DRAM BASED PARAMETER DATABASE OPTIMIZATION

Thesis

Author: Tadas Marcinkevicius

Supervisor: Jonas Löfgren

Examiner: Assoc. Prof. Johan Montelius

2012-06-28
Abstract

This thesis suggests an improved parameter database implementation for one of Ericsson products. The parameter database is used during the initialization of the system as well as during the later operation.

The database size is constantly growing because the parameter database is intended to be used with different hardware configurations. When a new technology platform is released, multiple revisions with additional features and functionalities are later created, resulting in introduction of new parameters or changes to their values. Ericsson provides support for old and new products.

The entire parameter database is currently stored in DRAM memory as a hash map. Therefore an optimal parameter database implementation should have low memory footprint. The search speed and initialization speed for the target system are also important to allow high system availability and low downtime, since a reboot is a common fix for many problems. As many optimizations have to consider memory size – speed tradeoff, it has been decided to give preference to reducing memory footprint.

This research seeks to:

- Analyze data-structures suitable for parameter database implementation (Hash map, Sparsehash, Judy hash, Binary search tree, Treap, Skip List, AssocVector presorted using std::map, Burst trie).
- Propose the best data-structure in terms of used memory area and speed.
- If possible, further optimize it for database size in memory and access speed.
- Create a prototype implementation.
- Test the performance of the new implementation.

The results indicate that a more compact database implementation can be achieved using alternative data structures such as Presorted AssocVector an Sparsehash, however some search speed and build speed is lost when using these data structures instead of the original Gnu Hash Map implementation.
# Table of contents

1. INTRODUCTION..........................................................................................................................1
   1.1. AIM AND RESEARCH QUESTIONS................................................................................................2
   1.2. PROBLEM.......................................................................................................................................2
   1.3. HARDWARE ENVIRONMENT AND DATABASE CHARACTERISTICS.................................3
   1.4. METHOD.........................................................................................................................................4

2. BACKGROUND.....................................................................................................................................8
   2.1. GNU HASH MAP DATA STRUCTURE, USED IN THE ORIGINAL DATABASE IMPLEMENTATION......8
   2.2. ASSOCVECTOR................................................................................................................................12
   2.3. BURST TRIE..................................................................................................................................14
   2.4. BINARY SEARCH TREE (BST)......................................................................................................18
   2.5. TREAP........................................................................................................................................18
   2.6. SKIP LIST.....................................................................................................................................20
   2.7. BACKGROUND FINDINGS SUMMARY...........................................................................................21

3. TESTED DATA STRUCTURE IMPLEMENTATIONS...........................................................................26
   3.1. PRESORTED ASSOCVECTOR.......................................................................................................26
   3.2. SPARSEHASH................................................................................................................................27
   3.3. JUDY HASH..................................................................................................................................28

4. DATA STRUCTURE TEST RESULTS...............................................................................................30
   4.1. MEMORY SIZE COMPARISON.......................................................................................................30
   4.2. PARAMETER SEARCH TIME COMPARISON................................................................................31
   4.3. THE SEARCH TIME DISPERSION COMPARISON...........................................................................33

5. THE CONCLUSION............................................................................................................................35
   5.1. JUDY HASH...................................................................................................................................36
   5.2. PRESORTED ASSOCVECTOR.......................................................................................................36
   5.3. SPARSEHASH................................................................................................................................37

6. FUTURE WORK.....................................................................................................................................38
   6.1. PROPOSITION 1: PARTIAL KEY COMPRESSION.................................................................................38
   6.2. PROPOSITION 2: CLEANING THE DATABASE CONTENTS...............................................................42

7. REFERENCES.........................................................................................................................................45
1. Introduction

This thesis suggests an improved parameter database implementation for one of Ericsson products. The parameter database is used during the initialization of the system as well as during the later operation.

The database size is constantly growing because the parameter database is intended to be used with different hardware configurations. Those changes in hardware configuration result in new or changed parameters. Since new products are presented by Ericsson, while support for old as well as new products is maintained, the size of parameter database grows.

Only parameters relevant to the configuration in use are needed for any particular unit, but with such a large number of different configurations it is impractical to create separate databases for each product as it would result in confusion and additional difficulties for the client who would need to keep track of all the versions and use the correct database. Instead all the parameters are held in one database and the correct parameters are identified automatically.

The entire parameter database is currently stored in random access memory (DRAM) as a hash map. Therefore an optimal parameter database implementation should have low memory footprint. The search speed and initialization speed for the target system are also important. Ericsson targets to have very high system availability and low downtime, so even the switch-on and off time matters greatly. After all, a reboot is a common fix for many problems. As many optimizations have to consider memory size – speed tradeoff, it has been decided to give preference to reducing memory footprint.

The switch-on time would depend on the time it takes to build the parameter database in DRAM, as well as the database search speed, because the parameter database is most heavily used during the initialization of various system components. The operation performance would only be affected by the search speed. However the database build time mostly depends on data insertion speed, which may be governed by other factors then the search speed, thus some database architectures may excel in one criteria but not the other. The common consensus towards speed is to choose the database that would simply save the most time, regardless by which criteria it does that. Overall the speed requirement is not as important as the memory footprint reduction requirement. Improving the speed is desired but not essential.
Since all the parameters for every possible hardware configuration are currently stored in the database, a sorted manner of storing the data in the database would also be desired, as it could ease the process of locating relevant data and improve the readability of the database.

The database memory size, speed and other characteristics are determined by the underlying data structure. Analyzing the existing database implementation and comparing it to alternative data structures could suggest a way to improve it. A theoretical overview may reveal the more obvious attributes of different data-structures and indicate how alternative implementations of the parameter database would perform, but only a test of the new implementation on the target system would yield concrete and reliable estimates.

The thesis will implement several different parameter database architectures based on different data structures, and demonstrate the memory and performance results to indicate the best future optimization path for the parameter database. It will be done by reviewing existing data structures, selecting, optimizing, implementing and testing the most suitable ones on the target unit.

1.1. Aim and research questions

This research seeks to:

- Analyze data-structures suitable for a DRAM implementation of one of Ericsson products parameter database containing thousands of parameters.
- Propose the best data-structure in terms of used memory area and speed.
- If possible, further optimize it for database size in memory and access speed.
- Create a prototype implementation.
- Test the performance of the new implementation.

Reduction of the memory overhead is the main requirement. Fast search speed and low database build time are important, but improving them is not essential. A sorted manner of storing data is desired but also not essential.

1.2. Problem

What data structure for the parameter database provides the lowest memory footprint and good speed performance?
1.3. Hardware environment and database characteristics

To build an optimal data structure for the parameter database, the system hardware environment and usage characteristics must be known. Since this thesis starts with a theoretical overview of data structures to select and explain the most suitable candidates for the database implementation, this paragraph provides the description of the system that uses the parameter database, thus enabling to argument the selection.

- The parameter database contains tens of thousands of parameters.
- It is mostly used during the initialization of the system, but some parameters may be read later during operation of the system due to reconfigurations triggered from the outside.
- The entire parameter database is currently stored in DRAM and is implemented using a hash map.
- The parameter database could benefit from reduction in memory footprint as well as search speed and initialization speed improvement, but the thesis gives preference to the memory optimization.
- Not all the parameters stored in the database are actually used, as they support different system revisions and configurations.
- During initialization nearly all parameters are read only once, as a result generating constant compulsory misses in the cache. Moreover, after initialization the system cache is used by multiple programs and the parameters are accessed so rarely (if ever at all) that they never remain in the cache for two consecutive reads.

Needles to say the current usage of the parameter database allows no real use of the system cache. Moreover, the greatest use of the parameter database is during initialization stage, where parameters are read once and in a random order (thus not in consecutive locations) and therefore no cache utilization is even possible (cache only provides speed improvement for consecutive reads, when it has already been loaded with the data). Manually reserving a part of cache for parameters is impractical due to the low use of parameter database, not to mention that Ericsson has refused the idea, as they want all the cache to be available for other programs. Thus whatever gains can be achieved from using the cache, they are unlikely to be significant for
this particular parameter database. Consequentially it is rational to consider data structures that can offer good speed without using the system cache, and avoid the ones that achieve their speed more due to their structure having better cache properties then the low number of operations in general and other good properties.

1.4. Method

1. The first part of the thesis – the overview of existing data structures – is a qualitative study of existing material and research on the world-wide-web about common data structures, their applications, properties and working principles, explaining if and how these data structures could be used to improve the parameter database and which aspects of it would be improved, in the context of the target system underlying hardware characteristics and desired results.

2. The second part of the thesis was the implementation and optimization of a parameter database for the target system. Such a design task meant applying the information and knowledge obtained during the first part of the thesis to implement the new data structures on the target system by creating new versions of the parameter database, and if possible – to further optimize the selected data-structures (or propose a new data-structure) for the DRAM parameter database, implementing them as prototypes and testing their performance.

2.1. The first part of the implementation process was the development of the benchmark code that would allow measurements of the parameter database speed and memory footprint.

2.2. The second part was the development and modification of the database objects in C++. A simulator of the target hardware was used for verifying the functionality of the databases based on new data structures. The different database implementations were stored as separate files, and used their individual namespaces to manage complexity of working with multiple implementations for the same database and to allow initializing several of these databases on the simulator simultaneously.

2.3. The third part was building of the created objects on the actual target system to obtain accurate speed and memory
footprint measurements. The objects created as separate files for the simulator could be used for this purpose with little additional modification of the code. However the changes to files which already existed as a part of the original database had to be moved to the files of the target system which had code differences compared to the simulator files.

Since the database is being optimized for a real system, software legacy dependencies also needed to be considered when implementing the optimized database.

**Performance measurement methods**

To obtain realistic results the tests were done directly on the target unit using a benchmark implemented by the author of this thesis as a part of the thesis.

The benchmark allows creating new instances of the parameter database, automatically generating a test-set of keys used in testing the search time, running the search time tests, measuring the parameter database initialization time and memory size.

The original database would be built and used mostly during initialization, but would always remain present in case of some reconfiguration triggered from the outside. Then the selected experimental implementation can be created using “create” command, and would contain the exact same data as the original database.

**The search time testing method**

The parameter database search time tests were done by reading the parameters from the database. In order to do that, a test-set of keys was necessary (parameter database stores key-data pairs and the parameters are found using a search function and a key). In order to make the search speed test results as representative of the expected actual performance as possible, the key test-set contents and the order of the keys should be as similar as possible to the expected actual usage pattern of the database.

**The expected usage pattern of the parameter database and the key test-set**

- The parameters in parameter database are expected to be searched in a random order and the majority of the parameters would only be searched and used once.
• During the run time, some parameters may be sparsely looked up in order to activate some additional components, which would be determined by the particular tasks the target system is doing at the time and by the user actions, or due to poor coding style. Nonetheless that is extremely rare. Moreover none of these accesses can be predicted to any meaningful extent, and they would differ for different system configurations.

• In initialization phase parameters will be accessed during startup of hardware components and their supervision. The order in which the parameters are read will be determined by the order of the components being initialized. This exact order is not known and will differ for different target system configurations.

Consequentially the expected usage pattern of the parameter database during both initialization and later operation can be best compared to continuous uniform random selection of keys, where used keys are removed from the test set (but not from the database) until all the keys are exhausted.

Such a behavior was approximated by iterating through a random-order test-set of keys containing all the keys that would actually be used (the parameter database is not fully filtered to have only the needed parameters).

Such a test-set was generated by:

1. Initializing the target system – initializing the parameter database, initializing the target system components by using the parameters from the parameter database and running the target system program. (By the end of initialization most parameters that are actually used have already been searched.)

2. Iterating through the parameter database and reading and saving to a vector the keys of all the parameters in the database that were flagged as “used” (flagging any accessed parameter with a “used” flag is automatically done by the target system program).

3. Randomizing the order in which these keys appear in the test-set vector.
Since the order of keys in the test-set is randomized – simply iterating through it for input keys used in the search function would result in an exhaustive search of all the “used” parameters in random locations of the database. Moreover every key would be used only once, which is the expected actual behavior for most of the keys.

The parameter insertion speed should be proportional to the database build speed, (except for the Presorted AssocVector, which parameter insertion speed will be proportional to the standard (not presorted) AssocVector build speed).

The memory size measurement method

The parameter database stored close to 15000 elements. The parameter database memory size was measured using a command (readily available on the target system) that displays the DRAM memory size of the entire program of the system, and a command in the benchmarks program that allowed obtaining the memory size of AssocVector implementation of the parameter database (due to implementation specifics obtaining the memory size of AssocVector was much easier then of other data structures). Then the memory size of AssocVector database implementation could be subtracted from the memory size of the entire program, giving the size of the rest of the program. With that known, the memory size of any other parameter database implementation could be found by subtracting the size of the remaining program from the entire DRAM memory used.

Method summary

- A theoretical overview of existing data structures will be done in an attempt to find alternatives for the existing Gnu hash map database implementation, which could reduce memory consumption while maintaining good speed.
- New parameter database implementations based on selected data structures will be created first on the simulator and then ported to the target unit.
- Speed and memory will be measured on the actual target system (not the simulator).
- For speed measurements a benchmark program (running on target system) was created using a real parameter database with real parameters and approximating expected usage pattern. Memory consumption was measured using commands readily available on the target system.
2. Background

The data-structures overviewed:

- Hash map [1] (*the existing implementation*)
- AssocVector [2] (*Loki::AssocVector*)
- A tree of AssocVectors
- Burst trie [3]
- Binary search tree [4]
- Treap [5]
- Skip List [6]

The thesis focuses on data structures that can be contained in the main memory and the memory overhead is considered the most important factor. The search speed is the second most important factor.

All analyzed data structures have O(Log N) or faster speed complexity for search operations, since given the large database size any data structure that couldn’t achieve at least O(Log N) speed complexity, would result in a large speed reduction as compared to the current hash map database implementation.

Also considered were data entry insertion and deletion speeds, the structure in which the data is stored (sorted, not sorted, in what manner sorted), ease of maintenance (resizing, insertion and deletion simplicity).

2.1. Gnu hash map data structure, used in the original database implementation

The currently used data structure is a Gnu implementation of a hash map (__gnu_cxx::hash_map) [1].

A hash map is one of the fastest data structures. It can search, add and delete entries with ideal and even average time complexity of O(1). The worst theoretically possible time complexity is however O(N), which can theoretically result due to hash collisions. Resizing can also negatively affect performance, as it requires rehashing all the keys in the hash map.

Hash map stores data units at locations corresponding to the hash derived from the key linking to that data unit. The hash is calculated using the hash function with the key as an input. If we assume that
calculating the hash function always takes the same time – then for as long as hash-collisions are avoided (the same hash value resulting from 2 different keys is calculated and used), the data search time remains constant and independent of the number of stored data entries N. This is especially useful in databases with large numbers of data entries. However the data search time can be negatively affected by hash collisions.

**Hash map look-up speed:**

The look up speed of a data entry mostly depends on the hashing function – as calculating the hash from the key is the only thing that needs to be done to pinpoint the exact location of the data. However there are some requirements for the hashing function that prevent us from making it the fastest one possible:

- The hashing function has to be optimized to generate as few hash collisions as possible, meaning the hashes must come out different if different key is used as the input.
- The bit size of the key has to be large enough to map (indicate) every pre-allocated bucket (memory location for storing one data instance).

Since some hash collisions are practically unavoidable – a collision resolution strategy has to be implemented. The time-efficiency of this collision resolution strategy may affect not only the time of adding the data instance that collides, but also the time it will later take to find that value, since some resolution strategies (like separate chaining [7]) result in colliding data stored not in the bucket with unique hash, but in a linked-list, or some other data structure that doesn’t exhibit the same look-up speed as a hash map, with possibly even as bad as O(N) time complexity. Gnu hash map uses internal quadratic probing to resolve hash-collisions [8]. Internal quadratic probing stores colliding keys in adjacent buckets. Full adjacent buckets are skipped in quadratic progression – next bucket address = previous address + (1+2i), where i is the index of a loop. This method is memory efficient as the colliding values are stored in the same array with all the other values without the use of pointers or creation of separate segments in memory that could cause heap fragmentation.
hash(key) = location (for example “9”)

Figure 1: Gnu hash map data structure illustration. Arrows illustrate quadratic probing when a key hashes to bucket 9, which is already taken (hash collision).

The collision rate (probability of hash collisions) usually depends on the hash map load factor (the proportion of the buckets that are filled to all the buckets). Thus speed can be traded for some memory overhead reduction.

**Cache utilization:**

The fact that hash map stores data in a seemingly randomly distributed locations in memory and consequentially jump around when accessing it results in a rather poor cache utilization.

**Data-structure overhead in memory:**

Gnu hash map stores all key-data pairs in an array of buckets, which is very concise and is a single segment in memory. It also uses internal quadratic probing to resolve hash-collisions, which is very memory efficient, as both the colliding and not colliding values are stored in the same array.

The only way the Gnu hash map wastes memory space is by keeping empty buckets in the bucket array to avoid too many hash-collisions and to avoid having to resize the array with each insertion (which requires rehashing all the values). The empty buckets take the same memory space as full ones but store no data.

Unfortunately load factor (used buckets ratio to all buckets) needs to be kept lower then 1 to have decent speed. The more loaded the hash map – the greater probability of a hash collision and the speed loss, which means that to have a good speed it will be necessary to allocate more buckets and memory space then needed just to store the data (although for good hashing functions the average look-up cost is not affected significantly by load factors < 0.7). Thus speed can be traded for some memory overhead reduction.
Conclusion:

Hash map performs best when the maximum database size is known in advance and the need for constant resizing can be avoided, thus it’s very well suited for parameter database as the parameter values are rarely changed. Moreover, since the parameter database is more then ten thousand elements large and the hash map approaches $O(1)$ search time complexity, it remains the fastest of the overviewed data structures despite resizing and collision problems.

Since an array, which is used to store key-data pairs in a hash map, is already a very compact data structure – the only way to further reduce the memory footprint would be to find a way to rid of the empty buckets that use memory space. To do that, a data structure as compact as an array would be needed, but it would also have to be possible and feasible to resize it for every insert operation.

A vector is somewhat similar data structure to an array, which allows dynamic resizing and is compact. The problem remains that in order to insert something into the middle of a vector – all the elements on the right side of the insertion location would have to be moved by one location to the right [9]. Similarly, in order to remove an element from the middle of a vector would require moving all the elements found right of the insertion location, by one location to the left. Needles to say, such an operation would be too time-consuming for any database with more then a few hundreds of entries, if all the data was stored in a single vector. Moreover, as the database size grows, the time overhead for insertion and removal operations would keep increasing with time complexity $O(N)$.

A possible solution to this hypothetical vector-based data structure would be to split this vector into multiple vectors. As long as the number of vectors is allowed to grow proportionally to the size of the database, the maximum size of an individual vector would remain constant, as would the average number of data entries to be moved during insertion / deletion operations. Then the vectors containing the key-value pairs could perhaps be kept small enough to allow acceptable insertion / deletion times for databases where these operations are not very common, but large enough to allow sufficient memory saving despite additional header metadata (multiple instances due to multiple vectors instead of a single array), some memory fragmentation and the costs of the data structure used to connect these vectors into a single coherent database.
As an alternative, some other compact data structures (that are efficient with a small number of data entries) could be used instead of vectors as large leaves for a trie or a map.

There are multiple data structures that already exploit vectors to save memory space, like Sparsehash (a memory efficient hash map originally created by Google, but later relinquished to the community) [10] and AssocVector [2], while data structures like Judy Hash [11] explore dynamically dividing the entire database data set between various data structures which are compact and efficient with small data sets.

### 2.2. AssocVector

(Developed by Andrei Alexandrescu.)

An AssocVector (as defined in Loki::AssocVector [2], [12]) stores all data entries as a vector, but achieves search time complexity of $O(\log N)$ by storing the data entries sorted by a search parameter (the size of a number if the key is numerical value, or alphabetical order for character keys). A major drawback of AssocVector is that in order to place new data entries in a vector in a sorted manner, after the insertion location is found ($O(\log N)$ complexity) if that location is not at the end of the vector – all data entries that are after that location, must be shifted by one location to the right ($O(N)$ complexity) to make space for insertion. Only then can the new value be inserted. Similar problem is encountered when deleting data entries, since all data entries to the right of the deleted one need to be moved left by one position, making insertion and deletion operations very slow for large datasets.

The main advantage of AssocVector is its size in memory, since all data entries share one header generating almost no metadata overhead and the vector is a single segment in memory resulting in very low heap fragmentation.

**Data-structure look-up speed:**

In AssocVector data entries are searched by comparing the alphabetical order of the key in the middle, then middle of one halve, so-on ... (like BST), until either an exactly matching key is found, or the keys immediately next to last searched location (on both sides) have already been checked and the wanted key wasn’t found, in which case the wanted data is not stored in the AssocVector.
The search operation is in principle analogous to a binary search tree, and thus is expected to take similar time.

**Cache utilization:**
The fact that AssocVector stores data compactly as a vector, sorted by key and in sequential memory locations, makes it efficient in terms of cache utilization.

**Data-structure overhead in memory:**
The memory overhead of AssocVector is very small, compared to other data structures. Key-value pairs are stored as a compact vector, with all the data entries sharing a single header. Also the vector that stores key-value pairs is kept as a single segment in the memory, resulting in practically no heap memory fragmentation.

**Conclusion:**
The AssocVector could be well used for databases that don’t change over time, in other words, are used only to look-up some values, but never written to or cleaned of any data after the initial build, or for databases where writes are very few, or writing and deletion speed are less important than the size overhead in memory. This is in fact the case with the parameter database where the parameters practically never change, therefore making the AssocVector a possible solution to the memory problem. AssocVector theoretically has the smallest memory overhead of all the data structures overviewed in this thesis. Moreover it stores data sorted by keys.

However the main drawback of this data structure is very long \(O(N)\) complexity database build time which could severely affect the initialization time of the target system, which has to wait for the parameter database to be built before it can start using the parameters.

One way to address this problem would be to instead use several AssocVectors of smaller size, connected via some other data structure (for example a map, or a BST). If for example instead of using a single AssocVector for the entire database, we used a 100 100-times smaller AssocVectors, it would result in the additional memory overhead of just 99 additional headers and footers for the other AssocVectors, and a 100-node map (a negligible memory overhead considering the size of the database). However the average number of data entries to be shifted for each insertion that is not at the end of the vector, would be
reduced 100 times. The time complexity for insertion would be O(N/100) for a large database.

Naturally, a good balance between the number of AssocVectors and their size could be found either experimentally, or through a theoretical investigation. The keys for the data structure that holds all the AssocVectors could be the first few characters from the key that identifies the individual data entry, and the remainder of that key could be used to navigate inside the AssocVector. Such a data structure is in principle very similar to a burst trie – a data structure analyzed next.

An alternative way of addressing the long build time problem for the AssocVector, would be to sort the data by the size of the key, before entering it to the AssocVector, so it would always be added at the end of the Vector during the actual AssocVector build. This solution however would not affect the parameter insertion speed during the later operation, after the initial AssocVector build. However there are practically no parameter insertions after the initialization of the parameter database addressed in this thesis, thus this effect should be unimportant.

2.3. Burst trie

A burst trie uses containers that are linked via a standard trie with these containers as its leaves (henceforth called “access trie”), to conserve memory space without sacrificing speed. The containers are small data structures that (before bursting) contain the stored data entries and pointers that maintain this data structure. Additionally each container has a header with information that is used to determine when to burst the container.

The nodes of the access trie contain an array of characters which are used to navigate through the access trie, pointers that can point to a container or to another trie node, and one empty-string pointer to a single data entry. The burst trie implements a sorted database.

Initially the burst trie consists of only one container. Upon fulfilling certain conditions (like reaching certain number of entries in the container, or a number of accesses, or a low proportion of immediate hits in the container) which indicate that continuing to store data in one container is no longer efficient, the container “bursts” – is replaced by a trie node, connecting new containers, which partition the data and keys form the original container among them selves. This allows using more compact data structures that are efficient with smaller datasets (such as a binary search tree, or a splay tree, or a
linked list, or AssocVector) to store the data end keys more compactly and also make it faster then a conventional tree.

Like a normal trie, the burst trie supports a pattern matching queries which take time proportional to the number of characters in the pattern. However, according to an article on burst tries [3] burst trie uses only 1/6th of the space used by a compact trie and is faster. According to the same article, they are also more compact then binary trees and splay trees, and are more then twice faster when used to implement a dictionary. A conventional trie was not individually analyzed in this thesis because of its impractically large size for it to be stored entirely in the main memory.

Search operation:
Search starts by identifying the container, which is done by using the first few characters of a key string to traverse the access trie which links to all the containers. Then the remainder of the key is used to find the record inside that container. If the container returns NULL – the wanted data is not stored in the burst trie. However not every access trie branch ends with a container.

The access trie is traversed by comparing the i'th character in the N characters long key string for selecting an access trie node at depth i.

The empty-string pointer is stored at the end of the key string of length N so that when iterating through key string characters during the search in the access trie: if the end of the string is reached before encountering a container, and the index of the string is further incremented to N+1 – we would access the empty-string pointer. The empty-string pointer can point to a single data entry, or to be NULL. If during the access trie traversal an empty-string pointer with NULL value was reached, than the wanted data is not stored in the burst trie. Else an empty-string pointer points to the wanted data entry.

Insert and delete operation:
The access trie part of the search algorithm is used to identify the container which should hold the new data entry. If a trie node of depth N+1 is reached before any container – the new data entry is added under the empty-string pointer. Else the container data structure method of insertion is used to insert the new data entry into the container.
Deletion happens in a similar way: locating the container, locating the data in the container and using the container data structure deletion method to delete the data entry. Only if the deletion results in an empty container and the containers parent trie node has no other children – that node should be removed from the trie. The process can be applied recursively up to the root.

**Data-structure look-up speed:**

Search operation usually consists of two operations: finding the location of the container that stores the wanted data, and then using the search method typical to the container data structure to locate the data inside the container. This way the look-up speed takes the advantage of limited size of containers, which are implemented as data structures that are most efficient for smaller number of data entries, and the speed of a standard trie for locating the containers. This double implementation allows optimizing the burst trie to use the fast properties of a trie for often accessed data, while containing the less used data in larger containers for memory overhead reduction. This balance is determined by the heuristics of bursting.

To know when to let a container to burst, it’s necessary to understand how this data structure achieves its advantages over normal trie. In the optimal implementation, the most common strings should be found via the fast access trie, without needing to search excessively inside the containers. Since containers are more space efficient than the trie nodes, they should only burst if accessed often and have high average search time, thus sacrificing area for speed where it’s most efficient. Therefore the heuristics of when to burst the container could be handled by a function with parameters that monitor the number of accesses to the container (so it doesn’t burst if not accessed often enough and save space instead), the average time of search or number of node transactions inside the container (so that slow performing containers would burst first) and the container size, as proposed by the article [3]. The same article also suggests that a well balanced burst trie is overall significantly faster than the binary trees, splay trees or even standard tries, when used to implement a vocabulary accumulation with uneven probability distribution, since shorter words tend to be more often used and also closer to the root. However it also suggests a lower efficiency for data with even probability distribution (characteristic to the parameter database), or equal lengths of strings.
Cache utilization:
The use of pointers in dynamic string data structures are the fundamental cause of cache-inefficiency, as they can lead to random memory accesses. If the containers are implemented as data structures with good cache properties (storing data in sequential memory locations) the cache efficiency could be improved, but every search starts at the top of the trie connecting containers, and the trie data structure uses pointers and is inefficient in terms of cache, thus compromising the burst trie. [13]

Data-structure overhead in memory:
The memory overhead of the burst trie depends on what data structure was used in the containers, what heuristics for bursting were used, how many branches on average the access trie nodes have, and the nature of data stored. It is possible to trade speed for some memory overhead reduction by allowing greater size of containers before they burst, or choosing container architecture that trades speed for lower memory footprint.

The two main memory consumers are the access trie and the containers. The overhead of an access trie is comparable to that of a trie, but each node additionally has an empty-string pointer. The number of nodes in the access tree decreases logarithmically with each higher layer. The containers are essentially small data structures (for example binary search trees, or linked lists) that search the wanted data using the remainder of the key characters which weren’t used by the access trie to find the container. Thus the memory consumed by the container depends on its internal implementation.

Conclusion:
I have already mentioned that implementing the parameter database as a group of AssocVectors linked by some other data structure is in many ways similar to the burst trie that could use AssocVectors as containers. It is worth to look at the differences of the two implementations and see how different alternatives they offer:

The burst trie uses a trie for the access of the containers. It starts by putting all the data entries in a container and decides when to burst it according to some heuristics function, like the size of container or average search time. The number of containers is decided by the performance of the data structure and is thus in a way adaptive and self
adjusting. The user doesn’t need to think about how to parse the data among the containers, apart from choosing a smart heuristics algorithm. However he also doesn’t control it. Moreover, selecting the right parameters for automatic bursting and accurately predicting the outcome may be very hard, not to mention complicated code implementation, but if implemented successfully, the burst trie could be more adaptive to different datasets then the simple implementation described below.

On the other hand, a simple data structure (a map or a binary search tree) could connect containers (for example AssocVectors) that are navigated through using the remainder of the key, while the access map /BST would be used to navigate to the correct container using the first part of the key. The container size would be limited by the number of unique keys that can be represented given the size of the remainder of the key used to navigate the container. It would be much easier to implement and therefore more reliable. It could be sorted as long as the container architecture implements sorting.

In the end both of these implementations will have some pointers which Gnu hash map avoids, but the pointers would connect containers each holding multiple key-data pairs. Therefore as long as these containers are kept large enough, the pointers will be few and may take less memory than the empty buckets in a hash map, resulting in lower total memory consumption.

2.4. Binary search tree (BST)

A typical binary search tree implementation [4] is a linked data structure made of nodes, which consist of the key, the pointer to the node of the left sub-tree, the pointer to the node of the right sub-tree, pointer to the parent node and the data record (the actual data stored). The left leaf node key is always smaller or equal to its parent node key, and the right leaf node key is always larger or equal.

If a pointer to either left or right sub-tree node contains NULL value – it indicates empty sub-tree. Since in this structure there are many pointers referencing NULL – it’s not space efficient.

**Data-structure look-up speed:**
The basic operations in a binary search tree require comparisons between nodes and take time proportional to the height of that tree, since the tree must be traversed vertically from the root node to the leaf node with the data. Thus in a balanced binary search tree the worst
case search time complexity is $O(\log_2 N)$, where $N$ is the number of entries, but if the tree is not balanced, the worst case time complexity can be as bad as $O(N)$.

**Cache utilization:**

If the cache was available, it would be logical to store the top levels in it, since the search always starts from the top. However the binary search tree uses many pointers and doesn’t store data in sequential memory location. The result is rather poor cache utilization.

**Data-structure overhead in memory:**

The object header metadata overhead and heap fragmentation are large, since every node is a separate segment in memory, has its own header (unlike in array, where the header for all array elements is stored only once and applies to the entire array). Additionally, the key data as well as 3 pointers to other nodes need to be stored per every node. This large amount of structural data, as well as object metadata and the NULL pointers take a lot of additional memory and result in low binary search tree efficiency in terms of memory overhead.

**Conclusion:**

The binary search tree offers a way to store alphabetically sorted data, which hash table lacks. Moreover, it doesn’t require re-indexing all the database items when resizing, or reserving additional memory space unlike the hash table, although considering the application of the parameter database, which after the initial build will hardly ever need resizing, this advantage is not so important.

However the binary search tree is dependant on a large number of pointers. Moreover the data objects that pointers connect also have their individual headers, which further complicate memory overhead reduction. The header metadata overhead could be mitigated by implementing the BST as a vector that holds the key-value pairs and the indexes of child data entries instead of pointers. The advantage of such implementation, as compared to the array hash table, would be the ability to add all new data entries at the end of the vector, thus there would be no need to move any other items. However this implementation would still need to store the mentioned indexes, which might induce a memory overhead similar in size to that of the empty reserved space in the array hash table.
Finally, the binary search tree has a search time complexity O(Log N), which prevents it from beating the hash table for speed, when the number of data entries is large.

### 2.5. Treap

A treap data structure is very similar to the binary search tree: they both allow binary searches among dynamic sets of ordered keys. Thus the treap could be viewed as a binary search tree with nodes ordered by a key and a randomly chosen numerical priority in such a way that the priority number of any leaf node must always be smaller then priority of its parent node (thus the root node has the highest priority). Then the keys are arranged in a way that the left leaf-node keys are always smaller then the parent node key, and the right leaf node key is always larger.

A treap can also be constructed with non-random priorities and a higher priority can be given to more important data entries, which will then be closer to the root and thus will be accessed faster. However non-random priority assignment also results in non-balanced treap that has a lower average-case performance.

**Data-structure look-up speed:**

Although the search operation has exactly the same steps as in a BST, according to two different studies \[5\] and \[14\], treap has a better search, write and deletion speed then the red-black tree and AA tree (which are balanced binary search tree implementations), skip-list, radix tree or a Heap, and for large datasets, even a Shell, when revolving these operations in descending or ascending order. Still the time complexity of all the mentioned data structures is O(Log N).

However the number of key comparison and rotation operations during a search of keys in a nearly sorted order, was shown by the study \[14\] to be nearly 50% higher for the treap then for the red-black tree and AA tree. For the insertion and deletion operations the treap still needed only half the number of comparison and rotation operations as compared to the red-black tree and AA tree. Since effective cache utilization is unlikely for the parameter database, as explained before, the number of operations and rotations are some of the most important factors in determining the speed performance that the data structure is likely to have.
Cache utilization:

A is affected by the same factors as that of a BST – a high pointer-to-data ratio, which compromises the cache-efficiency, although there are treap implementations (for example T-treap) designed to reduce this ratio ([15] and [16]).

Data-structure overhead in memory:

The memory overhead of the treap is composed of two parts:

1) The index of the treap is composed of \( \log_2(N) \) number of index entries, each holding a key and priority number.

2) The pointers to leaf elements of the tree constitute an additional overhead.

Conclusion:

Since the parameters in parameter database will be accessed in a random order during both the initialization and the later operation, the treap efficiency when searching data in ascending or descending order, would not be exploited. Thus the treap should be clearly outperformed by the hash map, when used with large enough datasets for the hash map to take the advantage of its \( O(1) \) time complexity.

The sources of memory overhead in a treap are similar to those in a binary search tree (only treap also stores node priority numbers), and should offer similar optimization possibilities. However, given the large memory overhead this data structure is unlikely to improve upon the existing parameter database implementation.

2.6. Skip list

A skip list [6], [17] is one of the simplest data structures that offers look-up, insertion and deletion time complexity of \( O(\log N) \).

A Skip list uses a hierarchy of linked lists to store sorted list of items. It is built of layers, where each layer is a linked list and elements from lower layer \( i \) appear in the higher layer \( i+1 \) with the probability \( P \). Typical implementations use \( P = \frac{1}{2} \), or \( P = \frac{1}{4} \). The lowest layer contains all the items of the skip list. Then the index is \( \log_P(N) \) levels in height (where \( N \) is the number of entries).
Searching for an item:
The items in the skip list are sorted by the size of the keys. The search for an item begins with checking the size of the first key in the top index layer: if it’s smaller then the key of the wanted item and there are larger keys in that layer – check the next key. Else, if the key is larger – return to the previous key position and move down by one layer. If the key is smaller but there are no more keys in that layer – go down one layer and continue the search in the same manner. Upon reaching the bottom layer – keep checking the keys in increasing order by size until you either find the wanted key or reach a key that’s larger then the wanted key, meaning the wanted key is not stored in the skip list.

Insertion of an item:
The insertion of a new item in a skip list is done by: finding the position (after the closest smaller key, O(Log N) complexity), then inserting the key and object as an additional item of the linked list, and finally building the tower: duplicating the key in higher level with the probability of duplication = P. The last operation (building the tower) involves random number generation and also has the complexity of O(Log N).

Deletion of an item:
The deletion of an item in a skip list is done by: finding the element (O(Log N) complexity) and deleting the tower (also O(Log N) complexity (height of tower)).
Data-structure overhead in memory:
The memory overhead of skip list is composed of two parts:

1) The index of the skip list is composed of \( \log_P(N) \) number of index entries (keys).

2) Since the levels themselves internally are linked lists – the pointers to elements of the linked list and the object headers of these elements constitute an additional overhead.

As the probability \( P \) (probability of a higher level to contain an item from lower level) decreases – the number of levels also decreases, reducing the memory overhead. However the number of searches in each level (i.e. key comparisons), needed to find the wanted key, increases and is on average = 0.5/P and never less then 1 (thus the former equation applies for any \( P \leq \frac{1}{2} \), and \( P > \frac{1}{2} \) is never used in practice).

For example with \( P = \frac{1}{4} \) there would be on average two compare operations per each level, but the number of levels would approach \( \log_4(N) \) and the resulting number of indexes would be approximately equal to \( N/3 \).

Data-structure look-up speed:
Since the number of levels, which is equal to \( \log_P(N) \), drops when decreasing the probability \( P \), but the number of searches-per-level increases – the resulting average total number of searches \( S \) can be calculated as

\[
S = \log_P(N) \times \left( \frac{P}{2} \right);
\]

\( \log_P(N) \) is the number of levels that would have keys. On average we need to compare \( P/2 \) keys per level (the average number of keys in a level before reaching a key that is duplicated in a higher level is \( P \), with equal probability to terminate at any one key.

For example if we try to calculate the average total number of key comparisons for \( N = 262114 \): with \( P = 1/2 \) we get \( S = 18 \); with \( P = 1/4 \): \( S = 18 \); with \( P = 1/8 \): \( S = 24 \); with \( P = 1/16 \): \( S = 36 \), with \( P = 1/32 \): \( S = 57.6 \) and so on. It is therefore visible that while there should be no notable speed difference between using \( P=\frac{1}{2} \) and \( P=\frac{1}{4} \), with even smaller \( P \) the number of key comparisons starts to grow and the speed is being sacrificed to gain reduction in memory overhead.
Cache utilization:
Skip lists are not optimal in terms of locality reference, since related elements are usually far apart. As a result their cache efficiency is low.

Conclusion:
As compared to the current Gnu hash map database implementation, the skip list has the following disadvantages:

1. It should be notably slower if used to store tens of thousands of key-value pairs, due to its time complexity being similar to a binary search tree, when using $P \geq \frac{1}{4}$, or even worse when using $P < \frac{1}{4}$.
2. The elements in a skip list can participate in more than one linked list and thus can have more than one pointer.
3. Skip list elements have their own headers and are separate segments in memory resulting in heap fragmentation.
4. The skip-list has a number of index layers, each containing copies of some keys from the lowest layer. The hash map doesn’t incur this additional overhead.

The advantages of a skip-list are:

1. The skip list doesn’t need to reserve some memory space in order to keep the load factor $< 1$, which is and additional overhead incurred by a hash map.
2. It stores data in a sorted manner.
3. It is simple to implement.

To conclude, the skip list doesn’t offer any reduction of the size in memory, which is the most important goal, and is expected to perform slower than the hash map. The sorted manner of storing data alone is not enough to justify an upgrade from the existing database implementation.

2.7. Background findings summary
The following data-structures have been overviewed in the background section:

- Hash map [1] (the existing implementation)
- AssocVector [2] (Loki::AssocVector)
- A tree of AssocVectors
- Burst trie [3]
The original Gnu hash map implementation is highly memory efficient due to avoiding pointers by using hashing to locate parameters in the container array and by using internal quadratic probing to resolve hash collisions. However some memory is lost due to keeping empty buckets in the array, which take the same memory space as the full ones.

The Binary search tree (BST), the Treap and the Skip List data structures were rejected as potential solutions since they are unlikely to take less memory space than the original hash map (the main requirement) mainly due to large pointer-to-data ratio (they rely heavily on pointers to implement their functionality) and in the cases of BST and Treap – also due to creating multiple nodes as separate segments in memory which can cause memory loss due to large heap memory fragmentation.

The AssocVector promises best memory savings due to its very compact nature – it stores key-value pairs in a single vector and doesn’t need to keep empty buckets unlike the original database implementation using a hash map. Unfortunately, if unmodified, this data structure would fail the build speed requirement due to its extraordinarily large insertion time – O(1) insertion time complexity. Using a tree of AssocVectors was proposed as keeping multiple vectors would improve the insertion speed and the build time, however a simpler and faster solution (using key presorting) was selected instead, as described and analyzed in section 3.1 – the Presorted AssocVector.

Finally a burst trie based data structure has the potential to reduce the memory consumption (uses large compressed containers each containing multiple parameters) and have good search time and speed performance (Trie-like or hashing based searches, O(log N) search and insertion time complexity). Judy Hash data structure (as described and analyzed in section 3.3), uses JudyL (burst-trie-like data structure) as a virtual array and hashing for searches. It was selected to implement the parameter database and to be tested along with Sparsehash and Presorted AssocVector.
3. Tested data structure implementations

The data structures tested were:

- Presorted AssocVector
- Sparsehash (Google:sparse_hash_map)
- Judy Hash (Judy_map_l [11])
- The original hash map (__gnu_cxx::hash_map) – for comparison purposes

3.1. Presorted AssocVector

This data structure is a modified version of the Loki::AssocVector developed by Andrei Alexandrescu [2]. The AssocVector stores all data (key-value pairs) in a vector sorted by the key value size. It is very concise but has very slow insertion speed, as the vector is sorted and inserting new values in the vector involves moving all values located on the right side of the insert location.

Keys are found by checking the size of the middle key, then if it’s smaller then the target – in the middle of the lower half, then in the middle of a quarter, and so continuing until the key is found or, adjacent keys on both sides have been checked, in which case the wanted key isn’t in the database. Search method and time is somewhat similar to BST search [18].

The only difference between AssocVector and Presorted AssocVector is how it is initially created. The modified version allows initially reserving any amount of memory space for the AssocVector and direct insertion of new elements using push_back() function, which allow bypassing the standard AssocVector process for inserting key-data pairs (no more sorting or resizing during insertions, instead can just append new entries at the end of the vector as long as the right amount of memory is reserved).

Since with the push_back() function new entries are always appended to the end of the vector, no moving of other values is necessary allowing fast database creation. However to maintain the AssocVector functionality the keys must be sorted by size in the vector. Therefore to append them using push_back() all the key-value pairs must be presorted before entering them to the vector.

Solution – use std::map for initial sorting:
1. Std::map uses pointers to connect its data entries, thus no shifting of data values is needed and sorting is easy and fast.
2. In std::map data is sorted in the same manner as it needs to be sorted for AssocVector – by the key size.
3. Resulting sorted map elements can be directly copied into AssocVector by simply assigning its contents to the vector at negligible time cost (around 50ms for copying the entire database).
4. Resulting build time of AssocVector ≈ (std::map build time (sorting expense)) * 1.25.

This data structure is well suited for a parameter database as new parameters are practically never added during the runtime, and initially known parameters can be sorted in advance to reduce the initial build time of the database. This data structure keys are also sorted in the alphabetical order, which makes it more readable and easier to navigate. Moreover, presorting is faster than instead having a map of multiple AssocVectors since even in smaller vectors some shifting still has to be done to allow insertions in the middle of a vector. Also, another implementation based on multiple vectors principle – the Sparsehash [see section 3.2] has already been selected to be implemented.

### 3.2. Sparsehash

Sparse hash map [10] stores key-value pairs in a number of groups, each holding up to 48 elements. Each group is made of a vector that
holds the key-value pairs, and a bitmap mapping 48 bits that indicate which of the available 48 positions are taken in that particular group.

Since the bitmap can map all 48 positions regardless of the size of the vector, there is no need to rehash all the values as the vector grows, thus leaving no empty buckets that waste space. Instead, when adding a new value, its location \( i \) in the “virtual array” embodied by the bitmap is coded as 1 by the corresponding bit in the bitmap. In reality the value is added to the vector.

The value location in the vector can be found by counting the number of ones \( \leq i \) in the bitmap, as it indicates the number of used buckets before the value, thus giving its location in the vector. If bitmap location denoted by \( i \) is =0, the value is not in the database. Since vector grows every time a new value is inserted, the empty buckets are virtual and don’t take any space.

The hash collisions are handled using quadratic internal probing \([8]\) which allows avoiding using unnecessary pointers, but also mitigates clumping effect that linear probing would cause.

\[
3 \times "1" \Rightarrow V[2] \quad \text{hash(key)} = 56
\]

Figure 3: Sparsehash data structure illustration.

### 3.3. Judy Hash

(Developed by Alexey Cheusov.)

Judy Hash (Judy_map_l \([11]\)) data structure is an attempt to implement a dynamically resizable hash map (allocates / frees memory with each data insertion / removal).

Instead of a simple linear array Judy Hash implementation uses JudyL \([19]\) set of functions from Judy library, created by Doug Baskins and Hewlett Packard. JudyL implements a compact compressed map (treated as an array at the API level) and enables Judy Hash dynamic resizing \([11]\).

The ability to dynamically resize the data structure with each modification allows avoiding empty buckets which saves memory
space, but it also has to use some pointers for implementing a trie. Since the main goal of the thesis is to reduce database memory consumption, it is vital that the trie uses very efficient compression method.

JudyL conserves memory in the following ways:

- JudyL has very few levels and very large branch nodes that contain up to 256 values / pointers. Node branches can be seen as sub-tries. The nodes are compact data structures that can take form of literal arrays, or virtual arrays implemented as bitmap or linear nodes. The Judy algorithm decides which type to use depending on array number of elements.
- The key is divided into 1 byte portions which are used to navigate through the trie nodes. Only the remainder of the key need exist in the sub-tries, thus avoiding key duplication.
- The leaf nodes contain the values stored in JudyL, while the branch nodes contain pointers to other nodes. Each leaf node is pointed to by a single pointer stored in one of the non-leaf nodes (unless the dataset is so small that the entire database is a single node), thus low pointer-to-data ratio.

Judy Hash Judy_map_l version resolves collisions by storing colliding elements in an external linked list.

![JudyL data structure illustration](image)
4. Data structure test results

The implemented data structures and the currently existing parameter database implementation were tested for size in memory, average and worst search time, search time dispersion and database initialization time.

The tests were done on the target unit using a benchmark implemented by the author of this thesis as a part of the thesis.

4.1. Memory size comparison

The DRAM memory used by the tested data structures:

- The “requested memory” refers to the memory needed to store all the elements of the parameter database (the database structure implementation + contents) and does not include the memory lost due to heap fragmentation.
- The “used memory” refers to the total parameter database memory consumption (including the heap fragmentation attributable to the parameter database objects).

Plot 1: data structure requested and used memory size comparison.
The original Gnu hash map parameter database implementation is the first data structure shown in the result plot for comparison purposes.

Two of the newly implemented data structures – the Presorted AssocVector and the Sparsehash succeeded in reducing the real memory consumption (including the heap fragmentation), while Judy Hash has failed to do that. However even the Judy Hash has a lower “requested memory” size then the Original Hash Map and fails in real memory consumption only due to exceptionally large heap memory fragmentation.

4.2. Parameter search time comparison

The parameter search time testing:

- Average search time was calculated from 444000 read operations (by iterating over the test-set of 444 unique keys 1000 times).
- The worst search time is the largest search time of all the 444 keys of the test-set.
- Search time dispersion characteristics were derived by iterating over 444 key test-set once while measuring the search time for every parameter discretely.

<table>
<thead>
<tr>
<th>Parameter search time:</th>
<th>Original Hash Map</th>
<th>AssocVector</th>
<th>Judy Hash</th>
<th>Sparsehash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min (µs)</td>
<td>56</td>
<td>77</td>
<td>64</td>
<td>62</td>
</tr>
<tr>
<td>Max (µs)</td>
<td>1300</td>
<td>2000</td>
<td>2000</td>
<td>2000</td>
</tr>
<tr>
<td>Average (µs)</td>
<td>94</td>
<td>120</td>
<td>110</td>
<td>110</td>
</tr>
<tr>
<td>25th percentile (µs)</td>
<td>70</td>
<td>92</td>
<td>84</td>
<td>79</td>
</tr>
<tr>
<td>Median (µs)</td>
<td>79</td>
<td>100</td>
<td>96</td>
<td>88</td>
</tr>
<tr>
<td>75th percentile (µs)</td>
<td>90</td>
<td>110</td>
<td>110</td>
<td>110</td>
</tr>
<tr>
<td>Standard Deviation (µs)</td>
<td>100</td>
<td>130</td>
<td>140</td>
<td>140</td>
</tr>
<tr>
<td>Initialization time (sec.)</td>
<td>0.94</td>
<td>1.3</td>
<td>1.1</td>
<td>3.4</td>
</tr>
</tbody>
</table>

Table 1: database search time characteristics and initialization time.
Min, Max, 25th percentile (Q1), 75th percentile (Q3) and median search time in microseconds (the smaller the better)

Plot 2: database search time characteristics.

The original hash map implementation outperformed to some degree all the new implementations in terms of average search time and even more so in terms of worst search time, which comes as no big surprise considering that the main goal of the thesis was to reduce parameter database memory size, and the original implementation – the Gnu hash map is a fast hashing-based and well developed, mature data structure.

The initialization speed was distinctly large for the Sparsehash database implementation. Presorted AssocVector and Judy Hash also underperformed the Original Hash Map by much smaller margin.
4.3. The search time dispersion comparison

Plot 3: the parameter search time dispersion of the original hash map database implementation.

Plot 4: the parameter search time dispersion of the presorted AssocVector database implementation.
Plot 5: the parameter search time dispersion of the Judy Hash database implementation.

Plot 6: the parameter search time dispersion of the Sparsehash database implementation.
The search time dispersion displays similar characteristics for all the tested data structures. Since the speed tests were done by using the same fixed set of keys with all data structure implementations, the same long keys that result in large search time values in one data structure implementation would also affect all the remaining implementations in a similar manner. Consequentially the dispersion plots tend to have similar tendencies (for example the spike around 1000 µs).

The standard deviation is quite similar in different database implementations and is larger where the average search time is larger, which may be due to the larger interval over which the results span over the X-axes (search delay in µs).

## 5. The conclusion

Sparsehash and Presorted AssocVector database implementations successfully reduced the parameter database memory consumption respectively down to 57% and 58.2% of the original memory consumption. Judy Hash on the other hand has failed to reduce memory consumption and in fact takes more memory space than the original implementation. The reasons for this result are analyzed in the conclusion section “Judy Hash” [5.1]. (Note: the memory consumption referenced here is the memory actually used and includes the heap fragmentation, as opposed to the memory requested).

All tested data structures underperformed to some degree the original hash map implementation in terms of search speed. That is understandable considering that the main goal of the thesis was to reduce parameter database memory size, and Gnu hash map being a well developed, mature and acknowledged hashing-based data structure, no compressed and functionally equivalent implementation could truly beat it in speed due to additional complexity originating from resizing with each insertion alone, if nothing else.

Fortunately, the parameter database is only used intensively when booting the target system, when the parameters are read only once. Any use after the initialization is almost non-existent and thus the parameter database speed does not affect the target system speed much.
5.1. Judy Hash

The test results from Judy Hash fail to address the main problem of reducing memory footprint. In fact this data structure ended up consuming even more memory space then the original hash map implementation. The large discrepancy between memory requested and memory used may shine some light on what went wrong:

Judy Hash database implementation has requested far less memory then the original hash map implementation, but in reality used more memory space due to large heap fragmentation. At this point it is worth remembering that Judy Array – a component data structure of the Judy hash used to actually store the key-data-pairs – was the only data structure used in any tested implementation, which could dynamically decide to change its configuration and node architecture. The node containers are resized with each key-value-pair insertion to the respective container. Moreover, the algorithms responsible for maintaining the branch-containers optimized, allow using different data structures for different containers, depending on the frequency of accessing that container and its contents, and even allow changing the data structure of an already existing container [19]. All this dynamic activity in memory added to using linked lists for resolving hash collisions can cause heap fragmentation (create a large number of gaps in the memory too small to reallocate). Finally, should the trie implementation, which uses segments of the key, result in a larger number of smaller branch containers due to dataset that sparsely uses characters of its very long keys (up to 128-character-long strings as keys for a database with under 20000 elements) – the header/ footer metadata would become far more dominant then in large array or vector implementations of other tested data structures. Thus this attempt of using dynamic resizing ended up paying for its architecture flexibility with overly large heap memory fragmentation.

5.2. Presorted AssocVector

The presorted AssocVector has to pay for presorting and copying of data with a bit larger initialization time then that of the original hash map database implementation, but as a result of being a single vector – it’s one of the most concise database implementations tested, and experiences very small heap fragmentation. Its low memory consumption has only been beaten by the Sparsehash, and only by 2%. The search speed is reduced by 23% as compared to the original database implementation, largely due to tree-like search as opposed to
using a hash function in a hash map. The worst observed delay (from reading 444 different parameters) was 56% greater than that of original database, which is not bad when moving from a hash function to a tree-like search.

Since this data structure sorts its keys in the alphabetical order, it makes it more readable and easier to navigate.

The only potential obstacle for using this data structure is that the initial data sorting has to be done in a pointer-based data structure before being copied to the AssocVector in order to avoid long sort-time. As a result, there is a moment during initialization, when 2 copies of the database must fit in the memory. It should however be affordable since during such early stage of initialization most of the programs that require DRAM use aren’t running yet, and the additional memory used can be reclaimed immediately after the copying process.

In the end it is this data structure that I would recommend as the best suited for the parameter database implementation, since it results in high memory savings (the primary goal) and doesn’t have any large failings like the high initialization time for the sparse hash, considering that the parameters are read very few times and therefore the search speed does not have as big an impact on the total time spent on the database, as the parameter database initialization time.

5.3. Sparsehash

Sparsehash had the lowest memory consumption of all the tested database implementations and slightly outperformed Presorted AssocVector in terms of search speed. Its only, yet serious drawback was its exceptionally large initialization time. The reason for it being so large (3.4 seconds, as compared to only 0.94 seconds for the original implementation) is because Sparsehash was the only tested database implementation that resized its vectors with each insertion but wasn’t presorted like the AssocVector. The fact that similar initialization time delay was not observed for Judy Hash (which also often resizes vectors, although not with every insertion) also confirms that Sparsehash array containers were large compared to Judy. The initialization time is pretty important for systems that need to be rebooted frequently and to have a low downtime, thus Ericsson may decide such a large initialization time to be unacceptable.
6. Future work

Future work propositions for database improvements offer alternative ways to address the main thesis problem, but do not involve changing the parameter database underlying data structure (Gnu hash map).

6.1. Proposition 1: partial key compression

The key compression method affects the contents of the parameter database rather than its data structure implementation, therefore it could be used with the new parameter database implementations.

Background specific to proposition 1

- The parameter database stores key-data pairs. The keys are strings of up to 128 characters and often take more memory space than the values they refer to.
- The parameters are retrieved from the database using two different functions “Get” and “Autoget”, depending on the application.
- The search operations with these keys involve hash calculations and in the case of more than 1 key hashing to the same bucket – even string comparisons that can be slow with long keys.
- The paradigm currently employed when generating keys, divides the key into segments, separated by slash “/”, where each segment represents one step in the hierarchy of identifiers distinguishing a unique parameter. For convenience from now on I’ll call the unique instances of a segment “subkeys”, (for example in a key “/aaa/bbb/cc/dddd/” “aaa” could be one possible subkey for the first segment of the example key).
- The order in which these segments appear in the key can vary and the “Get” function requires it to be exact (changing the order of key segments can cause different than intended “Get” behavior). On the other hand “Autoget” function identifies the type of segments (involves string compare which is slow with large keys) and doesn’t care about their order in the key.
- Some segments are common for many keys (for example the segment that identifies the platform generation would be the same for all the parameters used in that platform and there are few different platform generations). Moreover, as each segment is designed to be readable by humans, they are not simply
coded with the minimal number of characters, but instead names are used, sometimes resulting in long subkeys (for example a 17 characters long subkey has 136 bits, although not even 1/10 as many bits may be needed to distinguish all different possibilities for that segment).

- Some software generates a key by connecting parts of the key together and the ability to do that depends on maintaining the modular nature of the keys.

**The proposition**

A vast improvement in both speed and memory footprint of the parameter database could be achieved by the means of database key compression (coding the subkeys of commonly appearing key segments into as few bits as mandatory to distinguish between different subkeys). The common segments encoding could be done by using #define directives, and would require the following steps:

1. The first thing required would be to count the number of occurrences of all possible subkeys in the entire set of keys and to generate a list of subkeys that occur often enough to benefit from compression. To do that, the parameter database precompiled form (.txt with all the parameters and keys) could be parsed by a simple program that reads all the keys and simply counts the number of occurrences of each unique subkey throughout the entire database and would save the subkeys that occurred frequently enough to warrant their compression.

2. Once the list is ready, use #define directive to map each unique subkey to a unique symbol which can be defined (a char or a short string). Don’t use symbols that you can’t redefine, like numbers or symbols that have syntax meaning (quote “, slash /, etc.) or are used somewhere else in the code, or symbols that match some existing subkey. Also undefine them as soon as possible to limit the exposure to the rest of the code. Other then that, the minimal number of characters should be used to represent all distinct subkeys. (For example start from a char that is represented in binary form as “00000000”, and increment it by one for each new subkey as long as it’s ! =<forbidden_symbol>. When can not further increment due to insufficient number of bits – use one more character).
This way as many as 65536 (minus the forbidden symbols) of most common subkeys could be coded by using only 2 characters, and since #define was used, they would appear in the code in their original uncompressed form, thus maintaining human readability and API unchanged. Since only the subkeys that occur many times would be coded – the additional memory taken by all the “#define” would be much smaller then the memory saved. Moreover the hash function calculations and string comparisons take less time with shorter keys, thus the database speed would also be improved.

Note: there is no advantage in compressing subkeys that appear only once or twice, as one copy of the original subkey would still need to be stored for the #define, and the compressed form of the key would appear in both the database and the #define, thus using even more memory space. Also compressing the entire keys is not an option, as to be able to translate every key back to the original form would require storing all the unique keys, and unlike the subkeys, all the keys are unique. On the other hand, not having the capability to retrieve the original form of the key would require altering every single program that uses the database to use the new keys. The user readability would also be lost. Moreover, sometimes keys are generated by connecting parts of the key together and the ability to do that depends on maintaining the modular nature of the keys.

**Alternative method**

If the subkeys are successfully compressed to the size of only 2 or 3 characters each, the 1-character separators between them can take 1/3 or 1/4 of the entire key size. Thus even more efficient, but also more difficult to implement method would attempt to eliminate the segment separators by using set size subkeys for all the segments except the last one, which identifies individual parameters in some set. This way the segment separators “/” would no longer be needed, as the locations of the segments in the key would always be the same.

The subkeys that are compressed, could still be #defined individually in a similar manner to the earlier proposed implementation, however converting such compressed key back to its original uncompressed form would include inserting slash separators between the segments.

In this scenario, all the subkeys of all the segments, other then the segment indicating individual parameters, would have to be compressed. Since the key segments can be in a mixed order and
preserving that order is necessary – the number of characters needed to represent all the segments would have to be the same, or else the location of each segment could not be known without separators. Therefore the size of all compressed segments would have to be equal to the number of bits needed to represent the largest compressed subkey in any segment. The size of the key would be equal to the sum of all the segments. As the subkeys in segments other than the one indicating individual parameters would mostly appear more than once through the entire database (because they represent some set / type of keys, common for groups of individual keys) such compression would result in saving memory. The last segment indicating individual parameters could then be simply appended to the key without being modified.

Unfortunately not all segments are used in all the keys. To know both the location of the uncompressed segment and its length it would be necessary to know both the total number of segments in the key and the location of the uncompressed segment. Indicating both things would take only 1 character: 4 bits to code the number of segments (there are less than \(2^4\) segment types used in the keys) and 4 bits to code which of these segments is the uncompressed one. That’s a small price to pay for avoiding all the “slash” (8 bit character) separators between segments in terms of memory footprint, but it requires changing the way the parameter database handles the keys.

Finally this method would outperform the previous proposition only if the number of characters needed to represent the compressed segments would be the same as in the previous implementation, as using just one more character in each segment would take the same amount of memory as using 1-character separators between them. Moreover with this method we are forced to compress all the segments other than the one indicating individual parameters.

The max compressed segment size would end up being the same if all the subkeys in these segments could be coded using 2 characters, which is possible and even likely considering that 2 characters = 16 bits = 65,536 different combinations and the entire parameter database has only around 50,000 parameters, where it’s only necessary to code subkeys of segments that represent sets / types of keys, which are common for groups of individual keys, and repeat a lot.

**Additional risks and mitigation**

With compressed keys, a single bit flip (for example caused by a transient hardware fault) in the key would cause a different parameter
to be pointed to, instead of the parameter being simply not found. Solution: use 1 check bit per key to indicate if the number of “1” in the key is even or uneven – single error detection. Else use Humming code with 2 error detection and 1 error correction capability at the cost of 2 bits.

6.2. Proposition 2: Cleaning the database contents

The cleaning of database contents affects which parameters are stored but not the data structure implementation, or the format of the keys, therefore it could be used with the new parameter database implementations as well as the key compression. Only using this method with the Presorted AssocVector implementation of the database would result in very large time spent on removals of unnecessary parameters, as the Presorted AssocVector implementation is optimized only for initial build time, but not for later insertions and removals of the key-data pairs.

Background specific to proposition 2

• The parameter database before compilation is stored in a .txt file and contains all the parameters for every target system generation, revision and configuration. First it is filtered to exclude parameters for platform generations and other configurations known at the time to be different or unsupported by the target system. The resulting filtered .txt file is then used as a source file when compiling the first parameter database known as the database1.

• Some target system implementations can generally support different power supply and other parameters, but a specific target system may use only one of them and have explicit driver set. As a result many unused parameters still remain in database1 after initial filtering.

• Additional parameter filtering is done when initializing the target system when the exact target configuration becomes known and additional unsupported parameters can be identified. This filtering concludes with creation of database2 which contains pointers to the parameters in database1 that remain after the second filtering. However nothing is removed from database1 it self.
• Both database1 and database2 are stored in DRAM and remain there as long as the target system is on.
• The parameters are retrieved from the database using functions “Get” and “Autoget”, depending on the application. “Get” searches explicitly database1, while “Autoget” searches explicitly database2.
• Some parameters from database1 might be used by the target system as initial parameters in early stages of system initialization, but later be filtered out in database2 as a parameter more specific to that unit is found. For example due to driver dependencies some default parameters that shouldn't be used may be used for components before the exact component configuration is identified and the correct parameter is selected. Such behavior is considered a bug, but it may still be hidden in some older code.

The proposition

Use the pointers in database2 to remove unneeded parameters in database1:

Once the database2 is created, it should be possible to remove all the parameters not pointed to by database2 from the database1 without any consequence other then the reduced memory consumption. As a result, database2 it self would become unnecessary and could be removed, as the “cleaned” database1 would now have the same concise contents as database2.

If complete removal of database2 would be overly complicated due to “Autoget” function (which accesses specifically database2) being used in too many applications, the “Autoget” function it self could be modified to search parameters in database1 instead of database2. Then the application interface of the parameter database would remain unchanged and all the old code using it would still be supported. There may be too many places where “Autoget” is used, but only one place where the function it self is defined.

The main concern remains that some parameters that shouldn't be used and don’t belong to that target unit configuration, may nonetheless be accessed as defaults before the right component configuration is identified, or due to some other coding mistake. Even though such parameters are not truly needed for the system, an attempt to access parameter that does not exist could cause a crash.
Should it be decided that it is too costly to fix every such a bug, the cleaning of database1 could still be done at some point of system initialization when all the dummy default parameters have already been replaced by correct ones. Once the specific parameter is found, these older parameters will not be referred to again unless the target unit is rebooted, in which case the entire parameter database will be loaded to the DRAM all over again. Unfortunately there is no guarantee that cleaning database1 after initialization would free additional DRAM before the peek demand.

Risks

Should some code explicitly request parameters that do not belong to that unit configuration (either as a dummy default, as explained above, or for some other reason), these parameters would not appear in database2. Theoretically this should never happen, but in practice a large amount of old code needs to be controlled, including code which might be written even before strict guidelines were established, or written by contractors who may not have familiarized themselves with those guidelines well enough or failed to follow them.
7. References


[18]


[19]