Concurrent Chaining Hash Maps for Software Model Checking

Freark I. van der Berg
Formal Methods and Tools
University of Twente, Enschede, The Netherlands
f.i.vanderberg@utwente.nl

Jaco van de Pol
Aarhus University, Dept. of CS, Denmark
University of Twente, Enschede, The Netherlands
jaco@cs.au.dk

***Abstract*—Stateful model checking creates numerous states which need to be stored and checked if already visited. One option for such storage is a hash map and this has been used in many model checkers. In particular, we are interested in the performance of concurrent hash maps for use in multi-core model checkers with a variable state vector size. Previous research claimed that open addressing was the best performing method for the parallel speedup of concurrent hash maps. However, here we demonstrate that chaining lends itself perfectly for use in a concurrent setting.**

We implemented 12 hash map variants, all aiming at multi-core efficiency. 8 of our implementations support variable-length key-value pairs. We compare our implementations and 22 other hash maps by means of an extensive test suite. Of these 34 hash maps, we show the representative performance of 11 hash maps.

Our implementations not only support state vectors of variable length, but also feature superior scalability compared with competing hash maps. Our benchmarks show that on 96 cores, our best hash map is between 1.3 and 2.6 times faster than competing hash maps, for a load factor under 1. For higher load factors, it is an order of magnitude faster.

***Index Terms*—concurrency, data structure, hash map, high-performance, multi-threaded, thread-safe, model checking**

I. Introduction

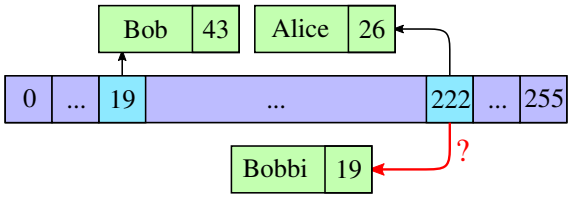
Stateful Model checkers store visited states. This has the advantage of being able to detect whether a state has already been visited, such that it need not be visited again. For storage of these states, options include hash maps [1], compression trees [2] and binary decision diagrams [3] to name a few.

Some model checkers make use of prevalent multi-core hardware by supporting multiple concurrent threads. Storage and checking of states thus additionally involves communication between threads.

Software model checkers that support the verification of programs that manipulate stack or heap memory can benefit from a fast hash table that can store variable-length states. The latter is required because the stack and heap of a program can grow or shrink, so not all states have the same size.

In this paper, we want to investigate the option of using a hash map for the purpose of multi-core software model checking. To this end, we do not require deletion or resizing from the hash map. However, we need the hash map to be thread-safe and to allow variable-length keys to be stored.

Fig. 1 Bucket collision: where to insert Bobbi's age?



A. Hash maps for storage

Hash maps are data structures that are used to map a *key* to a *value*. They are mainly useful when the set of all possibles keys is large or sparse. In a hash map, a *hash function* maps the large domain of the keys to a smaller domain. For example, one could add the ASCII values of all characters in a string together, modulo 256. This would give a number in the range $[0, 256)$. Thus, we can represent a hash map using the simple array of 256 elements. Subsequently, if we were to insert the age of Alice, who is 26, into this hash map, her age would end up at position 222 in the array, because $(65 + 108 + 105 + 99 + 101) \mod 256 = 222$.

However, a problem arises when we want to insert the age of Bobbi. The hash function we just thought of maps his name to 222 too, as $(66 + 111 + 98 + 98 + 105) \mod 256 = 222$. This is called a *bucket collision*: two keys map to the same position (bucket) in the array. In this case it is even a *hash collision*, since the keys map to the same hash. Figure 1 depicts the problem of where to put the node of Bobbi. In our example the hash function was chosen quite poorly: a lot of names share the same hash, causing many bucket collisions.

Bucket collisions are undesired, since two entries map to the same bucket and thus one of them needs to be put somewhere else. This increases the *probe count* for that entry: to put it into the hash map or to determine whether it is in the hash map requires multiple *probes*, i.e. checks if a bucket contains a specific key. Probes are expensive operations because they access the main memory and can cause *cache misses*.

B. Concurrent hash maps

There has been ample research about hash maps, both single-threaded and multi-threaded. Single-threaded research [4], [5] focuses on limiting memory overhead and algorithmic improvements in how to resolve bucket collisions.

In a multi-threaded application, where multiple threads operate on the same hash map concurrently, resolving bucket collisions is more complicated [6], [7], [8]. This is due to the nature of running threads concurrently: special atomic operations have to be used to avoid the hash map from becoming inconsistent or corrupt.

C. Contributions

The two main contributions presented in this paper are 1) an extensive comparison of a number of hash maps; 2) a new set of high-performance hash map implementations supporting variable-length key-value pairs. For our use case, our best hash map outperforms all competition. In addition to these, we convey the knowledge that chaining hash maps are perfectly suited for multi-core model checkers, contrary to what was previously believed [1].

We analyze and compare our implementations and 22 other hash maps by means of an extensive test suite. During the test, we capture hardware events and analyze them using Intel® VTune™ Amplifier. The result is an extensive dataset of statistics on 34 hash maps.

In Sections II and III we provide a background in hardware architecture and hash maps in general. In Section IV we describe how we implemented our hash maps and in Section V we discuss related research and related hash maps. In Section VI we explain the experimental comparison of all hash maps and Sections VII and VIII show the results for two test scenarios. We conclude and list possible future avenues of exploration in Section IX.

II. Background on hardware architecture

Computer hardware has evolved greatly in the past 70 years. From a simple single processor with a single core and single memory bus connected to a single memory bank, to a vastly complex machine with many processors, caches, memory busses and memory banks. It is an ever-increasingly interesting field for high-performance software.

1) Memory cache: The need for caches is due to the *memory wall* [9]: the CPU is getting faster and faster than the memory. Modern processors have a cache hierarchy between the cores of the processor and the main memory, to hide the latency of the much slower main memory: cache memory is significantly faster than main memory. When a value is not in the cache, it needs to be obtained from the slower main memory. This is called a *cache miss*.

2) Inter-thread communication: Because of physical limitations, the performance of CPUs is not increased by increasing the clock speed, but instead by adding more cores that can seemingly work independently. However, having multiple cores adds complexity not only to the hardware, but to the software as well. Multiple *threads* can run on multiple cores concurrently and in parallel. Correct communication becomes paramount: many interleavings of memory operations are possible and all of them need to be correct. Special memory instruction, such as compare-and-swap (`CAS`) and memory barriers, are expensive to execute, so they need to be kept to a minimum for high performance.

III. Hash maps: how to resolve collisions

There are many ways to implement a hash map [10]. The main distinction is how bucket collisions, such as the one between Alice and Bobbi in the introduction, are handled. One way is to just try a different bucket in the array that hosts the hash map. This is called *open addressing*. Another technique to solve a bucket collision is to link the new entry to the entry already in the bucket; this is aptly named *chaining*. In a chaining hash map, every bucket is a linked list of entries.

A. Open addressing

Open addressing does not use a chain of entries, avoiding the need for a `next` field per entry as can be seen in fig. 2a.

In theory, this requires less memory. In practice, the fuller an open addressing hash map gets, the worse it performs, so a margin is required. On a bucket collision, the next bucket can be determined for example by *linear probing*, *quadratic probing*, *cuckoo hashing* and *rehashing*.

1) Linear probing: The simplest way to find an empty bucket is by linearly probing the buckets, starting from the bucket collision, until we find one that is empty.

However, linear probing is susceptible to *clustering*. Clustering happens when a group of nearby buckets are occupied. When a bucket is full, it also increases the probe count for inserts to the bucket before it, if that one is full as well.

2) Quadratic probing: Quadratic probing refers to that the next bucket is determined by skipping an increasing number of buckets. In the ith probe, we try the i^2th bucket after the first, starting at 0. This lowers the effect of clustering, because there is increased space between buckets.

3) Cuckoo hashing: Cuckoo hashing [5] uses multiple hash functions. An element can only be found at the indices provided by these hash functions. If all of these locations are already used, one of them is taken out and the new element takes its place. The taken out element is reinserted similarly.

To be able to guarantee the constant-time lookup, there is an upper limit on how full the hash map can be. For two hash functions, this is 50%. If it is higher, due to collisions, the recursive rehashing can take significantly longer.

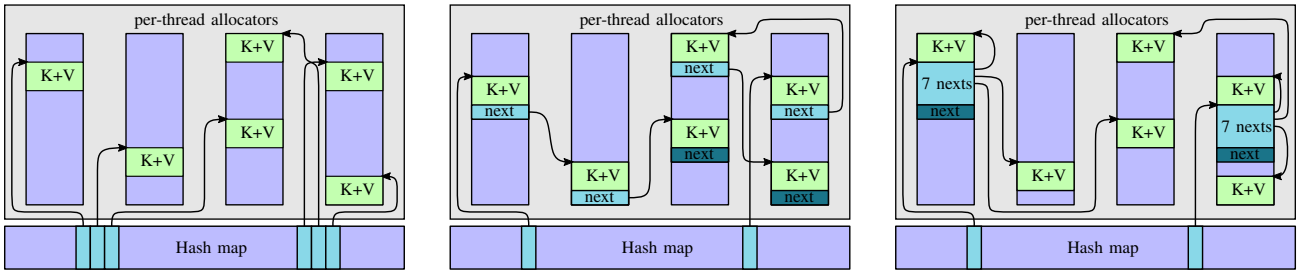
4) Rehashing: When a bucket collision occurs, it is also an option to attempt a rehash of the key or modify the hash in a deterministic way. It is vital that the rehash or modification be done deterministically in order to check if the key is already in the hash map. The modification can be done differently for nearby buckets, mitigating clustering even further.

B. Chaining

Each bucket, in addition to an entry, has a `next` field. When an entry is to be inserted in a bucket where already an entry resides, the entry is linked using the `next` field. In other words, every bucket is a linked list of entries, as can be seen in fig. 2b.

This has the advantage that there is no clustering due to neighboring buckets: bucket collisions are solved by just appending them to the linked list. The downside of this is that every linked in element requires a pointer and thus increases the memory footprint and accessing may cause a cache miss.

Fig. 2 Memory layouts for various hash map implementations. A dark (blue) fill indicates a `null` pointer



(a) Memory layout for an open addressing hash map

(b) Memory layout for a chaining hash map

(c) Memory layout for a chaining hash map with cache buckets

IV. Hash Map Design Considerations

Cache misses are expensive because the load operation needs to be serviced by the much slower main memory. In order to implement high-performance hash maps, we need to take this into account. Our implementations use existing algorithms, but are implemented with high-performance concurrency in mind. As such, we try to minimize the number of cache misses and the number of expensive memory operations that synchronize cores, such as `CAS` and memory barriers.

Moreover, the concurrent hash maps we implemented adhere to the C++11 memory model. We make use of C++11 release and acquire memory barriers. The C++11 compiler maps these to the hardware memory model and thus the hash maps can run correctly on all platforms supported by the compiler. For the implementation of our hash maps we refer to our Github repository[1].

A. Hash function

The hash function is an important part of a hash map, for it represents where to start probing for an empty bucket. Thus, it is vital that the hash function distributes keys as uniformly as possible over the buckets, to minimize bucket collisions.

We tried various hash functions: `FNV-1a`, `SDBM`, `MurmurHash3`, `MurmurHash64A`, and `SuperFastHash`. Of these, `MurmurHash64A` yielded the best performance, so we used this hash function for all tests.

B. Allocator for entries

Both chaining and open addressing hash maps use a specialized allocator for variable-length and fixed-length key-value pairs that do not fit in a single word. This allocator allocates a slab of memory for each thread, to avoid issues with concurrency. For chaining, a thread can write to memory allocated by another thread, but only to link in an entry. Allocating memory with this allocator is simply done by increasing a pointer by the number of bytes required and then returning the old value of that pointer. Since there is an allocator per thread, this can be done without expensive synchronization instructions. This design contributes to the speedup of our implementations. Figure 2 illustrates the memory layout for the hash maps and the allocator.

This allocator is not used for hash maps that map integers to integers. There, keys and values are stored in situ.

C. Optimizations

1) 16 upper bits: To identify the address where a key-value pair is stored, 48 bits is adequate. These 48 bits can index the slab allocated by the allocator. We can use the upper 16 bits to store a 16-bit version of the hash. We obtain this by combining the four 16-bit segments of the 64-bit hash of a key using `xor`. Then, we store this 16-bit hash in the 64-bit pointer that points to the key-value pair. Thus, when searching for a key, we can first compare this 16-bit hash before following the pointer and comparing the key itself. This can save a significant number of loads and thus cache misses.

2) Cache-aware: Instead of indexing at the bucket level, first we index at the *cache bucket* level. A cache bucket is a group of buckets that fit precisely on one cache line[2]. This can be combined with various probing methods. We implemented linear and quadratic probing. Within the cache bucket, we start at the bucket we would have started at if we would have indexed at the bucket level. We continue linearly, wrapping at the end of the cache line, until we reach the start bucket. If by now we have not found the bucket we needed, we go to the next cache bucket.

D. Implemented variants for experimentation

In order to determine the important factors influencing parallel speedup, we implemented 12 hash maps: 4 chaining, 4 open addressing and 4 open addressing in-situ, without an allocator, to establish a base line. Each 4 implement different optimizations, coded by the following suffixes: *Q*: the hash map uses quadratic probing instead of linear; *U*: the optimized use of upper-bits, explained in Section IV-C1; *C*: the hash map uses a cache-awareness optimization, explained in Section IV-C2; We also experimented with *D*: the use of one double word `CAS` instead of two single-word `CAS`, but this did not improve performance.

To avoid cluttering the results, we show only the best hash map in their category: ChainU, OpenAddrQCU and InsituQU. (cf. Table I). For open addressing, cache-aware quadratic probing with linear probing within cache lines was the most performant. For both chaining and open addressing, using the upper-bits for a 16-bit hash yielded improvement.

For chaining, attempting to make it cache-aware by using a cache-line as link in the chain (see fig. 2c) degraded performance, so the simple version was faster.

[1]Data and code can be found at https://github.com/bergfi/hashmap/

[2]A cache line is usually 64 bytes and would thus fit 8 buckets

Table I Of the 34 hash maps we tested we show results of these 11 hash maps. The first 3 we implemented, the other 8 we use for comparison. "best" refers to the hash maps being tested using our test suite and they outperformed others in their group.

	Name	Key	Value	Probing	Features / Comment
	OpenAddrQCU	variable	variable	open: quadratic + linear	Best of all 4 open addressing implementations
	ChainU	variable	variable	chaining	Best of all 4 chaining implementations
	InsituQU	48-bit	64-bit	open: quadratic	Best of all 4 in situ implementations
	dbsll [1]	constant	unique int	open: rehash + linear	unique int given by the hash map
	TBB [11]	constant	constant	chaining	Intel TBB's `concurrent_hash_map`
	TBBA [11]	constant	constant	chaining	As TBB, but with the allocator of this paper
	libcuckoo [12], [13]	constant	constant	cuckoo	libcuckoo with TBB's scalable allocator
	Junction:Crude [14]	64-bit	64-bit	open: linear	Best of all Junction hash maps
	Michael [15], [16]	64-bit	64-bit	sorted chaining	Part of LibCDS library, NOGC, no counter
	Skiplist [17], [16]	64-bit	64-bit	skip list	Part of LibCDS library, NOGC, no counter
	Folly [18]	64-bit	64-bit	open: quadratic	Hash map focusing on insertion performance

V. Related Work

Hash maps have been extensively researched, both single-threaded [4], [5] and multi-threaded [6], [7], [8]. Generally, the research focused on a hash map with support for deletion of entries. Since our purpose is stateful multi-core model checking, we focus on the `findOrPut` performance of hash maps and do not require support for deletion or resizing.

We compared our 12 implementations to 22 variants from mainly seven competitors, totaling 34 hash maps. Of these, we only display for each competitor the most performant one, according to our test suite. These are listed in Table I. While most focus on mapping 64-bit keys to 64-bit values, some support any constant-length keys. In order to investigate the effect of using different memory allocators, we also linked some of the competitor hash maps with our own memory allocator. The hash maps we used for comparison are:

- Intel TBB's `concurrent_hash_map` [11] is part of the *Threading Building Blocks* library. This popular concurrency library provides hash maps with support for any constant-length key-value pairs. We test with both their scalable allocator and the allocator in this paper .
- Junction [14] is a library made by Preshing containing hash maps with interesting implementations. While he did not publish a paper on this matter, he explains his hash maps in his blog post.
- libcuckoo [12] is a library implementing cuckoo hashing and supports constant-length key-value pairs.
- dbsll [1] is a hash map created with the purpose of mapping constant-length vectors to a unique integer, so the user cannot store a chosen value.
- From the CDS [16] library we measured the performance of the Michael Map using the Michael list and the Skiplist . Both are without item counter and are the NOGC versions, which stands for no garbage collection, i.e. they are append only. This levels the playing field, as our hash maps are also append only.
- Facebook's Folly AtomicHashMap [18]. This is a hash map with a focus on high-performance and is advertised as being 2x – 4x faster than TBB . While entries can be deleted, the used memory is not reclaimed.

In addition to these, we tried testing Cache Line Hash Table (CLHT) [6], but this hash map could not complete our test within a day. Smaller tests CLHT managed, but was still not able to beat our implementations. We also tried the concurrent hash set of DIVINE [19], but it had a similar problem.

The Grow Table library [7] consists of hash maps for general purpose, e.g. they support deletion and growing, but they only support integers for keys and values. We focus on the performance of `findOrPut` of vector data and do not require deletion. Additionally, we analyze all hash maps we tested using Intel® VTune™ Amplifier[3], to explain the performance of the hash maps. When putting their hash maps through our test suite, the performance was erratic and surprisingly low. We refer to our online data-set for the precise numbers.

Like us, the focus of dbsll [1] was to maximize `findOrPut` performance without support for deletion, to increase the multi-threaded performance of the model checker LTSmin. This makes dbsll our primary competitor and gives us a base line for the performance that is achievable. However, where they assumed open addressing to be superior to chaining, we implemented both and came to a different conclusion. They also only support constant-length vectors as keys, whereas our implementations support variable-length.

Feature-wise, the chunk table of LTSmin (not dbsll) is the only competitor, because it is the only other hash map supporting variable-length keys. However, its performance is two orders of magnitude lower, so we did not include it.

A. Other related work

Oortwijn et al. have investigated distributed hash maps [20] with the same goal of optimizing the throughput of `findOrPut`. These hash maps span not only multiple cores, but multiple computers.

Wijs et al. researched implementing hash maps on the GPU for the purpose of state space exploration [21]. GPUs use streaming multiprocessors that perform a single instruction on multiple data (SIMD), allowing great parallelism.

Tries [22] are an interesting data structure with a similar goal to hash maps. There has been extensive research, comparing [23] them, even in a concurrent setting [24].

[3] https://software.intel.com/en-us/intel-vtune-amplifier-xe

VI. Test Setup

We want to measure the performance of `findOrPut` under various conditions in order to determine which hash map is best suited for multi-core model checking. To this end with have a test scenario that inserts 64-byte vectors. We use constant-size key-value pairs in order to compare to other hash maps, because our competitors do not support variable-length.

In addition to this, we have a test scenario inserting 64-bit integers. We add this test to be able to compare to even more hash maps. Of this test, we provide an analysis using Intel® VTune™ Amplifier. The results are discussed in Sections VII and VIII. Here we explain our test setup.

A. How we tested

For both scenarios, a single test run contains a single test data preparation phase and 7 times the following steps are performed:

1) *Setup* phase of the hash map under test, where for example memory is allocated. Hash map is initialized to 2^{28} buckets;
2) *Insertion* phase: data is inserted using `findOrPut`: $2^{28} \cdot l$ key-value pairs are inserted, l is the load factor;
3) *Verification* phase: all inserted data is obtained from the hash map using `get` and verified to be complete;
4) *Clean up* phase, where the hash map is deleted and its memory freed.

Each thread has its own equally sized segment of the generated data, ensuring all generated data is inserted for each test run, regardless of number of threads.

B. Performance and Analysis

This single test run is executed two times: once without any analysis to measure the performance and once with Intel® VTune™ Amplifier using the Memory Access analysis. We do this separately so VTune does not influence the performance numbers. VTune uses the profiling data gathered by the Intel CPU and processes these hardware events into a performance analysis. It collects information such as number of cache misses, number of store operations, and many more. We pause the gathering of data when not in the insertion phase, such that the numbers shown are only for the insertion phase.

C. Test scenarios

1) 64-byte vector → integer: This test inserts 64-byte vectors as keys, which each map to an integer. During the *setup* phase, this test generates a number of unique 64-byte vectors. We test the influence of the number of threads and the load factor. For this test we do not separately control the collision rate, so it depends on the load factor.

We tested with inserting 50% duplicates as well, causing `findOrPut` to make an insertion only 50% of the time. The results of those tests showed a similar pattern as inserting unique keys, so we did not include them here.

2) integer → integer: This test inserts integer keys, mapping them to integer values. For this test, we want to investigate the influence of collisions. To achieve this, the hash map uses the identity function as hash function and each threads inserts I elements, $\{f(i) : 0 \leq i < I\}$, where

$$f(i) = h\Big(\big(i \mod (I/c)\big) + I \cdot t\Big) + \lfloor \frac{i}{I/c} \rfloor \cdot B$$

where c is the desired collision ratio, B is the number of buckets, h is the hash function (not the identity), t is the thread ID. Note that $1 \leq c$ and thread IDs start at 0 and are incremented by one. For example, $B = 32$, $I = 8$, $c = 2$, 4 threads, inserting 1 and 5 cause a collision, among others.

The collision ratio indicates the number of inserts per bucket. The collisions happen in the same thread, so two threads do not compete for the same bucket, but they still compete for the cache line the bucket is in. We also ran a few experiments where two or more threads do compete for the same bucket and this showed similar results.

D. Environment

The hardware we ran our experiments on is "caserta"[4], a Dell R930 with 2TiB of RAM and four E7-8890-v4 CPUs. Each CPU has 24 cores, 60MiB of L3 cache and 512GiB of RAM, offering 96 physical cores in total and 192 cores using hyper-threading. Hyper-threading is an Intel® technology that makes a single physical core appear as two logical cores to the operating system [25].

We ran our experiments on Ubuntu 16.04 GNU/Linux 4.4.0-116. All tests are compiled and linked using GCC 8.0.1.

The tests are executed in such a manner that threads are spread out over the cores, meaning that each CPU is assigned a fair share of threads. This approach has two main advantages. Firstly, there is more cache available, as each CPU has its own cache, resulting in potentially fewer cache misses. Secondly, the memory is allocated evenly over all memory banks[5], providing uniform access for all threads. If the threads would be clustered on one CPU, the access would be non-uniform, as our use case is a single large hash map that spans all available memory. VTune confirmed this by indicating that memory accesses were roughly 75% to memory banks on other CPUs

We attempted to create an equal environment for all hash map implementations, such that the performance of the hash map itself is the most significant factor in the results. All hash maps use `MurmurHash64A` as the hash function for vectors.

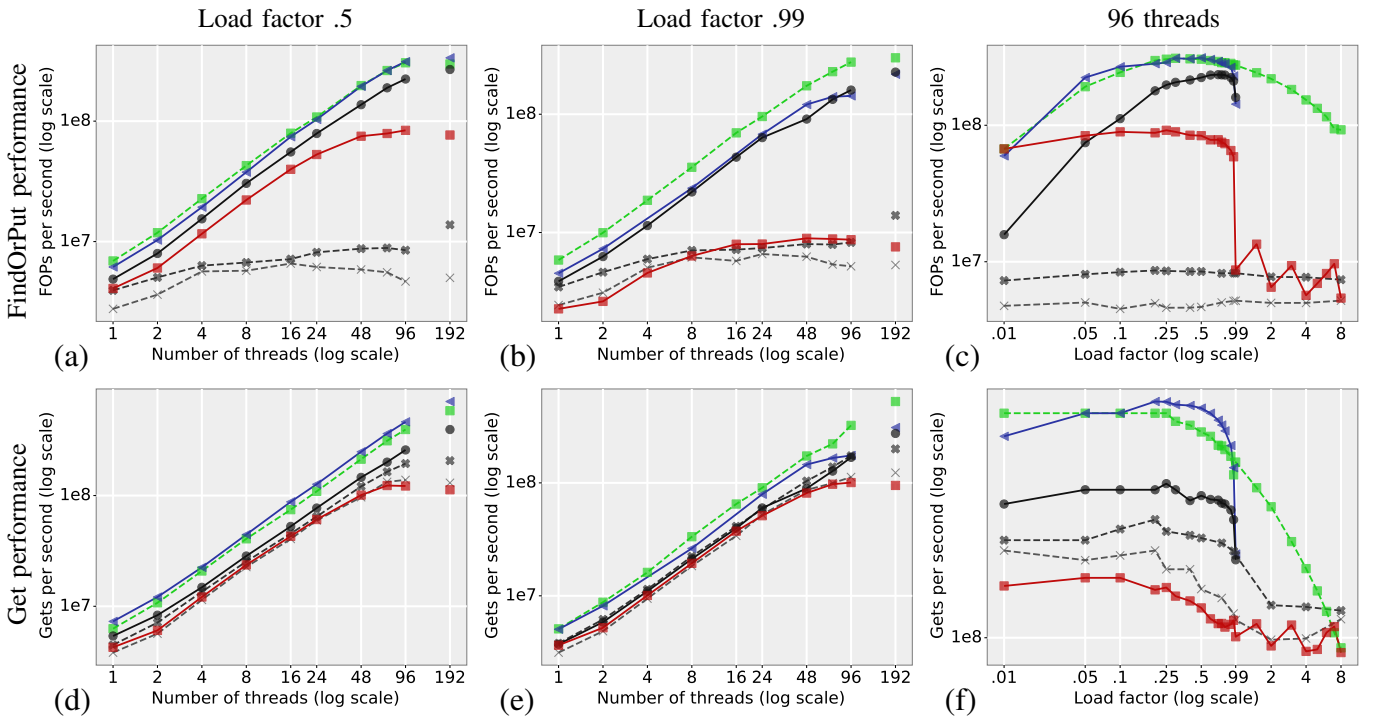
The entire test suite took 1105 hours (~46 days) of wall-clock time to execute, gather data and the processing of this data. The data gathered constitutes 7.2TiB of hardware events. The summary of this data and all generated graphs are available online[6], including the code that was benchmarked.

[4]Sponsored by 3TU Big Software on the Run project, http://www.3tu-bsr.nl/

[5]The Linux kernel allocates physical memory for a large region of virtual memory when a CPU writes to it and in the memory banks of that CPU [26].

[6]Data and code can be found at https://github.com/bergfi/hashmap/

Fig. 3 Results for scenario 1, inserting pseudo random 64-byte keys into 2^{28} buckets. Load factors indicate the number of `findOrPut` (FOP) operations. The results for 192 threads are with use of hyper-threading. For a legend, see Table I.



VII. 64-byte vector → integer results

Figure 3 shows the `findOrPut` and `get` performance of inserting 64-byte vectors. The first thing we notice is the significant difference between hash maps (3a). TBB is not well-suited for our use case, as it is an order of magnitude slower than other hash maps. Even using our allocator yields only a minor increase in performance, in TBBA.

While libcuckoo shows better performance for a .5 load factor, it is still more than a factor 3 slower. OpenAddrQCU and ChainU are quite evenly matched (3a). For load factors higher than .95, ChainU shows that chaining is better able to deal with an increased number of bucket collisions (3b).

For load factors above 1, open addressing hash maps dbsll and OpenAddrQCU cannot compete, because they do not support growing. libcuckoo does support growing and the results confirm that it grows when the size doubles. However, this is where chaining gains the most ground: ChainU is an order of magnitude faster than other hash maps (3c).

The performance for the verification phase (3d) varies much less, with only a factor 3 between the fastest and the slowest. For .99 load factor we again see ChainU outperforming all other hash maps (3e). Hyper-threading seems to have a more positive impact on `get` than on `findOrPut`.

Looking at the influence of load factor (3f), TBB even outperforms libcuckoo. As the load factor increases, the `get` performance of all hash maps seem to converge.

In summary, chaining outperforms open (quadratic + linear) by an order of magnitude for load factors above 1, while on par for lower load factors. Our ChainU beats all competitor hash maps in `findOrPut` performance by 1.3x–2.6x.

VIII. integer → integer results

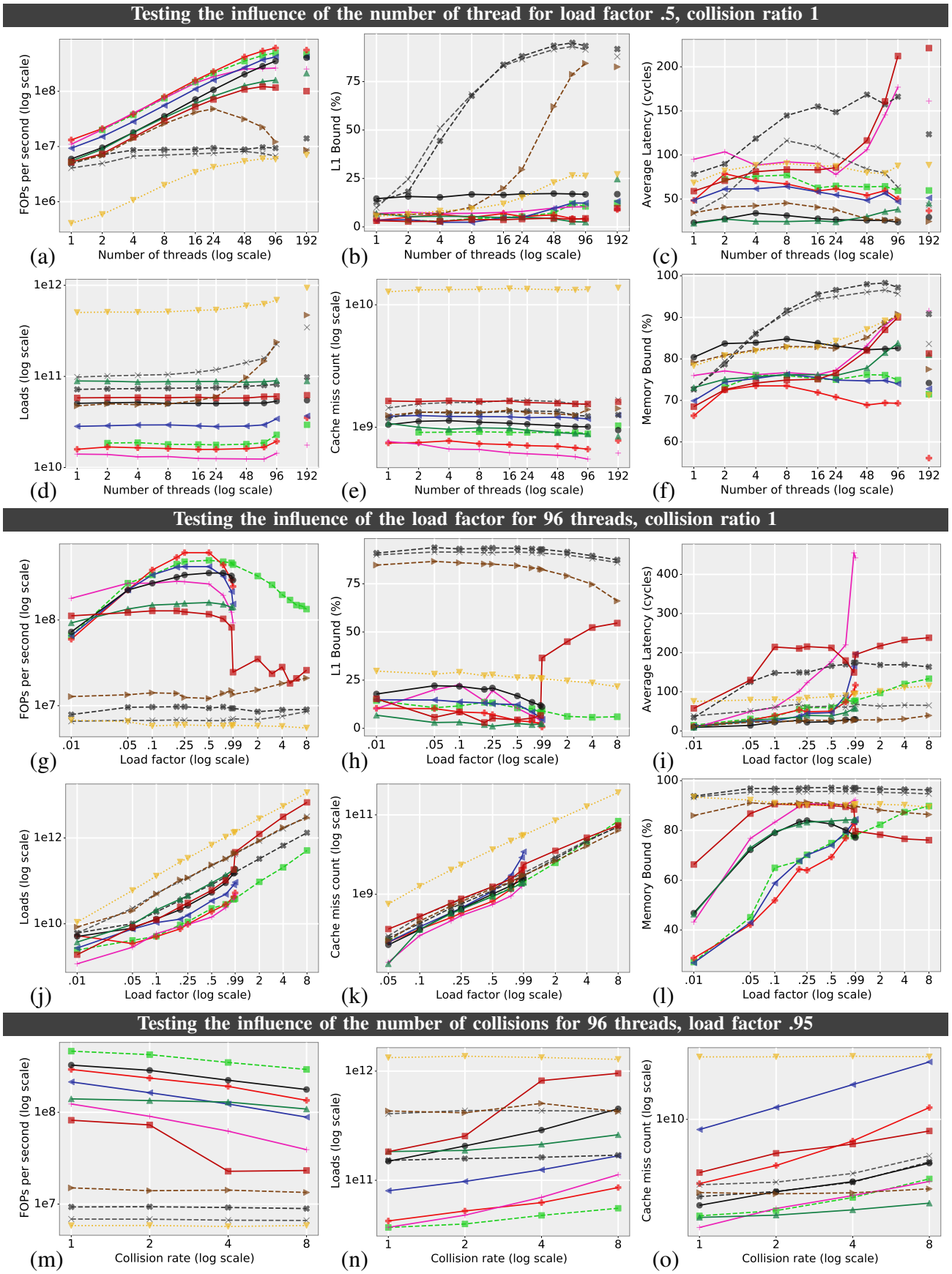
Figure 4 shows the results for scenario 2. Figures 4a, 4g and 4m show the performance of the insertion phase, while varying the number of threads, load factor, and collision ratio, respectively. The other graphs show statistics gathered with Intel® VTune™ Amplifier. We will go through each performance graph separately, explaining the performance by examining the statistics. Note that the *pattern* they show is most important, as they are based on sampling.

For an in-depth explanation of all statistics we refer to the VTune manual, but we explain a few essentials here. *L1 bound* is the percentage of loads serviced by L1 cache. A high value here can indicate high contention. *Average Latency* is the average number of clock cycles a load has to wait. A high number can indicate contention or a large number of accesses to remote (on other CPU) cache or memory. The *memory bound* percentage roughly indicates the amount of time a CPU core is stalled with loads in-flight. A high number can indicate contention or waiting on data from memory.

A. Influence of number of threads

As in scenario 1, TBB does not scale in the number of threads. With the performance statistics, we see that TBB has a much higher L1 bound (4b). According to the manual, this may indicate a high contention. While the implementation uses a lock per bucket, that alone cannot account for the contention [11]. TBB does keep track of the exact size of the hash map using a single atomic integer. All threads modify this single integer for every insert, so this can account for the high L1 bound. We tried TBB without the size counter

Fig. 4 Results for scenario 2, inserting integers into 2^{28} buckets, measuring the insertion phase. For a legend, see Table I.



and this improved the performance, but only by 30%. Using our allocator in TBBA has a small effect, allowing a lower number of loads (4d), but with a higher latency (4c).

Hash map Michael suffers from increased L1 bound with an increased number of threads as well, even causing degradation of performance. Here the cause is not a common element counter, since we explicitly disabled that. We tried Michael with TBB's allocator as well, but this did not change the performance.

The worst performing hash map is Skiplist. It has an order of magnitude higher number of loads and cache misses than most other hash maps. This is expected as the algorithm requires many loads that are often on different cache lines [17].

The reason libcuckoo and Junction:Crude do not gain performance above 48 threads is that the loads suffer from increased latency (4c) and memory bound (4f). This seems to indicate the threads are fighting each other over the cache lines: libcuckoo locks two buckets to perform its cuckoo hashing [13] and Junction:Crude uses linear probing [14], which is susceptible to clustering.

Overall, chaining ChainU performs slightly better than open OpenAddrQCU, even approaching InsituQU.

B. Influence of load factor

Again this is where ChainU scores the highest overall, showing the potential of chaining (4g). Below 5% it is outperformed by a number of hash maps, but above 5% only by the integer-only InsituQU. It outperforms its open addressing counter-part OpenAddrQCU, which features the same allocator, and the main competitor dbsll. For loads above 100% it is roughly an order of magnitude faster.

OpenAddrQCU outperforms dbsll for loads between .05 and .75, but above .75 dbsll has the upper hand. This shows rehashing + linear probing is better able to deal with increased loads than quadratic + linear.

On the opposite side we again see Skiplist suffering from high number of loads (4j) and cache misses (4k). TBB is only slightly ahead in terms of performance. Interestingly, for load factors above 1 the hash map Michael suffers relatively less from L1 bound (4h) and this shows in the performance.

Junction:Crude outperforms all others below 5%. However, for increased loads the performance quickly drops off. Again the load latency (4i) seems the cause. This is interesting, because it has generally the lowest number of loads (4j).

C. Influence of collisions

All hash maps suffer when collisions increase, but Folly handles these the easiest (4m). This seems to be because the number of cache misses is increasing only slightly (4o).

Notable is the jump in performance of libcuckoo from 2 to 4 collision rate. With collision rate 2, there is one collision per bucket, which often can be put into the bucket determined by the alternative hash function. Resolving another collision on the same bucket means rehashing some data. This can be seen in the jump in the number of loads as well (4n).

Overall, chaining and open addressing are similarly affected by collisions in terms of performance.

D. Reflection

Combining the performance graphs of fig. 3 and fig. 4 we see that chaining and open addressing perform similarly for load factors below 1, if we look at ChainU and OpenAddrQCU. For higher load factors, open addressing either cannot compete, e.g. dbsll, or must incur a significant performance penalty to resize the map, e.g. libcuckoo.

The influence of the used allocator is briefly evaluated, but not researched at length. For comparison between chaining and open addressing we use our hash map implementations. These use the same allocator, making the comparison fair in terms of the allocator used.

For comparisons with competitor hash maps, we evaluated the combination as a whole. dbsll has a similar allocator in terms of performance, but can be statically allocated since it only supports constant-sized keys. We tried TBB's scalable allocator in other hash maps, but this did not yield a significant change. Besides TBB, we tried our allocator in libcuckoo, without a significant change either. We also tried our allocator in Michael, but we could not get that to work.

A good algorithm is half the work. For correct high-performance concurrency, the implementation is as important.

IX. Conclusion

We implemented 12 concurrent hash map variants, focusing on insertion and retrieval performance and not supporting deletion or growing. Of these 12 hash maps, 8 support variable-length key-value pairs and 4 support only integers. Additionally, we made an extensive comparison with in total 34 hash maps[7], of which we showed 11. We analyzed what makes a fast concurrent hash map by examining hardware events using Intel® VTune™ Amplifier.

Contrary to what was believed previously [1], chaining lends itself perfectly for a concurrent setting. In fact, the overall best hash map implementation we tested is ChainU, performing similarly to the best open addressing hash map OpenAddrQCU for load factors under 1 and beating competing hash maps by 1.3x–2.6x. For higher load factors, ChainU is an order of magnitude faster than competing hash maps.

A. Future Work

Since the result of this paper is directly applicable to multi-core model checking, it is also applicable to planning [27]. In planning as model checking, plans are generated akin to state space exploration.

We implemented a version of our hash maps that support deletion of entries, but we did not perform our extensive analysis on this. Preliminary findings show the performance is on par, but more analysis is required.

Even without deletion, the chaining hash map presented in this paper will be used in a software multi-core model checker we are currently implementing. We target multi-threaded LLVM IR assembly code. Therefore, the support for variable-length key-value pairs is a requirement in order to support a growing stack and heap.

[7]Data and code can be found at https://github.com/bergfi/hashmap/

REFERENCES

[1] A. W. Laarman, J. C. van de Pol, and M. Weber, "Boosting multi-core reachability performance with shared hash tables," in *Proceedings of the 10th International Conference on Formal Methods in Computer-Aided Design, Lugano, Switzerland*, N. Sharygina and R. Bloem, Eds. USA: IEEE Computer Society, October 2010, pp. 247–256.

[2] ——, "Parallel recursive state compression for free," in *Proceedings of the 18th International SPIN Workshop, SPIN 2011, Snow Bird, Utah*, ser. Lecture Notes in Computer Science, A. Groce and M. Musuvathi, Eds., vol. 6823. berlin: Springer Verlag, July 2011, pp. 38–56.

[3] T. van Dijk, E. M. Hahn, D. N. Jansen, Y. Li, T. Neele, M. Stoelinga, A. Turrini, and L. Zhang, "A comparative study of bdd packages for probabilistic symbolic model checking," in *Dependable Software Engineering: Theories, Tools, and Applications*, X. Li, Z. Liu, and W. Yi, Eds. Cham: Springer International Publishing, 2015, pp. 35–51.

[4] P.-A. Larson, "Dynamic hash tables," *Commun. ACM*, vol. 31, no. 4, pp. 446–457, Apr. 1988. [Online]. Available: http://doi.acm.org/10.1145/42404.42410

[5] R. Pagh and F. F. Rodler, "Cuckoo hashing," *J. Algorithms*, vol. 51, no. 2, pp. 122–144, May 2004. [Online]. Available: http://dx.doi.org/10.1016/j.jalgor.2003.12.002

[6] T. A. David, R. Guerraoui, T. Che, and V. Trigonakis, "Designing ascy-compliant concurrent search data structures," p. 23, 2014, authors appear in alphabetical order.

[7] T. Maier, P. Sanders, and R. Dementiev, "Concurrent hash tables: Fast and general?(!)," *CoRR*, vol. abs/1601.04017, 2016. [Online]. Available: http://arxiv.org/abs/1601.04017

[8] N. L. Scouarnec, "Cuckoo++ hash tables: High-performance hash tables for networking applications," *CoRR*, vol. abs/1712.09624, 2017. [Online]. Available: http://arxiv.org/abs/1712.09624

[9] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.

[10] D. P. Mehta and S. Sahni, *Handbook Of Data Structures And Applications (Chapman & Hall/Crc Computer and Information Science Series.)*. Chapman & Hall/CRC, 2004.

[11] Intel, "Threading building block's concurrent hash map documentation," https://software.intel.com/en-us/node/506191, October 2018.

[12] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman, "Algorithmic improvements for fast concurrent cuckoo hashing," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14. New York, NY, USA: ACM, 2014, pp. 27:1–27:14. [Online]. Available: http://doi.acm.org/10.1145/2592798.2592820

[13] ——, "libcuckoo git repository," https://github.com/efficient/libcuckoo, December 2018.

[14] J. Preshing, "Junction hash map blog post," http://preshing.com/20160201/new-concurrent-hash-maps-for-cpp/, October 2018.

[15] M. M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '02. New York, NY, USA: ACM, 2002, pp. 73–82. [Online]. Available: http://doi.acm.org/10.1145/564870.564881

[16] L. authors, "Libcds git repository," https://github.com/khizmax/libcds, December 2018.

[17] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[18] Facebook, "Folly git repository," https://github.com/facebook/folly/, December 2018.

[19] J. Barnat, P. Ročkai, V. Štill, and J. Weiser, "Fast, dynamically-sized concurrent hash table," in *Model Checking Software*, B. Fischer and J. Geldenhuys, Eds. Cham: Springer International Publishing, 2015, pp. 49–65.

[20] W. Oortwijn, T. van Dijk, and J. van de Pol, "A distributed hash table for shared memory," in *Parallel Processing and Applied Mathematics*, ser. Lecture Notes in Computer Science, R. Wyrzykowski, E. Deelman, J. Dongarra, K. Karczewski, J. Kitowski, and K. Wiatr, Eds. Springer, 9 2015, pp. 15–24, eemcs-eprint-26785.

[21] A. Wijs and D. Bošnački, "Gpuexplore: Many-core on-the-fly state space exploration using gpus," in *Tools and Algorithms for the Construction and Analysis of Systems*, E. Ábrahám and K. Havelund, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 233–247.

[22] R. De La Briandais, "File searching using variable length keys," in *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference*, ser. IRE-AIEE-ACM '59 (Western). New York, NY, USA: ACM, 1959, pp. 295–298. [Online]. Available: http://doi.acm.org/10.1145/1457838.1457895

[23] P. G. Jensen, K. G. Larsen, and J. Srba, "Ptrie: Data structure for compressing and storing sets via prefix sharing," in *Theoretical Aspects of Computing – ICTAC 2017*, D. V. Hung and D. Kapur, Eds. Cham: Springer International Publishing, 2017, pp. 248–265.

[24] M. Areias and R. Rocha, "A lock-free hash trie design for concurrent tabled logic programs," *Int. J. Parallel Program.*, vol. 44, no. 3, pp. 386–406, Jun. 2016. [Online]. Available: http://dx.doi.org/10.1007/s10766-014-0346-1

[25] "Hyper-threading technology architecture and microarchitecture," http://www.cs.virginia.edu/~mc2zk/cs451/vol6iss1_art01.pdf, Intel Corporation, February 2002.

[26] "Linux kernel numa memory policy," https://www.kernel.org/doc/Documentation/vm/numa_memory_policy.txt, October 2018.

[27] F. Giunchiglia and P. Traverso, "Planning as model checking," in *Recent Advances in AI Planning*, S. Biundo and M. Fox, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 1–20.