Differential Hashing Functions : Application to Reachability Graph Generation

Bernard Cousin

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

This paper presents the differential hashing functions. A differential computation process enables hashing function processing time to be optimized. After a formal definition of the differential property for hashing functions, we show that not all hashing functions have this property, then we propose a characterization of the differential hashing function set. Next, we show the performance acceleration produced by the differential algorithms applied to five hashing functions found in common applications. The observed accelerations can be significant because they are proportional to key length. Last we study the performances of a differential hashing function for the reachability graph exploration of distributed systems specified by a Petri net. This application demonstrates the advantages and the limits of our differential technique.

1: Introduction

Complex and safety-critical systems such as distributed systems with their communications protocols and services, require the development of verification tools based on formal description techniques. These tools enable the complexity arising from the parallelism of such systems to be controlled. To describe the complex behavior of the specified system, graphs are generated by the verification tools. Numerous graphs have been introduced: reachability graphs [16], reduced graphs [15], [10], symbolic graphs [5], colored and high level graphs [12], [20], and stochastic graphs [9], etc.

The state space explosion inherent to the parallelism of the studied systems induces constant research: in particular to reduce the duration of the graph generation process and to increase the relevance and the density of the graphs [7]. We distinguish three main methods to deal with the state explosion. The first method proposes generation of specific graphs which enable only some restricted classes of properties to be studied [21], [4]. The second method proposes partial exploration of the graph using simulation technique like random or user control state exploration [24], [11]. The third method promotes data compression techniques and processing time optimizations [1], [13]. Obviously the three methods can be usefully combined. This paper describes an enhanced graph generation technique which belongs to the third method. However this technique does not only apply to distributed system verification; the chosen application example will exhibit the advantages and the restrictions of the proposed technique.

Hashing methods are searching methods used to accelerate the retrieval process of an element among a large set. Every element of the set is uniquely identified by its key. Hashing methods are built upon one hashing function and one collision resolution function. The hashing function transforms each key into one hashing value. The key definition domain is usually very large while the hashing value definition domain is smaller. The hashing value is used as an address to access a memory area where the element associated with the key is stored. If the storage area is structured as an array the hashing value is used as a table index. Due to the reduction of the definition domain the hashing function can associated several keys to a unique hashing value. The collision resolution function is in charge to allocate a specific storage location to each element.

In this paper, we describe differential hashing functions. These hashing functions use differential computation processes of the hashing value which can replaced the usual computation processes, and which can optimized the processing time. The obtained optimization is based on the following observation: the structure of the keys can be regarded as a record of items. The physical characteristics of items are determined so as to ease the computation of the differential process. 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.

We associate a mono-differential computation function family to the differential computation process. These functions enable the hashing value of a key to be computed from the hashing value of another key which differs from the previous one in only one item. We prove easily that, by function composition, the mono-differential function family enables the hashing value of a key which differs from another in several items to be computed.

A preliminary question is raised by our proposition. Is it possible to associated a differential process to every hashing function? In the second section, after a formal description of differential hashing functions, we prove that the answer is no. Nevertheless, we produce a characterization of the hashing function set which can be associated with a differential computation process. Afterwards we show that this set contains the majority of hashing functions used.

The third section answers to the following question: Are the differential computation processes always more efficient than the usual computation processes? We show that this is not always the case, but we establish the conditions which are required to ensure that efficiency. However this is not a final answer, because the algorithms used during the implementation of the differential and the usual processes have the preponderant influence on the performances. Nevertheless we show that naive implementations product very good acceleration of the processing speed, and multi-differential implementations, enabling very efficient coding, cut the processing time significantly.

0-8186-4212-2/93 $03.00 © 1993 IEEE

The fourth section gives the performances achieved by the differential hashing function used by the Bouster verification tool [3], based on formal description of distributed systems by Petri net. This reachability graph generation example allows us to exhibit the advantages and the limits of our differential technique.

2: The differential functions

2.1: Definition

Hashing function can be described as application with N variables from a product of set $\prod_{k=1}^{N} A_k$ to a set B:

$$\forall <x_1,...,x_i,...,x_N> \in \prod_{k=1}^{N} A_k,\ \exists!\ h(<x_1,...,x_i,...,x_N>) \in B.$$

We denote $h(<x_1,...,x_i,...,x_N>)$ the hashing value of the key $<x_1,...,x_i,...,x_N>$.

Definition 1: 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 as

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

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 the differential item). Knowing the value of the differential item of the father_key (y_i), the new value that this differential item must have in the son_key (x_i), and the hashing value computes from the father_key ($h(<x_1,...,y_i,...,x_N>)$), the mono-differential function associated to the differential item enables the computation of the hashing value associated to the son_key ($h(<x_1,...,x_i,...,x_N>)$).

We denote $\mathbf{P}(h,i)$ the set of F_i functions which respects this definition for the hashing function h.

We notice that the father/son relation is symmetrical, since the same function F_i enables the hashing value associated to the father_key knowing those of the son_key to be obtained.

Definition 2: 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: $\forall i \in [1,N],\ \exists\ F_i \in \mathbf{P}(h,i)$.

If a hashing function has a complete differential computation function family, we shall say that it is differential.

2.2: Characterization

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

All hashing functions do not admit differential computation functions. For instance, the hashing function "⊗" builds over a binary set product and defined as the binary operator "logical And" is not differential. In fact, the mono-differential computation functions can not 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 is in opposition to the function definition which ensures the uniqueness of the image.

Property:

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

$$\forall <x_1,...,x_i,...,x_N> \in \prod_{k=1}^{N} A_k,), \forall <y_1,...,y_i,...,y_N> \in \prod_{k=1}^{N} A_k,),$$
$$h(<x_1,...,x_i,...,x_N>) = h(<y_1,...,x_i,...,y_N>) \Leftrightarrow$$
$$h(<x_1,...,y_i,...,x_N>) = h(<y_1,...,y_i,...,y_N>). \quad (2)$$

The proof that the property (2) is necessary and sufficient condition in order that the hashing functions admit a complete differential computation function family can be found in [6].

2.3: Example

For instance, take the simple hashing function h defined as the "exclusive or" between all the key items. Each item belongs to the same set B. Formally

$$\forall <x_1,...,x_i,...,x_N> \in \prod_{k=1}^{N} B,\ h(<x_1,...,x_i,...,x_N>) = x_1 \oplus ... \oplus x_i \oplus ... \oplus x_N$$

with ⊕ the usual "exclusive or" operator from B to B.

The mono-differential hashing computation functions F_i can be defined as $F_i(h(<x_1,...,y_i,...,x_N>), y_i, x_i) = h(<x_1,...,y_i,...,x_N>) \oplus y_i \oplus x_i$.

We can prove the property (2) which demonstrates that the hashing function h is differential.

Assuming the following initial hypothesis:

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

By mono-differential hashing function definition:

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

By definition of F_i:

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

By initial hypothesis:

$h(<x_1,...,y_i,...,x_N>) = h(<y_1,...,x_i,...,y_N>) \oplus x_i \oplus y_i$

By definition of F_i:

$h(<x_1,...,y_i,...,x_N>) = F_i(h(<y_1,...,x_i,...,y_N>),\ x_i,\ y_i)$

By the mono-differential hashing function definition, we prove that we obtain the final assertion:

$h(<x_1,...,y_i,...,x_N>) = <y_1,...,y_i,...,y_N>$ (qed).

3: Performance study

3.1: Presentation

We are going to show in this section the increase in performance we can achieve using a differential computation process. To do so, we compare the performances obtained by

hashing functions using usual algorithms against differential algorithms.

We can find numerous performance studies on hashing methods [19], [17], [23], [22]. Our study focuses on the performances of the hashing functions without studying the collision resolution functions. Actually our technique can be used without any problem with all collision resolution functions, so the performance increasing obtained with differential hashing algorithms should be entirely preserved.

Five hashing functions have been chosen among those found in the literature. This sample does not cover all existing hashing functions, but these 5 functions cover a large spectrum of hashing functions commonly used. We do not hide the fact that the utilization frequency is not the only criterion which has intervened in the choice of these 5 functions, the coding simplicity of the usual algorithm of these hashing functions, which can provided good performance, has intervened also as criterion. In fact, a simple algorithm with short code is often faster than a complex and long algorithm. Some further studies are undertaken to research the differential algorithms of some other hashing functions, so increasing the study spectrum.

workstation with 16 Mbytes main memory. The performance measures of the five hashing functions have been collected in two tables: The first table collects the performance results of the usual algorithm [cf Table 1], the second table collects the results obtained by the differential algorithm assuming the median key item has been modified [cf Table 2]. Afterwards, we shall see that some hashing functions have differential computation process time which can depend on the index of the differential item: the median item is average choice. The time unit of the performance table is $1/60.10^{-6}$ second.

3.2: Preliminary discussion

We should like to established some facts about the different methods used by the five hashing functions before comparing the two algorithms types (usual and differential). At first we state that all the functions but the function number three have an increasing process time depending on the length of the key (cf Table 1). In fact, this third function uses fixed bit position extraction method such that the computation duration of the hashing value is constant. It is also the shortest time computation function. We note that the

Key length	1	2	4	8	16	32	64	128	256	512	1024	2048
Function #1	153	266	210	271	396	1089	1156	2175	4189	8190	16326	31958
Function #2	395	489	679	1039	1883	3635	7202	14389	28663	61917	129112	263277
Function #3	172	176	178	178	179	178	178	180	179	178	179	180
Function #4	304	704	1326	2266	3967	7985	18941	38516	79407	171301	419312	899450
Function #5	277	301	559	1126	1337	1598	2623	4890	9284	17688	35056	68872

Table 1: Usual algorithm performance

Key length	1	2	4	8	16	32	64	128	256	512	1024	2048
Function #1	113	107	114	113	113	113	114	114	113	114	113	113
Function #2	164	165	164	164	164	164	164	167	254	165	164	165
Function #3	111	128	137	127	128	129	128	127	128	129	129	131
Function #4	358	358	357	451	765	1088	1635	2350	4009	7094	13340	25863
Function #5	381	385	201	201	413	290	289	502	345	346	558	348

Table 2: Differential algorithm performance

The proposed hashing functions use the following basic operators: division/product, modulo, addition/subtraction, folding, bit extraction [19]. Others operators can be used: power/root, radix transformation, polynomial computation, etc. These other operators have long computation time, so we choose to not use them. Good hashing functions spread the hashing values over their definition domain to obtain low collision probability. So basic operators can use multiplicative constants chosen among prime numbers. The five proposed hashing functions combine several basic operators.

The first function (#1) uses a folding method based on the logical operator "exclusive or". The second function (#2) sums all its items. The third function (#3) uses extraction method of some bit groups among the bit suite making the key. The forth function (#4) is a weighted sum of the key items by prime constants. The last function(#5) called "hpjw" combines several logical operators like addition and rotation.

We succeed in finding the differential algorithms of all five functions. The functions have been coded in C language. Their codes can be found in [6]. To simplify the performance tests and without loss of generality, every key item is a byte. These performance tests have been executed on a Sun SPARC

hashing value distribution over the definition domain of this third function is unequal. In fact hashing functions must use all the key items, if they are significant, to compute a hashing value.

If the other hashing functions have similar behavior (computation time increase according to the key length) the computation time ratio for the same key length vary from 1/2 for the short keys to more than 1/20 for the longest keys. So the faster a hashing function is the shorter its computation time is for the long keys. That justifies our a priori choice of five simple, and consequently fast, code hashing functions.

We recall that the recorded times are not the overall performance times of the hashing methods, because hashing methods are based on a hashing/collision resolution function pair. We emphasize that an inadequate hashing function can generated 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 resolution collision function duration is independent of the key length, so it becomes to be negligible for the long keys compared to the computation time of the hashing functions.

We have made these performance tests over numerous versions of several hashing functions, in particular the first

and the third functions: word length computation (4 bytes item), modification of the overflow process, arithmetical to logical operator substitution, etc. The previous described behaviors have been maintained, even if some local optimizations have been measured, so we have chosen only one version.

3.3: Results

If now we compare the results obtained from the differential algorithm to the usual algorithm, we verify that the processing time of the differential algorithm is, on one hand, shorter than the processing time of the usual algorithm, on the other hand, constant with respect to key length, with the exception of the hashing function number 4 (cf Table 2). In fact, the differential algorithm used by this hashing function needs the power computation of the product of the differential item and its associated prime constant with an exponent equal to the index of the differential item. The power computation is a time expensive process which increases according to the exponent, i.e the location of the differential item.

The influence of the differential item location can be studied over all the hashing functions and their differential algorithms. This influence is very low over all the functions except the hashing function number 4, as established in the previous paragraph. The remaining functions, although their absolute durations are short, have some erratic variations (especially for the function number 5). In fact, theses variations are generated by either the coincidence or not coincidence with some constants used during the 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 computation duration 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 ratio between the differential algorithm durations and the usual algorithm durations, enables the number of item differences that the differential algorithm must not reach to be better than the usual algorithm, to be established. The recorded values during the performance test establish that, for keys longer than one hundred items, the differential algorithm is faster than the usual algorithm as soon as the number of differential items is less than the following values:

function #1	function #2	function #3	function #4	function #5
≤15	≤66	<2	≤12	≤8

The application proposed as an example in the next section has a mean difference ratio (2 to 10% from large to medium size graphs) which is in concordance with the difference ratio required by the hashing function used. This will give the explanation of the good results measured.

We notice that some functions (especially the function #1) can have multi-differential computation functions which duration is close to the mono-differential computation duration. This is due to preprocessing technique which can be used if the differential key can be obtained with no additional computation. It is the hashing function with this preprocessing technique which is used in the next section.

4: Application to reachability graph

The graph states need a very large storage area. In fact, the size of the graphs and the states depends on parallelism and accuracy of the system model. Our study leads to establish that real distributed system or protocol models have states whose size is significant: 1916 bytes for P-channel protocol [8]; hundreds of bytes for Holzmann [14]; or from our own experiments several hundred of bytes (Transport class 4 protocol).

Some graph generation tools prefer to use search method where the data storage structure is a binary tree or specific hierarchical structure [5]. These methods are not the most often used because either they have very high specificity (they are unsuitable for all sorts of property verification) or they present an overhead in space (indirection pointer) and in time (link processing) as opposed to the hashing method.

The performance tests have been done using the Bouster validation tool [3] on a set of several system models described by means of Petri net. These models have various number of places (2 to 50000), various numbers of transitions (1 to 50000), various numbers of arcs (1000 à 100000) and they generate graphs with several hundred thousand states. Using the Unix profiler tool, our study established that the hashing function ("f_hashing()"), the collision resolution and access function ("put_in_table()") and the string compare function ("bcmp()") are the most time intensively used functions: each of these three functions consume 15 to 30 % 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).

These results exhibit at the same time the importance and the limits of the gain which we can hope to achieved with our differential method. In fact, the processor spend about 40% 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. The two other main methods mentioned in the introduction (data densification and partial exploration) can be combined to advantage with our differential method, and this possibility will be our next study.

These optimistic conclusions must not hide an important phenomena already raised by numerous performance researchers before us: 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.

The string compare function "bcmp()" also comes from the Unix standard library. As it belongs to the three most used functions, we propose to study new optimizations, first to evaluate new comparison algorithms [18], [2], and second to use graph compression techniques [13] which enable the string compare function to be omitted.

5: Conclusion

The performance results establish that the differential hashing speed-up increases with the key length. In fact, usual hashing functions use all the 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. The application hypothesis (very large graph, huge state number) generate long key lengths, as corroborated by numerous examples. The differential algorithm complexity is proportional to the differential item index between the original key and new key. The chosen application generates distributed system graphs which have inherently successive states whose keys have few differential items (less than 10%). This fact is in perfect conformance with the behavior of the modeled systems: in a distributed system typically all the constituting subsystems do not evolve simultaneously and at every moment.

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 study establishes that, first, all the studied hashing functions can be associated to a complete mono-differential function set, second, for the majority of differential computation processes time is shorter than for the usual one, third, differential techniques require applications where the differential items can be obtained at low cost (no need of differential item researching). The proposed application has all these prerequisite characteristics, and consequently enables a substantial improvement of performances.

Numerous extensions to this work can be considered: optimization of existing differential algorithms; development of differential algorithms for new hashing functions; adaptation of hashing functions to new graphs, in particular reduced graphs; and searching of new applications for the differential hashing method.

References

[1] B.Algayres, & all, "Vesar: a pragmatic approach to formal specification and verification", Computer Network and ISDN Systems, vol 25 n°7, February 1991.

[2] R.S.Boyer, J.S.Moore, "A fast string searching algorithm". Communications of the ACM vol 20 n°10, 1977.

[3] C.Campergue, C.Nouaille, "Bouster : génération parallèle du graphe des marquages accessibles associé à un réseau de Petri", rapport interne ENSERB, Bordeaux-France, Juin 1992.

[4] L.Cacciari, O.Rafiq, "On improving reduced reachability analysis", 5th international conference on formal description techniques, Lannion-France, 13-16 octobre 1992.

[5] G.Chiola, "Symbolic reachability graph", 11th international workshop on theory and applications of Petri nets, Paris-France, 1990.

[6] B.Cousin, "Les fonctions de hachage différentielles : application à la génération de graphe d'états", Rapport de recherche INRIA n°275, Rennes - France, 15 juin 1993.

[7] D.D.Dimitrijevic, M.S. Chen, "Dynamic state explosion in quantitative protocol analysis", 9th international symposium on Protocol Specification, Testing and Verification (PSTV-IX), Twente-Netherland, 6-9June 1989.

[8] L.Doldi, P.Gauthier, "Veda-2: power to the protocol designers", 5th international conference on formal description techniques, Lannion-France, octobre 1992.

[9] C.Dutheillet, S.Haddad, "Regular stochastic Petri nets", On Petri nets and its applications, Bonn - Germany, 1989.

[10] A.Finkel, C.Johnen, "Construction efficace du graphe de couverture minimal : application à l'analyse de protocole", Colloque francophone sur l'ingéniérie des protocoles (CFIP'91), Pau-France, Septembre 1991, Hermès, 1991.

[11] R.Groz,"Unrestricted verification of protocol properties on simulation using an observer approach", 7th international workshop on Protocol Specification, Testing and Verification , Montréal - Canada, June 1986. North-holland.

[12] P.Huber, A.M.Jensen, L.O.Jepsen, K.Jensen, "Towards reachability trees for high-level Petri nets", Lecture notes in computer sciences n°188, p215-233, Springer Verlag, 1985.

[13] G.J.Holzmann, "An improved protocol reachability analysis technique", Sofware, practice and experience, vol 18 n°2, February 1988, p137-161.

[14] G.J.Holzmann, "Design and validation of computer protocols", Prentice-Hall, 1991.

[15] M.Itoh, H.Ichikawa, "Protocol verification algorithm using reduced reachability analysis", Transactions of the Institute of Electronic and Communication Engineers of Japan, vol E66 n°2, February 1983.

[16] R.M.Karp, R.E.Miller "Parallel program schemata", Journal of comp. and system sciences, vol 3 n°4, May 1969.

[17] D.E.Knuth, "The art of computer programming: sorting and searching", vol 3, Addison-Wesley, 1973.

[18] D.E.Knuth, J.H.Morris, V.R.Pratt, "Fast pattern matching in strings", SIAM Journal on Computing, vol6 n°2, June 1977, p323-349.

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

[20] M.Lindquist, "Parametrized reachability trees for predicate/transition nets", 11th international conference on application and theory of Petri nets, Paris-France, 1990, Springer-Verlag, p22-41.

[21] M.A.Marsan, G.Chiola, "Improving efficiency of analysis of DSPN models", LNCS n°424, Advances in Petri nets, Springer-Verlag, 1989.

[22] R.Morris, "Scatter storage techniques", Communication of the ACM Vol 11 n°1, January 1968, p38-44.

[23] J.D.Ulmann, "A note on the efficiency of hashing functions", Journal of the ACM vol 19 n°3, July 1972.

[24] C.H.West, "Protocol validation by random state exploration", 6th international workshop on protocol specification, testing and verification (PSTV-VI), Montréal - Canada, June 1986.