Incremental Hashing for SPIN

Viet Yen Nguyen[1] and Theo C. Ruys[2]

[1] RWTH Aachen University, Germany.
http://www-i2.cs.rwth-aachen.de/~nguyen/

[2] University of Twente, The Netherlands.
http://www.cs.utwente.nl/~ruys/

Abstract. This paper discusses a generalised incremental hashing scheme for explicit state model checkers. The hashing scheme has been implemented into the model checker SPIN. The incremental hashing scheme works for SPIN's exhaustive and both approximate verification modes: bitstate hashing and hash compaction. An implementation has been provided for 32-bit and 64-bit architectures.
We performed extensive experiments on the BEEM benchmarks to compare the incremental hash functions against SPIN's traditional hash functions. In almost all cases, incremental hashing is faster than traditional hashing. The amount of performance gain depends on several factors, though.
We conclude that incremental hashing performs best for the (64-bits) SPIN's bitstate hashing mode, on models with large state vectors, and using a verifier, that has been optimised by the C compiler.

1 Introduction

An explicit state model checker is a model checker where all states are explicitly represented in the state space. Explicit model checking is sometimes called stateful state space exploration, especially when checking reachability or safety properties (e.g. deadlocks, assertion violations).

Central to stateful state space exploration is the process of state matching: for every encountered state, it should be checked whether the state has already been visited or not. As the run-time of exploration is linear in the number of transitions (i.e. the amount of newly encountered states and re-visited ones), it is obvious that state matching should be as fast as possible. Typically, hash tables are used to store states. Upon exploration of each state, the hash table is consulted to check whether that state has already been explored or not.

The access to a hash table is through a *hash function*. Given a key k, a hash function h computes the hash code $h(k)$ for this key. This hash code $h(k)$ corresponds to the address in the hash table, where this key k should be stored. For model checking, this k is typically the (binary) representation of a state, called the *state vector*. Most *traditional hash functions* compute $h(k)$ by considering all elements of k. For example, if k is a string, a typical hash function h would compute $h(k)$ on the basis of all individual characters of k.

With respect to state space exploration, two observations can be made. Firstly, the size of a state vector is usually substantial. State vectors of several hundreds of bytes are not exceptions. This means that computing a traditional hash code for such states can become quite expensive. Secondly, when exploring the state space in a structured manner (e.g. depth first search), the transitions between two consecutive states is local: only a small part of the state changes with respect to the previous state.

This last observation is the idea behind so called *incremental hash functions*, which use the hash code of a previous key to compute the hash code for the new key. The application of incremental hashing within a model checker is not new. Mehler and Edelkamp [11] have implemented an incremental hashing scheme in the model checker StEAM, a model checker for C++. However, their incremental hashing scheme is only practicable for hashing (large) stacks and queues incrementally. We have improved their hashing scheme by generalising it for hashing vector-based data structures (like state vectors) incrementally by using cyclic polynomials from [2].

Our generalised hashing scheme was originally developed for MoonWalker [14][3] a software model checker for CIL bytecode programs, i.e. .Net applications. Unfortunately, after implementing our incremental hash function into MoonWalker, initial tests showed no measurable performance gain. We studied this observation using a profiler and found out that the stake of hashing in MoonWalker is extremely small [12]. Other tasks that have to be performed for each state (e.g. garbage collection, state compression, etc.) take much more time. Any performance gain in hashing would therefore not be visible in the total running time.

The model checker Spin [6] is arguably one of the fastest explicit state model checkers available. The current version of Spin uses two traditional hash functions: one composed by Bob Jenkins [20] and one composed by Paul Hsieh [19]. For Spin verifiers – unlike for bytecode model checkers – hashing accounts for a substantial amount of the running time.

We have implemented our generalised incremental hashing scheme into Spin 5.1.4. The incremental hashing scheme works for checking safety properties (`-DSAFETY`) for both 32-bit and 64-bit architectures. Furthermore, it works in exhaustive mode and both approximate modes: bitstate hashing and hash compaction. We have performed numerous experiments on the BEEM benchmarks [16] to compare the incremental hash functions against Spin's traditional hash functions. From these experiments we learnt that incremental hashing is faster than Spin's traditional hash implementations without sacrificing too much accuracy. The amount of performance gain depends on several factors:

- the verification mode: exhaustive or approximate,
- the architecture on which the verification is run: 32-bit or 64-bit,
- the size of the state vector, and

[3] MoonWalker was previously known as mmc: the Mono Model Checker. Due to several name clashes, mmc has recently been renamed to MoonWalker.

– the optimisation parameters used for the gcc compiler: the default setting (`-O0`) or the most aggressive optimisation setting (`-O3`), and
– the verifier arguments, e.g., hashtable sizes, maximal search depth.

Incremental hashing performs best for (64-bit) bitstate hashing mode, with larger state vectors and using an optimised verifier.

The rest of the paper is organised as follows. In section 2, we first discuss some related work in the context of (incremental) hashing. Section 3 presents the general incremental hashing scheme: both the intuition of the method and an implementation in C are discussed in detail. Section 4 explains how we have implemented the method in Spin and discusses the experimental settings of the benchmark runs that we have conducted. In section 5 we present the results of the benchmark runs and we discuss the outcome of the experiments. Finally, in section 6 we summarise the results and give pointers for future work.

2 Related Work

The hash table is the cornerstone of stateful state space exploration. Accesses to a hash table are in amortised $O(1)$ time. Although this is a good worst-case time-complexity, the constant costs are high if a bad hash function is chosen. A good hash function should fulfill the following requirements [10]:

– *Fast.* The computation of the hash function should be efficient and fast.
– *Accurate.* To avoid a large number of address collisions, the hash values should distribute evenly over the range of the hash function.

With respect to accuracy, the following rule of thumb is often used: "one bit change in the key should result in half of the bits flipped in its hash code".

A well known hash function for hashing arrays is the rolling hash function [1]. Given a ring R, a radix $r \in R$ and a mapping function T that maps array elements to R, the rolling hash code for an array $a = a_0 \dots a_n$ is computed as follows:

$$H(a) = T(a_0) + rT(a_1) + \ldots + r^n T(a_n) = \sum_{i=0}^{n} r^i T(a_i) \tag{1}$$

A possible suitable ring R is $\mathbb{Z}/B$, where B is a prime number and is also the amount of buckets in the hash table. This specialisation of the rolling hash function is called hashing by prime integer division [2].

It is not difficult to see that the rolling hash function is prone to overflow, especially due to the power operations with the radix. Remedying overflow is costly. A recursive formulation of the rolling hash code is less prone to overflow:

$$H(a_0) = T(a_0) \tag{2}$$

$$H(a_i) = rH(a_{i-1}) + T(a_i) \qquad 1 \leq i \leq n \tag{3}$$

Note that the radixes are reversely mapped to the array elements when compared to equation 1, and therefore hash codes derived from the recursive formulation should not be matched against hash codes derived from the non-recursive formulation.

2.1 Incremental Hashing

Karp and Rabin [8] describe an incremental recursive hash function for fast string pattern matching by using recursive hashing by prime integer division. Their idea is to reuse the hash code of the previous unmatched substring for the calculation of the shifted substring. In [2], this is generalised for matching of n-grams.

The rolling hash function is not only amenable for incremental recursive hashing, but also incremental linear hashing. The idea behind incremental linear hashing, is that the contribution of an array element is independent of the contributions of other array elements. In case of an array change, the influence of the old array element is known and thus can be removed, followed by adding the influence of the new array element [2]. In [11], this is expressed as follows. Consider an array $k = v_0 \dots v_i \dots v_n$ and its successor $k' = v_0 \dots v'_i \dots v_n$, then the hash code of k' can be computed as follows:

$$H(k') = H(k) - r^i T(k_i) + r^i T(k'_i) \tag{4}$$

Depending on the ring chosen, the power operation with a large index i can easily lead to overflow. Thus using this hashing scheme for arbitrary modification of large arrays is impractical. For stacks and queues however, [11] describes a rewritten version of that formula for push and pop operations with the power operation removed. They tested it in their StEAM model checker, and got at least a speedup factor by 10 compared to non-incremental hashing. Note that this speedup was achieved with fixed-sized stacks of eight megabyte. It is logical to assume that the speedup factor will be much lower with with arbitrary sized stacks.

2.2 SPIN

Spin [21, 6] is a state-of-the-art explicit state model checker, which is used as a reference for other model checkers. With respect to memory efficiency and verification time, Spin is hard to beat. Spin provides many ways to tune and optimize its verification runs. Most of these optimisation features can be enabled via compilation flags for the C compiler (e.g. gcc). See for details chapter 18 of [6].[4]

Spin supports three verification modes. The most commonly used verification mode is the *exhaustive* mode, where all states are stored until the memory to store the states is exhausted. Spin provides state compression techniques to fit more states in the same amount of memory.

For the cases where there is not enough memory to store all (compressed) states, Spin supports two lossy, approximate verification modes, which are both heavily based on hashing functions: bitstate hashing and hash compaction. A good survey and extensive discussion on various approximate methods can be found in [10].

[4] As this paper is concerned with tuning and optimizing Spin verification runs, we use several of these compilation parameters, usually prefixed with `-D`.

Bitstate hashing. Holzmann's bitstate (sometimes called supertrace) hashing [4, 5] algorithm works as follows. Under the assumption that the number of slots in the hash table is high in comparison to the number of reachable states, then it is possible to identify a state by its address in the hash table. In this case, a single bit suffices to indicate whether a state has already been visited or not.

The coverage of bitstate hashing can be improved (on the expense of time) by using k different, independent hash functions for the same hash table. A state is considered visited before if the bits for all k hash functions are set. This variant of bitstate hashing is called k-fold bitstate hashing. Triple hashing [3] improves upon this scheme by using three hash values to generate k hashes. For bitstate hashing to be effective, the accuracy of the hash function(s) involved should be high.

Initially, SPIN used $k = 2$ by default, but since October 2004, SPIN's default is set to $k = 3$. Of course, the variable k can be set to larger values using a run-time option.
Bitstate hashing in SPIN is enabled using the `-DBITSTATE` parameter.

Hash compaction. Wolper and Leroy [15] introduced hash compaction, a indirection variant of bitstate hashing. The idea of hash compaction is to store the addresses of the occupied bit positions in the bitstate table, rather than storing the whole array itself.

For hash compaction, the hash table is taken to be very large (e.g. 2^{64} bits), much too large to fit in memory. Now the address computed by the hash function (e.g. 64 bit-wide) is stored as being a normal state. Hash compaction is thus also viewed as a lossy form of state compression. Hash compaction is more accurate than k-fold bitstate hashing (for small k).
Hash compaction in SPIN is enabled using the `-DHC` parameter.

The current versions SPIN uses two traditional, linear hash functions:

- *Jenkins.* Since long, SPIN uses Jenkins hash function [20, 7] for both its exhaustive and approximate verification runs. Jenkins' hash function is considered a fast but still quite accurate hash function.
  For bitstate hashing, SPIN uses 96-bit and 192-bit versions of Jenkins' hash function. For exhaustive verification, a part of the 96-bit or 192-bit hash value is used.
- *Hsieh.* Since version 5.1.1 (Nov 2007), SPIN has adopted an alternative hash function by Hsieh [19]. Although perhaps not as accurate as Jenkins, Hsieh's hash function can in some cases be much faster.
  Hsieh's hash function can be enabled with the parameter `-DSFH`, which stands for 'Super Fast Hash'. But Hsieh's hash function is also automatically selected for 32-bit safety runs (`-DSAFETY`). To speed up such verification runs even further, SPIN's default mask-compression of states is disabled for `-DSAFETY` runs as well. Hsieh's hash function is not only fast, but its implementation is also suitable for aggressive optimisation by the GCC compiler (i.e. using `-O2` or `-O3`).

Currently, there only exist a 32-bit version of Hsieh's hash function. Furthermore, in the current version of SPIN, Hsieh's hash function can *only* be enabled for checking safety properties.

3 Generalised Incremental Hashing Scheme

This section presents the intuition behind the incremental property, the time-complexity of the incremental hashing scheme and implementation variants. A few concepts, like polynomial rings, from field theory are used to express this. Readers unfamiliar with this may consult [2, Appendix A].

3.1 Incremental Property

Consider a Galois field (also known as a finite field) $R = GF(2)[x]/(x^w + 1)$, the ring consisting of polynomials in x whose coefficients are 0 or 1, reduced modulo the polynomial $x^w + 1$. Make sure that w matches the computer's word size, thus 32 for 32-bits words. The polynomials are represented by w-sized bitmasks by placing the coefficients of x^i at the i^{th} bit, creating an one-on-one correspondence between polynomials in R and the bitmasks.

As a radix, the polynomial $x^\delta \in R$ is chosen. By setting radix $r = x^\delta$, the following incremental hash function is derived from equation 4:

$$H(k') = H(k) + x^{\delta i} T(k_i) + x^{\delta i} T(k'_i) \tag{5}$$

The minus operation from equation 4 is replaced by an +, because addition and subtraction are the same in ring R. Now, consider an arbitrary member $q \in R$ with $q(x) = q_{w-1}x^{w-1} + q_{w-2}x^{w-2} + \ldots + q_0$. The multiplication of the x and $q(x)$ is the following:

$$xq(x) = q_{w-1}x^w + q_{w-2}x^{w-1} + \ldots + q_0 x \tag{6}$$

$$= q_{w-2}x^{w-1} + q_{w-3}x^{w-2} + \ldots + q_0 x + q_{w-1} \tag{7}$$

Equation 7 is equation 6 reduced to modulo $x^w + 1$. The multiplication by polynomial x results to a left rotate of the coefficients in $q(x)$, hence the name cyclic polynomials. For most platforms, this is an efficient operation. At least all x86-architectures include native bit rotate instructions. Additions in equation 5 can be implemented using a exclusive-or operation, which is available on all processor platforms.

In order to reduce the amount of operations, equation 5 can be rewritten by applying the associativeness of the + operation, as shown in the next equation:

$$H(k') = H(k) + x^{\delta i}(T(k_i) + T(k'_i)) \tag{8}$$

So far, only one variable is left unmentioned, namely δ. The choice of a δ for the radix x^δ was experimentally evaluated by [2]. No δ clearly stood out. For $\delta = 1$, the incremental hashing function worked well and they used is subsequently for their experiments. For this reason, we also take 1 for δ.

Furthermore, as described in [2], cyclic polynomials have one weakness. They have a cycle length of size w for which it computes the hashcode of zero. For example, if a key of size $2w$ starts with w elements followed by another identical sequence of w elements, then the hashcode for that key is zero. In practise, such keys are extremely rare in model checking, as their size must be exactly nw-sized, where $n \in \mathbb{N}$, and that its contents should be also w-cyclic as well.

3.2 Time-complexity

The time-complexity of the incremental hashing scheme is differently defined compared to traditional hash functions. A fast traditional hash function has a time-complexity in $O(N)$, where N is the array length. The incremental hash function hash function has a time-complexity of $O(1)$ for one change to the array. Theoretically, the incremental hash function is faster if the amount of changes between successive states is smaller than N. This is usually the case in model checking, where the amount of changes is usually 1 or 2 and almost never near N.

3.3 Variants

From the perspective of implementation, there are several variants of the incremental hashing scheme at one's disposal, namely by reordering the coefficients with respect to the bitmask and by using different mappings of function T in equation 4.

Reordering the Bitmask. The coefficients of polynomials in R were initially mapped to a bitmask whose position coincide with those in the bitmask. This mapping was chosen to allow efficient left-rotate operations on bitmasks as the equivalent to multiplication by x. Another ordering of coefficients that works equally well is by ordering the coefficients reversely: the coefficient of x^j is placed at the $64 - j$ bit position. Such a mapping results to right-rotate operations as the equivalent to multiplication by x.

Different Mappings. Our initial experiments with the incremental hashing scheme led to high collision rates. This was caused by the initial mapping of function T in equation 5, for which we originally chose the identity function. The source of the collisions lied in the entropy of changes between state vectors of successive states. Transitions are often of low entropy, like changing changing a variable from 0 to 1 or add 1 upon variable `i`. The incremental hash function recalculates the hash function upon such changes, but since the entropy is low, the resulting hash would not differ much as one desires for a good hash function. In order to increase entropy, we experimented with different functions of T.

Our approach is by using integer hash functions as a T. We initially used Wang's integer hash [22], but its constant time-complexity is relatively big compared to that of the incremental hashing scheme, and we observed in experiments that the slowdown made incremental hashing slower than traditional hashing functions.

The function T has therefore be very fast. An integer hash function for which we observed that it works out well is Knuth's multiplicative constant [9]. This hash function simply multiplies the input by a word-size dependent constant. For 32-bit words, the constant is 2654435769, and for 64-bits words, the constant is 11400714819323198485. The constant is calculated by multiplying one-bitmask (i.e., the largest number in the sized word) by the golden ratio $(\sqrt{5}-1)/2) \approx 0.618034$. In [9], Knuth shows this integer hash function has a high likelyhood of spreading the bits through the word, thereby increasing entropy.

Other constants are also applicable. We also experimented with the magic constants of the FNV hash funnction [18], which is 2166136261 for 32-bits words and 14695981039346656037 for 64-bits words. These FNV constants are 'magic', because their effectiveness was only evaluated by emperic evidence.

3.4 Implementation Examples

Here we present several C implementations of the incremental hashing scheme. The implementation below is one for SPIN:

```
c_hash(int i, unsigned int old, unsigned int new) {
    const unsigned long knuth = 11400714819323198485UL;
    const unsigned long fnv = 14695981039346656037UL;
    unsigned long diff = ((new)*knuth) ^ ((old)*knuth);
    chash ^= ((diff << i) | (diff >> (64 - i)));
#if defined(BITSTATE) || defined(HC)
    unsigned long diff2 = ((new)*fnv) ^ ((old)*fnv);
    chash2 ^= ((diff2 >> i) | (diff2 << (64 - i)));
    chash3 ^= ((diff >> i) | (diff << (64 - i)));
#endif
}
```

For exhaustive search, only Knuth's multiplicative constant with left-rotatable bitmask are used. For hash compaction, a second hash value is maintained using FNV's constant. For bitstate hashing, triple hashing is used by maintaining a third hash value using a right-rotable bitmask in combination with Knuth's multiplicative constant. We will refer to this implementation as `CHASH` (where the 'C' stands for cyclic).

Another triple incremental hashing approach is by viewing three words as one word, upon which a bit rotate is performed. This approach is less optimisable because no processor supports a native bit rotate operation for triple-word sized values. We experimented shortly with this approach but found out it always outperformed by the above variant.

4 Experimental Method

We originally implemented `CHASH` in Spin 4.3.0, and the results with it are described [12]. Since that thesis and this paper, version Spin 5.1.4 came out and we ported `CHASH` to it. We subsequently used this newer implementation and benchmarked it extensively against Spin's default hash functions.

4.1 Implementation

The difficulty of implementing `CHASH` varies from language to language. In MoonWalker, which is written in C#, the implementation was extremely easy due to object encapsulation of the state vector, and therefore also all writes calls to it.

Spin however is implemented in C and therefore lacks the expressive means for encapsulated state vector access. The state vector in Spin is accessible via the global point `now` and is updated by writing to an offset from this pointer. In order to detect all these writes, which can happen throughout the generated verifier, we had to add a call to the incremental hashing function just before the state vector is updated at that point.

Besides this, we had to overcome several other issues due to specifics in the C language. For one, our implementation uses the memory address of the written variable as the index argument to function `c_hash`. This however did not work for the Promela datatypes `unsigned int` and `bit`. Spin uses bitfields as the underlying C datatype, and bitfields have by definition no addresses. To solve this, we created a virtual memory mapping for unsigned ints and bits. When the verifier is generated, the address of the variable in the symboltable is used as the index instead. We could not use a virtual mapping for all variables because of arrays. Accesses to arrays in Spin may have an expression as indexer and its value is only known at runtime, not when the verifier is generated.

Also, we could not just use the memory addresses, but we had to use memory offsets. Using offsets is important for approximative methods, because the approximation of the explored state space can slightly differ due to changed memory addresses. These are suspectible to operating system semantics. Offsets are relative and remain the same between runs.

Additionally, we optimised the time-complexity at a small cost of memory. When the DFS search backtracks, `CHASH` has to be called for a reverse operation in order to calculate the correct corresponding hash code. However instead, we store the hash values on the DFS stack, and write this value as the corresponding hashcode.

4.2 BEEM Benchmarks

We used the BEEM benchmark suite for evaluating the effectiveness of the incremental hashing scheme. This suite consists of 57 models, ranging from communication protocols, mutual exclusion algorithms, election algorithms, planning and scheduling solvers and puzzles [13, 16]. The model are parameterised to yield different problem instances. The total amount of models is 298 and 231 of them are in Promela. We initially evaluated all the Promela models for our experiments. From this evaluation, we made a selection of the 40 largest models that did not run out of memory. These were subsequently used for comparing the different hashing configurations.

Due to space constraints, we are only able to present a selection of the results from these 40 models. We chose to present the ten best problem instances, the

ten worst problem instances and averages of the forty selected BEEM benchmark suite, thereby giving a nuanced perspective of the results.

4.3 Setup

We ran the benchmarks on nine identical nodes, each equipped with Intel Xeon 2.33 GHz processors and 16 GB memory. When compiling the models, we always enabled the `-DSAFETY` and `-DMEMLIM=15000`. As arguments to pan, we fixed the maximal search depth to $20 * 10^6$ and disabled stopping on errors and printing unreachable states. Furthermore, we conducted runs with the following compiler flags and pan arguments:

Compiler flags	Pan arguments
`-m32 -DHASH32 -DSFH`	`-w26`
`-m32 -DHASH32 -DSPACE -DNOCOMP`	`-w26`
`-m32 -DHASH32 -DCHASH -DNOCOMP`	`-w26`
`-m32 -DHASH32 -DSPACE`	`-w26`
`-m32 -DHASH32 -DCHASH`	`-w26`
`-m32 -DHASH32 -DBITSTATE`	`-w32 -k3`
`-m32 -DHASH32 -DBITSTATE -DCHASH`	`-w32 -k3`
`-m32 -DHASH32 -DHC`	`-w27`
`-m32 -DHASH32 -DHC -DCHASH`	`-w27`

These are 32-bits runs. Additionally, we also ran a series of 64-bits runs using the following compiler flags and pan arguments:

Compiler flags	Pan arguments
`-m64 -DHASH64 -DSPACE`	`-w28`
`-m64 -DHASH64 -DCHASH`	`-w28`
`-m64 -DHASH64 -DBITSTATE`	`-w36 -k3`
`-m64 -DHASH64 -DBITSTATE -DCHASH`	`-w36 -k3`
`-m64 -DHASH64 -DHC`	`-w29`
`-m64 -DHASH64 -DHC -DCHASH`	`-w29`

All configurations were run twice, namely without compiler optimisations (`-O0`) and with (`-O3`). We furthermore used the GNU profiler on all configurations. All these configurations come down to the total amount 2400 of verifications runs of which we captured their output, processed it and analysed it to present it in the next section.

5 Results and Discussion

Our benchmark runs generated extensive results which we cannot put all here. We therefore highlight the interesting observations and in case of interest, the full result set is downloadable from the incremental hashing webpage [17].

5.1 Exhaustive Verification

In 32-bits exhaustive verification we compared `CHASH` and Jenkins's against Hsieh's. The result is shown in table 1. The first column is the state vector size, followed by the state space size in 10^6 and transitions in 10^6. Collision rates are indexed against the state space. They can be higher than 100 because Spin

model	sv (bytes)	states (·10^6)	transitions (·10^6)	collrate Hsieh (%)	collrate Jen. (%)	collrate CHASH (%)	time -O3 Hsieh (sec)	time -O3 = 100% Jen. (%)	time -O3 CHASH (%)	time -O0 Hsieh (sec)	time -O0 = 100% Jen. (%)	time -O0 CHASH (%)
elevator_planning.2	48	11	93	939	51	434	47	81	79	54	87	66
firewire_link.5	404	6	12	3	3	3	24	104	86	34	105	61
adding.6	28	8	12	41	2	2	6	92	87	7	95	78
telephony.4	52	12	64	26	20	28	28	99	89	37	106	71
train-gate.3	136	20	57	16	16	15	49	101	90	73	102	67
schedule_world.3	44	4	44	105	21	20	18	101	90	26	102	75
phils.6	84	14	143	93	92	96	86	108	91	109	103	68
peterson.6	44	9	33	13	12	15	16	99	92	21	101	77
lann.3	128	5	24	8	8	11	24	98	94	33	104	77
fischer.6	56	8	33	17	11	11	17	102	94	24	101	76
...												
protocols.5	100	3	8	2	2	16	6	106	100	9	101	74
elevator2.3	40	8	55	166	22	185	20	105	101	28	99	73
driving_phils.4	84	11	30	6	6	106	15	103	102	22	104	72
elevator.3	140	19	70	25	24	37	74	108	103	112	103	79
lamport_nonatomic.4	180	16	60	19	19	82	69	98	105	96	101	73
msmie.4	100	7	11	2	2	2	10	103	106	16	104	79
bridge.2	60	9	27	12	12	12	21	103	107	36	102	89
sorter.4	60	13	27	11	8	23	19	102	112	28	103	87
reader_writer.3	160	1	4	1	1	1	16	99	126	29	101	94
Average	**98**	**12**	**47**	**52**	**17**	**47**	**30**	**102**	**97**	**43**	**102**	**75**

Table 1. 32-bits exhaustive search BEEM benchmark results of Hsieh versus Jenkins (Jen.) and `CHASH`.

counts each chain hit in the collision chain as a collision. The verification times of Jenkins's and `CHASH` are indexed against Hsieh's.

The table shows that the average gain of `CHASH` over Hsieh's is three percent when the models are compiled with `-O3` and 25 percent when the models are compiled with `-O0`. Later on, we shall discuss the differences between `-O0` and `-O3`. Most of the ten worst performing models have higher collision rates when used with `CHASH` in comparison to Jenkins's. It is also noteworthy that Jenkins's has the lowest collision rates (as reflected in the average collision rate), followed by `CHASH` and Hsieh's.

In 64-bits mode (see table 2) we see that `CHASH` is on average six percent faster than Jenkins's for `-O3` and nine percent faster for `-O0`. This gain is visible on all models for both `-O0` and `-O3`, even though the collision rates of `CHASH` are either on par or worse.

5.2 Bitstate Hashing

For bitstate hashing, we denote the accuracy of the search by the coverage indexed against the full state space size, which we know from the exhaustive verifications. Furthermore, we indexed the verification times of `CHASH` against Jenkins's here.

With 32-bits bitstate hashing (see table 3), we observed with `-O3` an average performance decrease of two percent using `CHASH`. With `-O0` there is a performance gain by 23 percent. With 64-bits bitstate hashing (see table 4, we see a performance gain of `CHASH` by 26 percent (with `-O3`) and 61 percent (with `-O0`).

The coverage rates of Jenkins's shows that it is a good hash function in terms of accuracy. `CHASH` performs less in that respect, as few models like hanoi.3, protocols.5, sorter.4 and reader_writer.3 have relatively low coverage rates. We

model	sv (bytes)	states ($\cdot 10^6$)	transitions ($\cdot 10^6$)	collrate Jen. (%)	collrate CHASH (%)	time -O3 Jen. (sec)	time -O3 $\leqq$ 100% CHASH (%)	time -O0 Jen. (sec)	time -O0 $\leqq$ 100% CHASH (%)
brp.5	148	11	20	1	3	29	61	50	90
brp.4	148	7	13	1	4	21	64	35	90
driving_phils.4	88	11	30	2	9	22	88	46	83
hanoi.3	116	14	43	5	7	39	91	91	86
phils.6	140	14	143	23	23	108	92	243	86
elevator2.3_prop4	52	8	55	5	9	26	92	57	88
elevator2.3	52	8	55	5	9	26	92	58	86
mcs.5	68	29	116	10	11	63	93	142	87
train-gate.3	164	20	57	4	29	49	93	123	97
telephony.7	64	22	114	9	15	55	93	130	87
...									
schedule_world.3	52	4	44	5	12	23	96	52	87
bakery.5	48	7	25	2	31	13	97	30	82
lann.3	140	5	24	2	2	24	97	54	99
msmie.4	180	7	11	0	1	14	97	31	92
production_cell.4	304	10	42	5	6	50	97	138	97
elevator.3	152	19	70	6	9	74	97	177	93
protocols.5	112	3	8	1	7	8	98	17	96
frogs.4	68	17	36	2	2	27	98	60	92
train-gate.2	164	18	50	3	30	42	99	108	97
reader_writer.3	276	1	4	0	1	19	99	39	100
Average	**120**	**12**	**47**	**4**	**20**	**34**	**94**	**79**	**91**

Table 2. 64-bits exhaustive search BEEM benchmark results of Jenkins (Jen.) versus CHASH

model	states ($\cdot 10^6$)	coverage Jen. (%)	coverage CHASH (%)	time Jen. (sec) -O3	rate ($\cdot 10^3$ states/sec) Jen. -O3	rate ($\cdot 10^3$ states/sec) $\leqq$ 100% CHASH (%) -O3	time Jen. (sec) -O0	rate ($\cdot 10^3$ states/sec) Jen. -O0	rate ($\cdot 10^3$ states/sec) $\leqq$ 100% CHASH (%) -O0
firewire_link.5	6	100	100	18	322	116	36	167	161
production_cell.4	10	100	100	42	237	109	92	109	153
phils.6	14	100	100	86	166	109	134	107	144
train-gate.3	20	100	100	43	465	109	78	253	139
train-gate.2	18	100	100	38	474	108	69	259	137
elevator2.3	8	100	47	25	302	107	36	211	122
fischer.6	8	100	100	20	426	107	30	280	128
elevator_planning.2	11	100	100	37	311	106	56	204	132
telephony.7	22	100	97	52	419	106	84	262	133
brp.5	11	100	100	18	603	105	32	337	135
...									
driving_phils.4	11	100	41	17	653	95	27	413	121
elevator.3	19	100	100	68	274	95	121	154	125
lann.4	13	100	99	55	231	92	105	120	123
bridge.2	9	100	100	24	392	92	40	230	106
msmie.4	7	100	100	12	605	92	20	359	123
frogs.4	17	100	100	26	662	90	42	412	111
protocols.5	3	100	63	8	413	89	12	265	111
sorter.4	13	100	89	21	625	89	34	383	117
reader_writer.3	1	100	91	16	47	76	30	25	102
hanoi.3	14	100	0	33	429	4	55	260	6
Average	**12**	**100**	**91**	**31**	**421**	**98**	**51**	**265**	**123**

Table 3. 32-bits bitstate search BEEM benchmark results of Jenkins (Jen.) versus CHASH.

found out this is due to the low entropy input as discussed earlier. Though this is combated using integer hash multiplication, in rare cases as these it is yet insufficient. We found out that additional effective measures are reordering the declaration of variables and/or making the state vector more sparse by adding dummy variables can be quite effective. Depending on the model, we gained nearly on par coverage. A patch against Spin that generates such models can be downloaded from the incremental hashing webpage. Note that these more unconventional measures are not universally effective and we measured that in general they decrease coverage. For this reason, they are not enabled with `CHASH` by default.

model	states (·10^6)	coverage Jen. (%)	coverage CHASH (%)	time Jen. (sec) -O3	rate (·10^3 states/sec) Jen. -O3	rate (·10^3 states/sec) CHASH -O3	time Jen. (sec) -O0	rate (·10^3 states/sec) Jen. -O0	rate (·10^3 states/sec) CHASH -O0
train-gate.2	18	100	100	76	234	163	138	129	207
production_cell.4	10	100	100	77	130	160	158	63	214
train-gate.3	20	100	100	79	250	149	155	128	208
production_cell.3	6	100	100	67	87	145	131	44	186
firewire_link.5	6	100	92	42	141	144	77	77	234
phils.6	14	100	100	208	69	137	305	47	196
bakery.7	28	100	100	83	332	136	130	211	161
fischer.6	8	100	100	40	206	135	67	125	170
brp.5	11	100	100	38	285	134	65	168	176
lamport_nonatomic.4	16	100	100	111	146	132	179	91	176
...									
schedule_world.3	4	100	100	38	113	120	60	71	149
bridge.2	9	100	100	41	229	120	70	134	133
peterson.6	9	100	100	42	206	120	59	146	141
lamport.7	5	100	100	25	186	119	35	134	140
adding.6	8	100	100	19	399	115	25	309	129
elevator_planning.2	11	100	100	78	147	114	117	98	156
at.4	7	100	100	32	203	113	49	136	150
sorter.4	13	100	88	39	335	112	64	207	151
reader_writer.3	1	100	91	26	29	98	46	16	114
protocols.5	3	100	37	19	165	58	26	119	78
Average	**12**	**100**	**97**	**61**	**205**	**126**	**100**	**130**	**161**

Table 4. 64-bits bitstate search BEEM benchmark results of Jenkins (Jen.) versus `CHASH`.

5.3 Hash-Compaction

The results from hash-compaction are similar to those from bitstate hashing. Here too we measured a small decline in performance when `CHASH` is used in 32-bits mode and `-O3` and a significant performance improvement of 26 percent when `-O0` is used. See table 5.

With 64-bits hash-compaction (see table 6), we see that `CHASH` improves by ten percent over Jenkins's with `-O3` and 29 percent with `-O0`. For the same reasons as for bitstate hashing, we see that here too a lower coverage goes accompanied by lower performance.

5.4 Optimisation Flags

While we ran our benchmarks with and without compiler optimisations enabled, Spin users usually do without them. XSpin does not enable them by default and

model	states (·10^6)	coverage: 100% Jen. (%)	coverage: CHASH (%)	-O3: time Jen. (sec)	-O3: rate Jen. (states/sec)	-O3: 100% CHASH (%)	-O0: time Jen. (sec)	-O0: rate Jen. (states/sec)	-O0: 100% CHASH (%)
firewire_link.5	6	100	100	16	372	122	33	181	180
phils.6	14	100	100	63	229	116	109	132	149
train-gate.2	18	100	100	32	565	110	63	283	140
train-gate.3	20	100	100	36	554	110	71	278	139
production_cell.4	10	100	100	36	275	109	84	119	158
telephony.7	22	100	97	37	587	109	70	312	137
telephony.4	12	100	100	22	566	106	40	304	140
lamport_nonatomic.4	16	100	72	45	357	105	91	177	141
mcs.5	29	100	93	42	692	105	72	405	126
peterson.6	9	100	100	14	621	105	22	393	123
...									
lamport.7	5	100	100	7	661	97	11	423	113
driving_phils.4	11	100	41	12	923	95	23	489	127
protocols.5	3	100	63	6	561	94	10	327	119
lann.4	13	100	99	47	267	93	98	128	126
elevator_planning.2	11	100	100	29	398	91	48	237	121
bridge.2	9	100	100	19	497	90	35	264	110
msmie.4	7	100	100	9	826	89	16	433	123
sorter.4	13	100	89	16	840	83	29	455	111
reader_writer.3	1	100	91	15	50	75	29	26	99
hanoi.3	14	100	0	25	583	4	47	306	8
Average	**12**	**100**	**91**	**23**	**582**	**98**	**44**	**322**	**126**

Table 5. 32-bits hash-compaction search BEEM benchmark results of Jenkins (Jen.) versus `CHASH`.

model	states (·10^6)	coverage: 100% Jen. (%)	coverage: CHASH. (%)	-O3: time Jen. (sec)	-O3: rate Jen. (states/sec)	-O3: 100% CHASH. (%)	-O0: time Jen. (sec)	-O0: rate Jen. (states/sec)	-O0: 100% CHASH. (%)
phils.6	14	100	100	83	174	131	150	95	179
production_cell.4	10	100	100	39	256	128	91	110	150
train-gate.2	18	100	100	36	500	123	73	246	142
train-gate.3	20	100	100	39	507	122	84	237	149
firewire_link.5	6	100	92	19	312	120	38	157	156
mcs.5	29	100	100	54	535	118	94	308	146
lamport_nonatomic.4	16	100	100	49	328	117	98	165	136
elevator2.3	8	100	100	23	332	116	39	199	131
telephony.7	22	100	100	45	493	116	78	281	135
production_cell.3	6	100	100	34	168	116	78	74	137
...									
lamport.7	5	100	100	12	391	106	17	273	118
msmie.4	7	100	100	13	541	104	23	313	128
bridge.2	9	100	100	23	400	104	43	216	107
driving_phils.4	11	100	83	16	690	104	27	416	126
schedule_world.3	4	100	100	19	220	102	34	126	114
adding.6	8	100	100	9	836	101	12	634	111
sorter.4	13	100	88	21	628	99	34	388	111
elevator_planning.2	11	100	100	36	317	97	62	185	129
reader_writer.3	1	100	91	19	40	94	35	21	101
protocols.5	3	100	37	9	343	51	14	226	68
Average	**12**	**100**	**97**	**29**	**429**	**110**	**53**	**250**	**129**

Table 6. 64-bits hash-compaction search BEEM benchmark results of Jenkins (Jen.) versus `CHASH`.

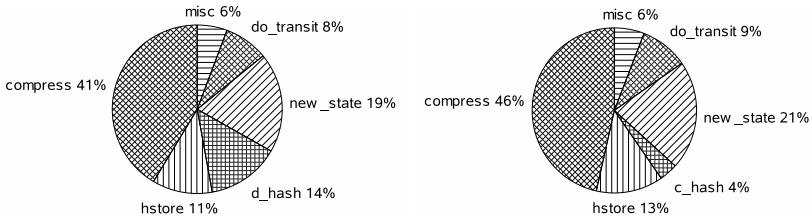
users usually forget to enable them manually. Based our results, we see that enabling optimisations (i.e. `-O3`) makes a significant difference, as it reduces the verification time nearly by an half. This substantial improvement costs only a few seconds additional compilation time.

5.5 Memory Consumption

We also extracted memory utilisation statistics for runs[5] with and without `CHASH`. For both `-O0` and `-O3`, we measured an average memory overhead by `CHASH` of six percent compared to runs with Jenkins's. This is caused by our implementation, which maintains hash values on the DFS stack such that a reverse `CHASH` operation does not have to be computed.

5.6 Profiler Runs

We also wanted to find out how much `CHASH` improves and whether there is more room for improvement. We ran a profiled version on our selection of the BEEM benchmark suite. For pointing out the interesting points, it sufficies to only present the combined profiler data from 32-bits and 64-bits exhaustive verification. See figures 1 and 2. In this figure, `d_hash` is Jenkins's hash function, `c_hash` is the `CHASH` implementation, `hstore` is the hashtable storage function, `new_state` is the DFS routine, `compress` is the mask-compression function, `do_transit` performs one transition from the current state and misc are all other functions.



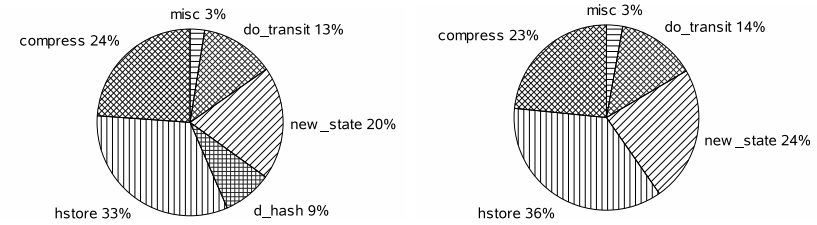
(a) Jenkins. (b) `CHASH`

Fig. 1. Percentual stakes of five most time-consuming functions for BEEM benchmarks compiled with `-O0`.

The stake of hashing with Jenkins's is for both `-O0` and `-O3` clearly visible in the total running time. When `CHASH` is enabled, it eliminates hashing as a visible stake in the total running time. For `-O3`, the stake of `CHASH` is near zero and therefore not depicted in the figure.

Also noteworthy is that `compress` and `hstore` have a significant stake for both `-O0` and `-O3`. The former is in Spin 5.1.4 disabled by default in case of 32-bit

[5] We were not able to include runs with compress disabled for measuring memory overhead, as Spin does not output memory statistics when compress is disabled.



(a) Jenkins. (b) `CHASH`

Fig. 2. Percentual stakes of five most time-consuming functions for BEEM benchmarks compiled with `-O3`.

exhaustive verification of safety properties. We measured the impact of this in our benchmarks, and detected that disabling mask-compression improves performance by ten percent in `-O3` and 36 percent in `-O0`. This however comes at the cost of increased memory consumption. Unfortunately, this was not measurable because runs with compress disabled do not output memory statistics.

5.7 Extremely Long Runs

We also experimented with models that either run out of memory or have a high verification times. We specifically reran the BEEM benchmark with 64-bits bitstate enabled, profiler enabled, compiler optimisations enabled, hashtable size of 2^{36}, maximal depth of 20 million and $k = 3$. The table below is a selection of five models with the longest verification times:

model	sv size (bytes)	type	states ($\cdot 10^9$)	transitions ($\cdot 10^9$)	depth	time (hours)	rate (states/sec)	gain (%)
firewire_link.6	420	Jenkins	19	64	1887952	54.9	94598	134
		`CHASH`	19	67	1281175	42.1	126380	
peg_solitaire.6	60	Jenkins	2	25	36	12.4	53540	103
		`CHASH`	2	25	36	12.0	55238	
driving_phils.5	96	Jenkins	6	15	304	4.6	339291	133
		`CHASH`	5	14	304	3.3	450963	
lamport_nonatomic.5	224	Jenkins	2	8	max.	4.0	105885	121
		`CHASH`	1	7	max.	3.1	128062	
telephony.6	64	Jenkins	1	8	max.	2.1	186624	125
		`CHASH`	1	8	max.	1.7	233323	

The gain represents the verification rate index of `CHASH` when compared against Jenkins's. Runs for which the maximal depth was reached are indicated by max. in the depth column.

As can be seen, `CHASH` improves greatly over Jenkins with an improvement of up to 34 percent. As can be seen from the times, this improvement can save hours of verification. Also particularly interesting is that this selection of models have quite large state vectors. This suggests a correlation between the state vector size and the performance gain by `CHASH`. The BEEM benchmark suite includes

too little models with large state vectors and significant large state spaces in order to measure such a correlation.

6 Conclusions

There are still several ways to improve our incremental hashing scheme as implemented in Spin. First, we used integer hash functions to improve its uniformity, and though this works out well, there is room for improvement. We saw that for some models, the collision rates were relatively high and/or the coverage rates relatively low. By devising other methods for function T, the mapping of integers to the ring R, this may be improved. Our experiments with sparse state vectors and variable reordering for bitstate hashing also help, but more investigation is required to define an approach that is on average substantially better.

Also, currently untested is the use of incremental hashing with multi-core model checking (available since Spin 5) and the verification of liveness properties. This is likely to require additions to the `CHASH` implementation.

`CHASH` can be also used orthogonally upon traditional hash function, as a good second opinion second hash. This hash code can be stored along the state in the hash table, and used as an additional check before byte-for-byte comparison is done. This can improve the performance of `hstore` function, of which profiler results have shown that improvements in this function is likely to be reflected in the total running time.

The concept of incremental computation can also be extended to mask-compression and the state collapser. Having an incremental collapse also enables a nicer implementation of incremental hashing, as it will not be necessary anymore to add a `c_hash` at every update of the state vector.

Lastly, the BEEM benchmark suite served their purpose for the greater part of our experiments. It was only lacking on one point, and that is where we wanted to unfold a correlation between the state vector size and the performance gain. The problem lies in the lack of models that have both large state spaces and large state vectors. Adaptations of models in the current suite, or a series of new models that do have those properties would be welcoming for increasing the usefulness of the BEEM benchmark suite even further.

Conclusive, we described an incremental hashing that is applicable to any state vector datastructure, implemented it in Spin and evaluated it using the BEEM benchmarks. From this evaluation, we observed that Spin's default settings, namely with compiler optimisations disabled, that incremental hashing is superior to Hsieh's `SFH` and Jenkins's in all cases. With the most aggressive safe compiler optimisations enabled, namely `-O3`, `SFH` is generally better for 32-bits exhaustive search, Jenkins's for all other 32-bits verification modes and incremental hashing is better in all approximate modes and 64-bits in particular. The average reduction of applying compiler optimisation is nearly a half. We recommend it to always enable it, and in case when 64-bits machines are used, combine this with incremental hashing.

The full result set from the BEEM benchmarks and `CHASH` patch against Spin 5.1.4 can be downloaded from the incremental hashing webpage [17].

References

1. S. Baase and A. van Gelder. *Computer Algorithms.* Addison-Wesley, 2000 (3rd).
2. Jonathan D. Cohen. Recursive Hashing Functions for N-grams. *Transaction On Information Systems*, 15(3):291–320, 1997.
3. Peter C. Dillinger and Panagiotis Manolios. Fast and Accurate Bitstate Verification for SPIN. In Susanne Graf and Laurent Mounier, editors, *Model Checking Software - Proc. of the 11th Int. SPIN Workshop (SPIN 2004), Barcelona, April 1-3, 2004*, LNCS 2989, pages 57–75. Springer-Verlag, 2004.
4. Gerard J. Holzmann. On Limits and Possibilities of Automated Protocol Analysis. In *Proc. 7th Int. Workshop on Protocol Specification, Testing and Verification (PSTV '87)*, pages 137–161, Amsterdam, 1987. North-Holland.
5. Gerard J. Holzmann. An Analysis of Bitstate Hashing. *Formal Methods in System Design*, 13:289–307, 1998.
6. Gerard J. Holzmann. *The* Spin *Model Checker – Primer and Reference Manual.* Addison-Wesley, Boston, Massachusetts, USA, 2004.
7. Bob Jenkins. Hash Functions. *Dr. Dobbs Journal*, 22(9), September 1997.
8. R.M. Karp and M.O. Rabin. Efficient Randomized Pattern-Matching Algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.
9. Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd ed.) Sorting and Searching.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.
10. Matthias Kuntz and Kai Lampka. Probabilistic Methods in State Space Analysis. In Christel Baier, Boudewijn R. Haverkort, Holger Hermanns, Joost-Pieter Katoen, and Markus Siegle, editors, *Validation of Stochastic Systems – A Guide to Current Research*, LNCS 2925, pages 339–383. Springer-Verlag, 2004.
11. Tilman Mehler and Stefan Edelkamp. Dynamic Incremental Hashing in Program Model Checking. *ENTCS*, 149(2):51–69, 2006. Proc. of Third Workshop of Model Checking and Artificial Intelligence (MoChArt 2005).
12. Viet Yen Nguyen. Optimising Techniques for Model Checkers. Master's thesis, University of Twente, Enschede, The Netherlands, December 2007.
13. Radek Pelánek. BEEM: Benchmarks for Explicit Model Checkers. In Dragan Bosnacki and Stefan Edelkamp, editors, *Proc. of the 14th Int. SPIN Workshop (SPIN 2007)*, LNCS 4595, pages 263–267. Springer-Verlag, 2007.
14. Theo C. Ruys and Niels H. M. Aan de Brugh. MMC: the Mono Model Checker. *ENTCS*, 190(1):149–160, 2007. Proc. of Bytecode 2007, Braga, Portugal.
15. Pierre Wolper and Denis Leroy. Reliable Hashing without Collosion Detection. In Costas Courcoubetis, editor, *Proc. of the 5th Int. Conference on Computer Aided Verification (CAV '93)*, LNCS 697, pages 59–70, 1993.
16. BEEM: BEnchmarks for Explicit Model checkers. `http://anna.fi.muni.cz/models/`.
17. CHASH - Incremental Hashing for SPIN. `http://www-i2.cs.rwth-aachen.de/~nguyen/incrementalHashing`.
18. Fowler/Noll/Vo (FNV) Hash. `http://isthe.com/chongo/tech/comp/fnv/`.
19. Paul Hsieh. Hash functions. `http://www.azillionmonkeys.com/qed/hash.html`.
20. Bob Jenkins. A Hash Function for Hash Table Lookup. `http://burtleburtle.net/bob/hash/doobs.html`.
21. SPIN: on-the-fly, LTL model checking. `http://spinroot.com/`.
22. Thomas Wang. Integer Hash Function. `http://www.cris.com/~Ttwang/tech/inthash.htm`, 2007.