Model Checking in a Development Workflow: A Study on a Concurrent C++ Hash Table*

Petr Ročkai

Faculty of Informatics, Masaryk University
Brno, Czech Republic
{xrockai}@fi.muni.cz

Abstract. In this paper, we report on our effort to design a fast, concurrent-safe hash table and implement it in C++, correctly. It is especially the latter that is the focus of this paper: concurrent data structures are notoriously hard to implement, and C++ is not known to be a particularly safe language. It however does offer unparalleled performance for the level of programming comfort it offers, especially in our area of interest – parallel workloads with intense interaction.
For these reasons, we have enlisted the help of a software model checker (DIVINE) with the ability to directly check the C++ implementation. We discuss how such a heavyweight tool integrated with the engineering effort, what are the current limits of this approach and what kinds of assurances we obtained. Of course, we have applied the standard array of tools throughout the effort – unit testing, an interactive debugger, a memory error checker (`valgrind`) – in addition to the model checker, which puts us in an excellent position to weigh them against each other and point out where they complement each other.

1 Introduction

Designing correct software is hard and implementing it correctly is possibly even harder. This is especially true of 'plumbing' – low-level code which must be both robust and perform well. Of course, there are established libraries of such code in wide use and considered correct precisely because they are universally used and nobody has found a defect in them for a long time. However, for the same reason, those same libraries are somewhat dated and assimilate new functionality at a very slow rate.

A typical case would be data structures for representing sets and associative arrays. Until 2011, the only implementation available in standard C++ libraries were rebalancing binary search trees. Changes in computer hardware, however, have gradually made data structures based on pointers, like linked lists and search trees, less favourable when compared to more compact, array-like structures. In particular, it is often much better to implement sets and associative arrays using hash tables, even though search trees have, in theory, superior

* This work has been partially supported by the Czech Science Foundation grant No. 18-02177S.

complexity. The 2011 revision of the C++ standard has seen the inclusion of `unsorted_set` and `unsorted_map` container classes, which are represented using a hash table. Unfortunately, the design of those container classes is such that a conforming implementation needs to use *open hashing* and must not invalidate references to items while rehashing the table, strongly suggesting chained buckets. None of these design choices are particularly suitable for modern processors, though they do make the hash table easier to use.[1]

The remainder of the paper is organized as follows: the rest of this section highlights the contributions of this paper and surveys the related work, while Section 2 outlines the basic premises of the paper, including the basic methodology and tools which we used. Section 3 then gives a high-level overview of design criteria and design choices we have made for the hash table. Section 4 describes the implementation and verification process in detail – one subsection for each development iteration, where each of the iterations constitute 2-3 person-days of combined programming, debugging and verification effort. Finally, Section 5 summarizes and concludes the paper.

1.1 Contribution

The main goal of this paper is to describe the experience of using a comparatively heavyweight model checker in an otherwise lightweight development process. The main takeaways are the following:

1. It is surprisingly easy to use model checking with self-contained C++ code.
2. While automated, exhaustive verification of source code seems quite remote, using a model checker has real, practical benefits in day-to-day development.

We also hope that the moderately detailed description of our development process can serve as a mostly positive example of applying formal methods in the trenches of rather low-level programming.

1.2 Related Work

Formal methods have been the subject of steady interest from both academia and industry, as evidenced by a substantial body of surveys on their applications, e.g. [4, 16]. A number of case studies have been performed and published, in many instances with production software. Unlike the present work, most of the existing studies focus on specific mission-critical software – flight control [3], satellite control systems [5], particle accelerators [6, 8] or real-time operating system kernels [10]. A unique effort in this area is seL4 [7], which is, however, based on heavyweight methods centred around theorem proving.

The most closely related study to ours is perhaps [8], since it involves directly applying model checking to a C++ implementation, and employs the same model

[1] In particular, they provide nicer iterator and reference invalidation semantics and are less susceptible to pathological behaviour when using sub-optimal user-supplied hash functions.

checker – DIVINE – for this task as we do. The system examined, however, was quite different, and included a separate model for checking liveness (LTL) properties using SPIN.

2 Preliminaries

In this section, we first give a very brief overview of hash tables and their design criteria, then of testing and debugging, and finally of model checking software at the implementation level.

2.1 Hash Tables

A hash table is a data structure that represents an (unordered) set, in which *keys* can be *looked up* and into which new keys can be *inserted*. Besides encoding sets, hash tables can be straightforwardly extended to encode associative arrays though we will only discuss the simpler case of sets in this paper. Both the abovementioned operations are, on average, in $\mathcal{O}(1)$ – the average number of steps is a constant. Of course, this complexity depends on the properties of the *hash function* which the hash table uses and even with very good hash functions, it is always possible to construct a sequence of operations that will exhibit pathological behaviour (i.e. individual operations running in linear time). In the remainder of this paper, we assume that a suitable hash function is available.

An important aspect of modern data structure design is the safety of *concurrent access*,[2] which stems from properties of contemporary hardware: high computation throughput can only be achieved with specific designs which minimize *contention*. Luckily, in normal operation, hash tables naturally cause only minimal contention. Of course, this assumes that there are no global locks or some other heavyweight synchronization mechanism: either each cell, or at most a small segment of cells, needs to be protected from *race conditions* individually. Operations that access multiple cells of the hash table need to be carefully designed to not require additional communication to avoid races.

2.2 Testing and Debugging

Testing is the natural backbone of any validation or verification effort – examples are the most intuitive tool for understanding the behaviour of processes and systems. In the case of data structure design, the majority of testing is usually done at the unit level – the programmer writes down small example programs which exercise the data structure and inspects the results for conformance with their expectations.

The most obvious and immediate problem that arises with (unit) testing is concurrency. A single (sequential) unit test is a very concrete entity and is easy

[2] Here, concurrency means that multiple CPU cores perform operations on the same data structure without additional synchronization.

to work with and argue about. Unfortunately, test scenarios that involve concurrency lack this concreteness: there is now an implicit quantifier over all possible reorderings of concurrent actions. Every time we execute the test scenario, we see a different ordering, and many types of problems will only appear in some such orderings, but not in all of them. Due to the nature of concurrent programming systems, it is usually also very hard to reproduce the exact ordering that led to an error. In our effort, we have used a software model checker to deal with the quantification over allowable event reorderings (for more details see Section 2.3).

Even with sequential test cases, the root cause of a failure is not always obvious: even short test cases can generate a fairly long sequence of steps, depending on the internal complexity of the operations under test. There are two basic techniques to clarify the sequence of steps that led to a failure, in addition to mere inspection. First, tracing statements can be inserted into the program under test, allowing the programmer to more easily follow the execution of the program. Information about which branches were taken and the values of key variables are usually included in such traces.

Second, the user can inspect the execution using an interactive *debugger*, such as `gdb` [14]. In this case, the execution can be stopped at various points, the user can instruct the debugger to only proceed with execution in small, incremental steps and can inspect values of the variables at any point in the execution. Usually, it is only possible to step or execute the program forward in time, though extensions exist to allow *reversible* debugging [15, 9] – the ability to step back, to a point in the execution that was already visited once. While an interactive debugger is an invaluable tool, it is in some sense also the tool of last resort: it is the least amenable to automation, and modifications in the program require a possibly long sequence of interactions to be repeated.

2.3 Model Checking

There are few choices when it comes to applying formal methods to C++ implementations of concurrent data structures. Heavyweight formal methods often require the design to be 'implemented' in a special language (in addition to the C++ implementation). The special language in question might be a protocol modelling language (e.g. ProMeLa), or it might be a proof language (e.g. Coq), but in either case, there is non-trivial effort involved in the translation of the informal (working) description into this language.

Additionally, this special language is overwhelmingly likely to be unfamiliar and hence pose additional challenges, including an increased risk of mistakes. Even in proof languages, it is essential that the semantics of the theorems that are being proved are properly understood, otherwise it might easily happen that they do not correspond to the informal properties that we were interested in.

Finally, there would be no guarantee that the C++ version, which is what will be actually executed, corresponds to the one that has been verified in any meaningful way. While there has been an effort in the seL4 project to extend the proofs to cover machine code [11], anything of this sort is still extremely far-fetched for code that relies on the extensive C++ runtime libraries.

For these reasons, we are more interested in comparatively lightweight methods which can directly work with the implementation. There are, obviously, significant gaps: such tools tend to be more complex (especially when considering correctness-critical cores), their semantics are less rigorous and there is inevitably a disconnect between the assumed and the actual semantics of the underlying hardware.

In our case, model checking for safety properties appears to be the most appropriate choice. The technique is very similar to testing (among their other shared attributes, it is a dynamical method) but with efficient handling of universal quantification – either over inputs, over event reorderings or over possible interactions with the environment.

There is a wide selection of (safety) model checking tools based on LLVM or on CBMC which both have usable C++ frontends. Unfortunately, the latter does not support modern C++ features and the tools based on the former often lack support for concurrency (and exceptions). The one tool which fits all mentioned criteria is DIVINE [1], even though it also has a number of limitations. Verification of parallel programs with DIVINE is rather resource-intensive, since it enumerates the entire state space. Even though it employs various state space reductions [13], the state spaces for small programs with only 2 threads are rather large. In the verification tasks we have performed, we were limited to 100 GiB of RAM, an amount which unfortunately proved rather constraining.

Another limitation that we encountered in DIVINE is that enabling support for weak memory models causes an additional blowup in the size of the state space. However, our implementation only uses sequentially consistent atomic operations, which hopefully means that this is not a significant issue.

2.4 Assertions

Assertions are a programming aid where a statement in the source code describes a condition which the programmer expects to hold in every execution through the given source code location. The program then checks, at runtime, that this condition is indeed satisfied and aborts execution when the check fails. Assertion statements are commonly used in testing and are compiled out of the program in production builds. Depending on the number and quality of the assertion statements, they can range from a sporadic sanity check to comprehensive pre- and post-condition annotations reminiscent of inductive verification.

Besides testing, assertion annotations are also extremely useful in model checking, since the verifier can easily execute the runtime check and ensure that it holds true on every execution covered by the quantification provided by the model checker (whether it is over event reorderings or input values).

3 Design

The following points summarize our upfront design choices for the hash table (i.e. the choices were made before the implementation work started).

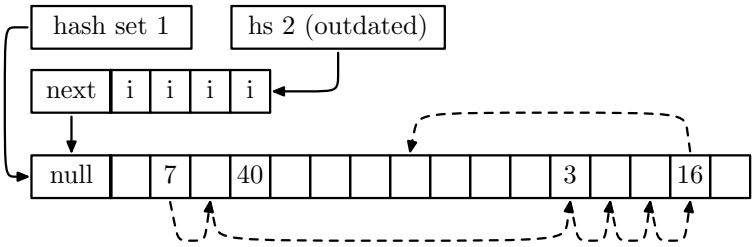

Fig. 1. Example of a pair of `hash_set` structures (each belonging to a different thread), sharing a single list of `hash_table` instances. The dashed arrows represent the lookup sequence for an item with the hash value 1. Cells marked 'i' are invalid.

1. The hash table will be stored as a flat array of *cells*[3] using open addressing and will use a cache-friendly combination of linear and quadratic probing for collision resolution.
2. There will be multiple cell implementations for different scenarios and the table will be parametrized by the cell type:
   - a low-overhead cell for sequential programs,
   - lock-free cells for small keys based on atomic compare & exchange,
   - lock-based cells for medium-sized keys.
3. The table will employ on-demand, parallel rehashing, so that it can be used in scenarios where the number of distinct keys is not known upfront.
4. It will implement key removal based on tombstones. The ability to shrink the table (automatically or manually) is not a priority.

Our design was informed by the considerations outlined in Section 2.1 and by our past experience with hash table design [2]. To simplify our task, we assume that keys have a fixed size and are small (i.e. they are integers or pointers or that a single key consists of at most a few pointers) and expect the user to use indirection as appropriate to store larger or variable-sized keys. Together, these criteria naturally lead to *open addressing* in a flat array of cells. A hash table based on open addressing resolves the inevitable hash collisions by *probing*: computing a sequence of indices instead of just one, and trying each index in turn, until it either finds the correct key or an empty cell.

This immediately brings a few problems: the cell must be able to distinguish a special 'empty' state, which is distinct from all possible key values. In some circumstances, this could be a specific key picked by the user, but this approach is fragile and not very user-friendly. Additionally, another special state is required for tombstone-based key removal.

[3] Each cell can hold at most a single key. Basic operations on a cell include storing a key and comparing the content of the cell with a key.

4 Development and Verification

The implementation and verification work was done in 4 iterations, each iteration including both programming and testing and/or model checking activities. Each iteration corresponds to 2–3 person-days of effort, adding up to approximately 2 weeks (10 days).

4.1 First Iteration

In this phase, we have done the programming equivalent of a 'first draft', knowingly omitting part of the expected functionality and using sub-optimal, but known-correct (or at least presumed correct) primitives from the standard library. We have also re-used pieces of code from previous hash table designs, especially for cell implementation. The main goal in this iteration was reduction of memory overhead associated with our previous design.

The implementation after the first iteration uses a per-thread object which only consists of a single `std::shared_ptr` (2 machine pointers) and a flat table with the cells and additional 5 machine words of metadata (the size, a rehashing counter, another `std::shared_ptr` and finally the reference count managed by `std::shared_ptr`). By the end of the fourth iteration, the per-thread object was reduced to a single machine pointer and the per-table metadata to 4 words.[4]

Initial Implementation The code is split into 3 layers:

1. the top-level class template is `hash_set`[5] and uses a simpler, fixed-size `hash_table` to implement the actual storage, lookup and insertion of keys
2. `hash_table` is a compact structure (i.e. it is laid out contiguously in memory and does not use pointers or references) whose main part is an array of *cells*
3. there are multiple implementations of the *cell* concept and each can hold a single key and possibly a small amount of metadata

The `hash_set` class is responsible for providing an interface to the user and for managing the capacity of the hash table: when the `hash_set` runs out of space, a new, bigger `hash_table` is constructed and the content of the previous one is rehashed into the new one. The individual `hash_table` instances form a linked list – there might be an arbitrary number of them, since some threads may fall behind others and hence keep the old instances alive (see also Figure 1).

For efficiency reasons, the linked list must be lock-free and concurrent-safe: checking the existence of an item linked to the currently active `hash_table` instance is used to determine whether another thread has initiated rehashing.

[4] In comparison, the design from [2] uses 3 machine words per thread, 64 + 32 words of fixed overhead per a hash table instance and 11 words per cell vector. It also incurs 3 indirections compared to the single indirection in the current design.

[5] The class naming has changed during later development stages. We use the last iteration of the class names throughout the paper to avoid confusing the reader.

In this iteration, a standard reference-counted smart pointer (`std::shared_ptr`) was used for memory management. A standard atomic compare-and-swap operation on those pointers was then used to implement the linked list in a concurrent-safe fashion.

The rehashing is done in contiguous segments of the original `hash_table`,[6] each segment being rehashed as a single unit by a single thread. Allocation of segments to rehash to threads is coordinated using a pair of atomic counters, one counter in each `hash_table` instance involved in the rehashing.

Finally, a separate 'sequential' variant of the `hash_set` is included in the implementation. Since it does not need to provide safe concurrent access, it is somewhat simpler and also slightly faster.

The first iteration resulted in 820 lines of C++ (discounting unit tests), distributed as follows: the cell implementations took 220 lines, utility classes, shared interface code and other helpers 110 lines, the sequential implementation of `hash_set` was 170 lines and its concurrent counterpart was 280 lines (including `hash_table`). The remainder went to high-level comments and type alias declarations.

Verification (round 1) The initial implementation was only subject to unit testing and to stress testing via concurrent application code (i.e. no model checking was performed in this iteration). No ill effects were observed during the initial testing run (though a number of problems were uncovered in later iterations). The hash function was more or less assumed correct (however, the only real requirement is that it is a function in the mathematical sense).

The implementation itself contains 7 consistency assertions which were checked during unit testing, in addition to the checks performed by the tests themselves. Those assertions were included for multiple reasons:

- first of all, they serve as machine-checked documentation, informing the reader about local pre- and post-conditions and about assumptions the author of the code has made,
- assertions make initial development easier, by quickly alerting the programmer about unwarranted assumptions and about mistakes in code composition (where e.g. a pre-condition of a subroutine is violated by a caller),
- finally, they are checked during unit tests and model checking, serving as additional checks of correctness.

We also include a few statistics about the unit tests:

- the unit testing code (written against the interface of the `hash_set` class template) consisted of approximately 250 lines of C++,
- there were 6 sequential test scenarios and 4 concurrent access scenarios,
- the sequential implementation was subjected to the sequential test cases with 2 sequential cell types,

[6] Clearly, the keys stored in a continuous segment of the smaller `hash_table` may end up distributed arbitrarily in the entire range of the bigger (target) instance.

– the concurrent implementation with 2 concurrent cell types were subjected to both the sequential and the concurrent test cases,
– this added up to 32 unit test cases; code coverage was not measured.

The concurrent test cases were mainly of the stress-testing type: they performed many operations in parallel in the hopes of more or less blindly hitting problems. This approach, while rather inelegant, tends to work for 'shallow' or high-probability bugs, but overall, it is not a very reliable method.

4.2 Second Iteration

The main driver of the second iteration was unification of the code between the sequential and concurrent-safe implementations of the `hash_set` class. The comparatively expensive rehashing protocol is statically disabled in the sequential variant.

Implementation Changes The sequential implementation of `hash_set` has been made a special case of the concurrent variant, with certain paths through the code cut short: this change saved about 150 lines of code and a substantial amount of duplication, leading to cleaner and more maintainable code. Overall, by the end of this iteration, the implementation was down to 680 lines of code, a net reduction of 140 lines.

This iteration has also seen improvements to the initially crude interaction with the client code (especially in case of errors). In particular, a failure to rehash the table is now reported as an exception and does not abort the entire program. There are two reasons why rehashing may fail: either due to insufficient uniformity of the hash function (in which case, aborting the program might be appropriate, since it is really an implementation error in the client code) or due to adversarial inputs that cause excessive collisions. In the latter case, throwing an exception is much more appropriate, since the circumstance is likely outside the control of the client application.

Verification (round 2) Due to the merge of the two `hash_set` implementations, one of the consistency assertions was removed (having become redundant), leaving 6. In addition to the assertions, tracing statements were added to the implementation to ease analysis of test failures and counterexamples from model checking: the trace output shows up in the counterexample traces from the model checker, which makes the said counterexamples much easier to understand, even without using the interactive simulator [12]. Like consistency assertions, the traces are kept in the code permanently, but compiled out in release builds.

The unit tests in this iteration were extended with cases of longer (68 byte long) keys, which uncovered errors in the sequential cell types, which were in turn fixed. Additionally, a problem was discovered through a combination of unit testing and the use of `valgrind`. In this case, the unit tests were executed on a different platform than usual and exhibited sporadic failures affecting test

cases which used the atomic cell type, indicating a race condition or a memory error. Since the latter are easier to diagnose, the unit test code was first executed in `valgrind`, which quickly pinpointed the problem to a read of uninitialized memory. The problem turned out to be due to padding bytes which the compiler in some cases inserted into the implementation of the atomic cell. Because the entire cell was accessed using a single atomic compare & exchange operation, padding bytes were included in the comparison, leading to failures.

Besides the padding problem, a few edge cases were found in the implementation of the atomic cell which warranted a minor redesign of the metadata stored in the cell and the operations on them. In the new version, 2 bits are reserved to encode special cell states: empty, invalid,[7] tombstone and full.

This was also the first iteration to use model checking for verification, with a small initial set of scenarios to be considered. In each case, there are 2 threads,[8] both accessing a single hash table, with each thread performing a fixed sequence of operations (including fixed arguments), but arbitrary reorderings of those operations are considered by the model checker. The scenarios included in this iteration were following:

- a check for correctness of insertion concurrent with rehashing running in the other thread (without the insertion triggering another resize),
- a case running an erase concurrent with insertions in the other thread,
- same, but an insert also triggered a rehashing concurrent with the erase.

No new counterexamples were found at this time, but the effort helped to validate the fixes that were done in the atomic cell implementation based on problems uncovered via testing.

4.3 Third Iteration

The third iteration was comparatively minor, with focus on application-level testing and changes and improvements in the interface provided to the client code. Minor mistakes caused by rearranging the code were caught by existing unit tests and corrected.

Implementation Changes This iteration included, besides a few small optimizations in the key lookup code, mainly changes to the outside class interface: the growth pattern (sequence of `hash_table` sizes to use) was made into a static (compile-time) parameter and the first iteration of the `adaptor` interface was added. Among other things, this change further reduced the memory overhead of `hash_set` in cases where the user-provided object which performs hash computation contains data members. At the end of this iteration, the code was 685 lines long (a net growth of only 5 lines).

[7] Cells are invalidated when they are rehashed, so that concurrent insertion will not accidentally use an empty cell that has already been 'moved' to the next generation leading to a loss of the inserted key.

[8] Unfortunately, model checking of scenarios with more than 2 threads does not seem to be realistic, at least not with the allocated budget of 100 GiB of RAM.

Verification (round 3) The verification effort in this iteration was centred around a first major push in application-level testing: the hash table was used in an application which does a mix of key insertions with both new keys and with keys already present in the hash table. These keys are indirect – the hash table stores a pointer, and an adaptor is provided to compare the actual structured, variable-size keys stored behind the pointer. The application was then exercised in both single-threaded and in multi-threaded configurations.

During this testing, an error was discovered in the rehashing code, where rehashing failures were not properly detected (instead, the code went into an infinite loop making the hash table ever larger). Rehashing failures happen when the table is used with a poor hash function which cause collisions that do not get remapped as the hash table grows (i.e. they are collisions which are already present in the hash output and are not introduced by scaling the range of the hash function to the current size of the hash table). This was detected thanks to a bug in the custom hash function which was part of the application code.[9] Additionally, with the code path exposed to testing, a deadlock was discovered in a previously unreachable error path in the same area of code.

Both errors were tracked down with the help of tracing and `gdb` (which quite easily provided backtraces at the location of the deadlock once it could be reproduced). It is also worth noting that even if the path was covered in model checking scenarios (by using an intentionally conflict-prone hash function), the deadlock was caused by spin-based synchronization – a type of error which is rather hard to specify in current versions of DIVINE.[10]

Finally, no new test cases or model checking scenarios were introduced in this iteration, though in the manner of continuous integration, existing unit tests were executed after each self-contained change to the code.

4.4 Fourth Iteration

The last iteration reported in this paper brought a number of substantial changes in the internals of the hash table, even though only indirectly. The well-tested `std::shared_ptr` was replaced with an *intrusive pointer*[11] which did not yet support atomic compare & exchange. This operation was added along with more basic support for atomic access (atomic loads and stores of the pointer, and support for manipulating reference counts atomically). Neither unit testing nor application-level testing uncovered any problems in this code.

In addition to the changes in internals, the interface of the `hash_set` class was finalized in this iteration, providing the desired level of flexibility and ease of use. The verification part of the effort was focused on correctness in general and uncovering possible concurrency problems in particular.

[9] This incident also revealed a weakness in our testing methodology, where only 'good' hash functions were used during unit testing and model checking – cf. Section 4.1.

[10] Deadlocks involving `pthread` mutexes are detected, but these mutexes are too expensive to be used in concurrent data structures.

[11] A pointer which can only point at objects which manage their own reference counter.

Implementation Changes As outlined above, there were two main avenues of change in this iteration. The first group of changes was directed at the `hash_set` interface:

1. the iterators which allow enumeration of values stored in the hash table were slightly simplified,
2. the `adaptor` now offers the capability to store data outside of the hash table proper, opening way to alternate collision resolution methods, which were validated by creating an experimental concurrent-safe linked-list bucket implementation.[12]

The other major change was the migration away from `std::shared_ptr`:

- unlike `std::shared_ptr` which is represented as a pair of machine pointers, the intrusive reference-counting pointer (`refcount_ptr`) only needs one, reducing the memory overhead
- `refcount_ptr` provides low-contention (even if not entirely atomic) compare & swap operation – the least-significant bit of the pointer itself is used as a lock while the reference counts are rearranged

Finally, it was noticed that in certain cases, the sequential cell types could contain uninitialized values – this defect was also corrected. The final implementation spans 730 lines of code, a net increase of 45 lines, mainly attributable to the changes in the `adaptor` interface.

Verification (round 4) As mentioned above, the main focus of this iteration was model checking. We have mostly focused on three instances, which all allowed for runs, on which an insertion concurrent with rehashing could trigger another rehashing of the table. The variants were as follows:

1. only insertions, with exactly 1 concurrent insert on one of the threads
2. same as above but with a call to erase sequenced after the isolated insertion,
3. another insert-only scenario where both threads perform 5 fully concurrent insertions in opposite orders, exposing further arrangements of concurrent inserts triggering rehashing[13]

The main 'workhorse' of the model checking effort was the first scenario. In most cases, it took about 2 hours to produce a counterexample, and consumed about 40 GiB of RAM. The second scenario did not differ substantially from the first, while the third took almost 4 hours. We have uncovered a number of problems during this endeavour:

[12] The implementation was tested, but not model checked, and is not part of the hash table implementation (it was done in application-level code in an application-specific manner).

[13] All the scenarios used a `hash_set` with the growth pattern 2-4-8. The last scenario was however also attempted with the pattern 1-2-4, reducing the size of the state space and memory requirements considerably, from 85 GiB to about 15 GiB.

– the initial lock-free implementation of compare / exchange on refcount_ptr was wrong (it could cause one of the objects to be freed prematurely due to a race condition),
– the contended case of a helper function, `atomic_add_if_nonzero` was wrong, updating the pointer even if it was, in fact, zero (though the return value correctly indicated that the counter reached zero),
– a race condition in the rehashing protocol (during a concurrently triggered rehashing).

Interestingly, all three problems uncovered in model checking were caught by the same consistency assertion in the rehash protocol during model checking of the same scenario (the first one in the above list). Since the model checker only provides a single counterexample, the problems were detected sequentially: the first counterexample pointed at the reference count problem; after that was fixed, another counterexample cropped up due to the incorrect `atomic_add_if_nonzero`, and finally after this problem was also fixed, the race condition in the rehash protocol surfaced.

Finally, after the initial fix to the race condition passed model checking (i.e. no more counterexamples were reported), a deadlock was quickly found during application-level testing. After it was confirmed that the race fix introduced the deadlock, the problem was quickly analysed and corrected, partially because the change was in a single line. The final fix for the race (which no longer caused the deadlock) was confirmed by running the model checking scenarios again. No further problems were detected.

4.5 Discussion

From the point of validation and verification, the most useful way to split the code is along the sequential – concurrent axis. The sequential code is rather trivial – almost all bugs were quickly discovered via unit testing and fixed almost immediately after being introduced. The one exception to this rule was the uninitialized memory read that was discovered and fixed in iteration 2. What is noteworthy about this bug is that unit tests exercised this part of the code quite heavily, but since we execute each unit test in a new process, the test cases reliably encountered zeroed memory, which masked the problem. Automatically running the unit tests through `valgrind` would have alerted us to the problem earlier and we will consider altering our work flow to this effect in the future.

The concurrent case is very different, even though the code is still comparatively simple on its surface and some of the concurrent behaviours can still be tested with success: specifically, problems on paths that are commonly taken can be often uncovered with test cases that do not much differ from the sequential instances. However, concurrent code often contains paths which are only traversed under heavy contention (which is hard to trigger in traditional testing), illustrated e.g. in the `atomic_add_if_nonzero` problem. In this case, the detection was further hampered by the fact that the contention had to coincide with zeroing of a counter, which only happens at most once for each such counter.

The second source of hardness in concurrent code is 'interaction at a distance'. This was the case with the race condition in the rehashing protocol: two distinct synchronization handshakes on the same variable (number of segments to be rehashed) accidentally matched up. In testing, a thread never woke up at exactly the right moment to trigger the problem, even though unit tests triggered a number of concurrent rehash operations. Code inspection did not uncover this problem either – each handshake considered separately appears to be correct.

5 Conclusions

Like other authors, we have found that model checking can be a valuable tool despite its numerous limitations. Our use case was perhaps somewhat unique in the level of integration of model checking into the development process and in the low-level nature of the code to which it was applied. Our experience shows that testing and model checking nicely complement each other and that employing both can mitigate some of their individual drawbacks. One feature of the model checker which was essential in our endeavour was its ability to work with unrestricted and annotation-free C++ code – in this sense, model checking was entirely transparent. Preparing inputs was essentially the same as writing unit tests, the only difference being that the cases must be *small*: running only a few operations over small data structure instances. Fortunately, this 'smallness' does not seem to limit the ability of the model checker to uncover problems.

In our case, the most important weakness of model checking was, in agreement with previous accounts, its high resource consumption and substantial delay between asking a question and obtaining an answer. Sometimes, a simpler tool can do the job: for instance `valgrind` can often detect similar classes of programming errors. However, it only works well with deterministic, sequential programs where the universal quantification over event reordering provided by the model checker is not required.

Finally, interpretation of counterexamples was challenging, though not disproportionately so when compared to traditional debugging. The two tools we used the most in this context was ability to add text-based tracing (without having to worry about reproducing the problem at hand), and the interactive simulator, which allowed us to step through a single fixed counterexample.

Bibliography

[1] Z. Baranová, J. Barnat, K. Kejstová, T. Kučera, H. Lauko, J. Mrázek, P. Ročkai, and V. Štill. Model checking of C and C++ with DIVINE 4. 2017.

[2] J. Barnat, P. Ročkai, V. Štill, and J. Weiser. Fast, dynamically-sized concurrent hash table. In *Model Checking Software (SPIN 2015)*, volume 9232 of *LNCS*, pages 49–65. Springer, 2015. ISBN 978-3-319-23403-8. doi: 10.1007/978-3-319-23404-5_5.

[3] Z. Chen, Y. Gu, Z. Huang, J. Zheng, C. Liu, and Z. Liu. Model checking aircraft controller software: a case study. *Software: Practice and Experience*, 45(7):989–1017, 2015. doi: 10.1002/spe.2242.

[4] J. Fitzgerald, J. Bicarregui, P. G. Larsen, and J. Woodcock. *Industrial Deployment of Formal Methods: Trends and Challenges*, pages 123–143. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-33170-1. doi: 10.1007/978-3-642-33170-1_10.

[5] X. Gan, J. Dubrovin, and K. Heljanko. A symbolic model checking approach to verifying satellite onboard software. *Science of Computer Programming*, 82:44 – 55, 2014. doi: https://doi.org/10.1016/j.scico.2013.03.005.

[6] Y. L. Hwong, J. J. Keiren, V. J. Kusters, S. Leemans, and T. A. Willemse. Formalising and analysing the control software of the CMS experiment at the large hadron collider. *Science of Computer Programming*, 78(12):2435 – 2452, 2013. doi: 10.1016/j.scico.2012.11.009.

[7] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *SOSP*, pages 207–220. ACM, 2009. doi: 10.1145/1629575.1629596.

[8] J. Lång and I. S. W. B. Prasetya. Model checking a C++ software framework, a case study. 2019. doi: 10.1145/3338906.3340453.

[9] R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush. Engineering record and replay for deployability, 2017. arXiv:1705.05937.

[10] J. Penix, W. Visser, E. Engstrom, A. Larson, and N. Weininger. Verification of time partitioning in the DEOS scheduler kernel. In *International Conference on Software Engineering*, pages 488–497. ACM Press, 2000.

[11] D. Potts, R. Bourquin, L. Andresen, J. Andronick, G. Klein, and G. Heiser. Mathematically verified software kernels: Raising the bar for high assurance implementations. Technical report, NICTA, Sydney, Australia, 2014.

[12] P. Ročkai and J. Barnat. A simulator for LLVM bitcode. 2017. URL `https://arxiv.org/abs/1704.05551`. Preliminary version.

[13] P. Ročkai, J. Barnat, and L. Brim. Improved state space reductions for LTL model checking of C & C++ programs. In *NASA Formal Methods*, volume 7871 of *LNCS*, pages 1–15. Springer, 2013.

[14] R. Stallman, R. Pesch, and S. Shebs. Debugging with gdb. 2010.

[15] A.-M. Visan, K. Arya, G. Cooperman, and T. Denniston. URDB: a universal reversible debugger based on decomposing debugging histories. In *PLOS '11*, 2011.

[16] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4):19:1–19:36, 2009. doi: 10.1145/1592434.1592436.