 Results

If we compare the results obtained from the differential algorithm to those from the usual algorithm, we observe that the processing time of the differential algorithm is, on the one hand, shorter than the processing time of the usual algorithm, and on the other hand, constant with respect to key length, with the exception of the hashing function number 3 (Figure 3). In fact, the implementation of the differential algorithm of the third hashing function uses multiple precision integer arithmetic. So the processing time increases according to key length.

The influence of the modified item location can be observed over all the hashing functions and their differential algorithms, but this influence is very low over all the functions except the number 3 hashing function, as established in the previous paragraph. The other two functions, although their absolute duration is short, have some erratic variations. In fact, theses variations are generated by either the coincidence or not coincidence of the modified item location with some constants used during differential computation process.

We recall that the previous results have been established by algorithms based on mono-differential functions. In case of multiple differences (case where the son_key differs from the father_key by more than one item) the processing time can be deduced from the mono-differential result : it is equal to the product of this value and the number of differences. So the duration is proportional to the number of different items between the two keys.

The Figure 4 exhibits the following behaviors: the processing times of the usual or precomputed algorithms are constant : they are independent of the number of simultaneous modified items; even though the processing time of differential algorithm is proportional to the number of simultaneous modified items. As stated in

our previous work, the processing time of differential algorithm is shorter than the one of usual algorithm, only if the number of modified item is low (>10%). Previous experiments have pointed out that this condition is verified in practice, and accordingly differential algorithm improves the processing time of state space exploration process.

Nonetheless the current measurements show real improvement of the processing time using the precomputed differential algorithm compared with the two other algorithms. Shorter are the processing time for usual or differential algorithm, shorter are the precomputed algorithm, for a ratio from 20/1 to more than 100/1.

4. Application to state space exploration

Usual state spaces can be very large. In fact, the graph size and the number of states depend on parallelism and accuracy of the modelized system. Previous studies lead to establish that real distributed system or protocol models have states whose size is significant: 1916 bytes for P-channel protocol [Doldi 92]; hundreds of bytes for Holzmann [Holzmann 91]; or from our own experiments several hundred of bytes (Transport class 4 protocol).

Large state space exploration raises two main problems : very huge storage and very long computation time. Precomputed differential computation process only address the second problem, but many methods, trying to reduce the storage requirement, increase their processing time (it is the usual time/space trade-off). Several of these methods make intensive use of hashing functions [Holzmann 88, Wolper 93]. In this case our proposition can be very effective if it can be combined with these methods : under those circumstances, state space exploration requires less storage space and shorter processing time.

An intensive performance test campaign has been carried out using either the Bouster validation tool [Campergue 91] or the Open/Caesar verification environment [Garavel 90] on a set of several distributed system models. These models had various number of places (2 to 50000), various numbers of transitions (1 to 50000), various numbers of arcs (10 to 100000) and they produce several hundred thousand states. These models either have been obtained from an automatic tool which generates models with specific or symmetrical topologies, either have been found in literature, or have been provided by the package of the tool itself.

Using the Unix profiler tool, our study established that the hashing function, the collision resolution and access function, the transition selection function and the compare function are the most time consumer functions: each of these functions consume on average 15 to 40 % of the processor time in user mode depending on the distributed system studied, and in various order. The order and the utilization time of these functions are variable because they depend on the collision rate which itself depends on the hashing function, the size of the hashing table and the model characteristics. All the other functions without exception used less than 10% of the processor time (most of them significantly less). Some specific methods, like bitstate method, can raise the processor utilization rate of hashing function to more than 50%, if the collision rate is kept low (i.e. hashing table size is close to explored state space size).

We have built three versions for both verification tools using the h^2 function which is faster than the h^3 function and has better distribution than the h^1 function. The first version uses the usual computation process, the second version the differential computation process, the third one the precomputed differential computation process. Let us recall that the first tool (Bouster) uses Petri net as

description language, while the second one (Caesar) uses Lotos. We have selected some significant results (Table 5).

First, all the measurements show an hashing function speed-up produced by the precomputed differential computation process compared with the usual computation process from 100/1 to 70/1, so our technic is rather efficient. Second, in contrast with the differential speed-up which varies with the studied system, the precomputed speed-up is quite constant and independent of the context. Third, the results show that the acceleration is independent of the description language and of the verification method used.

	Total time	usual comp.	diff. comp.	precomp. proc.
TP4 protocol (P. net)	2064.6	621.5	147.8	18.9
alternate bit protocol (P. net)	880.3	367.1	67.5	7.3
alternate bit protocol (Lotos)	2357.0	1043.4	214.4	40.5
symmetrical ring (P. net)	5040.6	2405.3	1070.1	23.6
symmetrical ring (Lotos)	14867.1	8655.4	4367.2	96.3

Table 5 - processing time

These results exhibit at the same time the importance and the limits of the gain which can likely be achieved with the precomputed differential process. In fact, on average the processor spends about 40% in the average of its user mode time in the code of the hashing function and collision resolution function. A fast hashing function, judicious and balanced, should enable this processing time to be reduced, decreasing the collision rate and hashing value computation time. Nevertheless, the performance increase due to a differential technique can not magically reduce the inherent complexity of the system studied, in particular the huge number of states that we sometimes need to generate. Other methods like data densification, partial exploration, on the fly verification, etc can be combined to advantage with our method.

These promising results must not hide an important phenomena already raised by numerous performance researchers: the influence of the virtual memory mechanism on execution time. In fact, a considerable slow down is noticed as soon as the data application can not longer be kept in core memory. Nevertheless the direct access technique offered by the hashing method as long as the collision rate is kept low, favors this method against all other proposed methods because it reduces the inputs and outputs between secondary and main memory.

5. Conclusion

The results show that differential hashing speed-up increases with key length. In fact, typical hashing functions use all key items (this process is recommended to enable the hashing value distribution to be balanced); hence the usual algorithm complexity is proportional to the key length. In the context of many applications (large graph, very numerous states) long key lengths are generated, as corroborated by several examples. If the differential algorithm complexity is proportional to the modified item number between the original key and the new key, on the contrary, precomputed differential algorithm complexity is fixed, and its processing time is comparable to the processing time of one function call of the mono-differential algorithm, which is very short.

The differential hashing functions are not a general answer to all the computation time problems: they do not always exist, and when they exist, they are not always the most efficient. But our previous work establishes that, first, numerous hashing functions can be associated with a complete mono-differential function set (i.e. they

are differential); second, for the majority of differential computation processes processing time is shorter than for the usual ones; third, differential techniques require applications where the modified items can be obtained at low cost (no need of modified item searching). The proposed application (state space exploration) has all these prerequisite characteristics, and consequently enables a substantial improvement of performances.

The current work shows that, if the hashing function is regular for the transition functions and if the hashing function is differential then its coding produces drastic improvements in processing time. In conjunction with state space compression methods the precomputed differential implementation of hashing functions have shown their great efficiency : reducing of both storage and processing time requirements to achieve the verification of distributed systems.

References

[Algayres 91] B.Algayres, & all, "Vesar: a Pragmatic Approach to Formal Specification and Verification", Computer Network and ISDN Systems, vol 25 n°7, February 1991.

[Bonafos 90] B.Bonafos, E.Domingo, "Leda : Structured Language for Automata Description and Verification", rapp. de recherche, Bordeaux-France, june 1990.

[Campergue 92] C.Campergue, C.Nouaille, "Bouster : génération parallèle du graphe des marquages accessibles", rapport ENSERB, Bordeaux-France, Juin 1992.

[Cousin 93] B.Cousin, "Differential Hashing Functions : Application to Reachability Graph Generation". ICCI'93. Sudbury - Canada, 26-29 May 1993.

[Cousin 93b] B.Cousin, "Les fonctions de hachage différentielles". CFIP'93. Montréal - Canada, 7-9 septembre 1993. Hermès, p525-541.

[Deudon 92] G.Deudon, C.Houillon, "Techniques de hachage", rapport interne ENSERB, Bordeaux-France, Juin 1992.

[Dimitrijevic 89] D.D.Dimitrijevic, M.S. Chen, "Dynamic State Explosion in Quantitative Protocol Analysis",PSTV-IX, Twente-Netherland, 6-9June 1989.

[Doldi 92] L.Doldi, P.Gauthier, "Veda-2: Power to the Protocol Designers", FORTE'92, Lannion-France, 13-16 Octobre 1992.

[Garavel 90] H.Garavel, J.Sifakis, "Compilation and Verification of Lotos Specifications", PSTV-X, Ottawa, june 1990.

[Holzmann 88] G.J.Holzmann, "An Improved Protocol Reachability Analysis Technique", Sofware, Practice and Experience, vol 18 n°2, Feb. 1988.

[Holzmann 91] G.J.Holzmann, "Design and Validation of Computer Protocols", Prentice-Hall, 1991.

[Knuth 73] D.E.Knuth, "The Art of Computer Programming: Sorting and Searching", vol 3, Addison-Wesley, 1973.

[Knott 75] G.D.Knott, "Hashing Functions", The Computer Journal, vol 18, n°3, August 1975, p265-278.

[Lum 71] V.Y.Lum, P.S.T.Yuen, M.Dodd, "Key-to-Adress Transform Techniques : a Fundamental Performance Study on Large Existing Formatted Files", Communications of the ACM vol14 n°4, April 1971.

[West 86] C.H.West, "Protocol Validation by Random State Exploration", PSTV-VI, Montréal - Canada, June 1986.

[Wolper 93] P.Wolper, D.Leroy, "Reliable Hashing without Collision Detection", CAV, Elounda - Greece, June 1993.

[Zhao 86] J.Zhao, G.Bochmann, "Reduced Reachability Analysis of Communication Protocols: a New Approach", PSTV-VI , Montréal - Canada, June 1986.