Alteration Predictive Hybrid Search

Micheli Kurt
e1026558@student.tuwien.ac.at

September 13, 2018

ABSTRACT - Providing two different search algorithms for the same data leads to a choice and a possible lose of performance. Especially when the two search algorithms are efficient in different scenarios. The dynamic binary search works in logarithmic time and the static order preserving minimal perfect hash map (OPMPHM) has constant search time but must be constructed in liner time. Every data alteration leads to a complete reconstruction of the static OPMPHM. This makes the data alterations to the crucial point for the performance and the choice of search algorithm. The proposed hybrid search combines both search algorithms with a modified branch predictor. The modified branch predictor speculates about the data alterations and determines what search algorithm would be faster. The results made with random cases had shown that the hybrid search is except for small data sizes (elements < 600) almost always faster compared to the standalone binary search. The performance increase strongly depended on the measured hardware. In the random cases where the hybrid search is faster, on average $\approx 8.53\%$ to $\approx 20.92\%$ of time was saved.

Contents

1 Introduction

A hybrid is differently composed. One component of the hybrid search is a dynamic search algorithm. The dynamic search algorithm efficiently handles data that is often altered. The other component of the hybrid search is a static search algorithm. The static search algorithm is optimal for data that is seldom altered.

The dynamic search algorithm is a binary search that operates on always sorted elements. The static search algorithm is an order preserving minimal perfect hash map (OPMPHM). The OPMPHM is a static hash map algorithm thus there is no single element insertion and deletion operation. The OPMPHM is build for a set of elements and when only one element is removed or added the OPMPHM must be build again.

The hybrid search combines both search algorithms by using a modified branch predictor. The CPU uses a branch predictor to speculate at branches about the execution of the next program statement. To keep the execution pipeline also at branches always filled. The hybrid search speculates about data alterations with a modified branch predictor. This speculation is used to determine if the dynamic or static search algorithm would be faster.

1.1 Goal

Elektra[1] is a configuration framework and has the search operation `ksLookup (...)`, that already implements the dynamic search algorithm. The goal of this thesis is the extension of Elektra's search operation with an OPMPHM, but without any change of the Elektra API. Providing two search algorithms will inevitably lead to a choice and picking the wrong search algorithm could decrease the performance of Elektra's search operation. To transfer the choice away from Elektra's API and to increase the performance of Elektra's search operation the hybrid search is developed.

Before developing the hybrid search the OPMPHM algorithm must be implemented in Elektra. The OPMPHM build process works in iterations, each iteration is a random search for an OPMPHM. The iterations are repeated until an OPMPHM is found. The number of iterations until an OPMPHM is found depends on two constants. The OPMPHM build relies also on sub components such as hash function and pseudo random function. There is less known about good performing values for those constants and there is no knowledge how the OPMPHM build performs with those sub components. Since the OPMPHM is static and the OPMPHM build searches randomly it is important for the overall hybrid search performance that the OPMPHM build performs well. Therefore

[1] libelektra.org

the first research question is solely concerned with the runtime of the OPMPHM build:

RQ (i) Is the runtime of the OPMPHM build minimal?

After the OPMPHM algorithm is implemented in Elektra it is compared with a common hash map algorithm:

RQ (ii) How is the time performance of Elektra's OPMPHM build in comparison to the hsearch[2] build?

The hybrid search is evaluated in the context of the goal of this thesis and therefore compared with Elektra's standalone binary search. Although configurations tend do not have much alterations the hybrid search is evaluated harder with the research question:

RQ (iii) How much superior and how much inferior is the hybrid search time compared to the standalone binary search time in random cases?

2 Background

2.1 Hash Functions

Hash functions take arbitrary input of arbitrary length and calculate a fixed length output influenced by the input. A good hash function gives on the slightest change of the input a different output. Hash functions can have collisions, meaning that there are two different inputs leading to the same output. Collisions can not be avoided in general, but good hash functions rarely have collisions.

A ***seeded hash function*** is an extension of a hash function that additionally takes (beside the arbitrary input of arbitrary length also) a seed as input. A ***seed*** is a fixed sized random data. The output of a seeded hash function is influenced by both inputs. One seeded hash function in combination with multiple and different seeds represents multiple and different hash functions.

2.2 Hash Maps

A ***hash map*** is a data structure, that supports constant time search operations. Each element has a name and a value, the element's name is usually a string and the element's value is arbitrary. A ***search operation*** is a procedure that takes as input an arbitrary element's name and returns the whole element or not found. Search operations are used to access the element's value, only by knowing the element's name. Hash maps pass the element's name to hash functions and use the output for arithmetic calculations, to find the element. The time complexity of a hash function used in a hash map is constant, because the complexity calculation considers only the number of elements and not their length. Thus the hash map's search operation has constant time. Unlike a binary search algorithm where the complexity grows with the number of elements. Hash maps support also insertion and deletion operations. A search for

[2]Standard C library function http://linux.die.net/man/3/hsearch

an element can only be successful if the element was inserted in the hash map before.

The essence of a hash map is its ***mapping scheme*** that defines how the arithmetic calculation is done. The mapping scheme includes one or more hash functions and describes how an element's name is mapped to the element. Hash maps use buckets, that contain the elements inserted in the hash map. Let m be the predefined number of buckets. A mapping scheme would be:

$$hashFunction(name) \ mod \ m$$

Where *mod* is the modulo operation, that limits the output of the hash function to the buckets $\{0, \ldots, m-1\}$. When the number of buckets m changes, all elements that were inserted in the hash map need to be reinserted. This reinsertion is needed to make sure that all elements are in the right buckets after the number of buckets changed. The reinsertion of a set of elements previously in the hash map is known as ***rebuild***. A ***build*** is the creation of a hash map over a set of elements. During the build a empty hash map is created and each element inserted into it. The number of buckets (m) is not fixed, it depends on the number of elements in the hash map (n) and the desired load of the hash map. The hash map ***load*** α is defined as $\alpha = \frac{n}{m}$. For example a load of 0.25 with 50 elements would lead to 200 buckets. When the load is over 1 there must be a bucket that contains more than one element, this destroys the property of constant time search operations, because an extra search in the bucket is needed. As long as the maximum number of elements is known, hash maps are suitable for data that changes often, because expensive rebuilds can be avoided.

2.3 The Order Preserving Minimal Perfect Hash Map

The Order Preserving Minimal Perfect Hash Map, referred to as ***OPMPHM***, is a hash map satisfying all of the following properties:

Order Preserving Each element to index with the hash map has a predefined order, denoted as $order(s), s \in S$, where S is the set of element to index with the hash map. The mapping scheme of the hash map can be influenced to represent this predefined order. In the context of a common hash map the order of the influenceable mapping scheme defines to which bucket each element belongs.

Minimal The resulting hash map's size is $\mathcal{O}(n)$, where n is the number of elements in the hash map. Basically the common hash map is minimal but the load of a minimal hash map is maximized.

Perfect The search operation always takes $\mathcal{O}(1)$ time.

The static and randomized OPMPHM algorithm used in this thesis is based on the work of Botelho[2] and Czech et al.[4]. There is no single element insertion and single element deletion operation, this makes the algorithm static. The OPMPHM is constructed for a specified set of elements (S), to represent an index over the set of elements (S), where each element must have an order. This construction is the OPMPHM build. When the specific set related to the build OPMPHM or a single order of an element changes, the OPMPHM must be build again. This separates the OPMPHM for common hash maps that support

single element insertion and single element deletion operations. The OPMPHM algorithm is a Las Vegas type. Randomized ***Las Vegas*** algorithms give always a correct result but the runtime is random. Therefore the OPMPHM build runtime is random.

2.4 Elektra

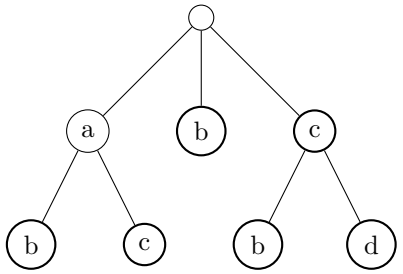
Elektra[3] is a library that implements access to a configuration storage. Configurations contain user preferences or other application settings and a configuration storage makes configuration permanent. The configurations are represented as keysets, a ***keyset*** is a sets of keys and a ***key*** consists of a name and a value. The key name is a sting and the key value is arbitrary data. Each key in a keyset has a unique name and the key names organize the keys in a hierarchy.[8]

This example shows a keyset and its hierarchy tree:

(a) Key Names

```
/a/b
/a/c
/b
/c
/c/b
/c/d
```

(b) Hierarchy Tree



In the hierarchy tree all vertices except the root have names (strings). In this example hierarchy tree every thick vertex represents a key, not all vertices must be keys (`/a` is not a key), but the leaves have to be keys. Every vertex except the root has a parent and vertices can have children. All the children's name of a parent must be unique. The key names are constructed from all possible paths from the root to each key vertex. In all possible paths the vertices are substituted by there name and the edges by a "/". The level of a vertex is the number of edges minus 1 on the path to the root. In this example vertex `a` has level 0.

The keys are stored in an array sorted by there name in the keyset, thus the binary search takes only $\mathcal{O}(\log_2(n))$ time, where n is the number of keys in the keyset. The OPMPHM just extends the keyset, because the OPMPHM represents an index and requires no keys reorganization.

2.5 Branch Prediction

A ***dynamic branch predictor*** predicts with the knowledge of previous binary events the outcome of a single binary event. The CPU's dynamic branch predictor distincts the branching instructions of a program by there addresses. More sophisticated dynamic branch prediction algorithms are applicable with this additional information. However on the level of a portable software library it is impossible to distinct between the API function invocations. Thus in Elektra's

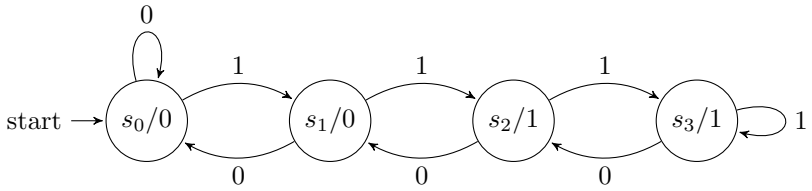
[3] libelektra.org

search operation `ksLookup (...)` it is indeterminable which function invocation of the program invoked the function. This forces the usage of a simpler dynamic branch prediction algorithm.

The hybrid search uses the dynamic branch prediction algorithm from Yeh and Patt[10], where the predicted events where modified. The dynamic branch predictor from Yeh and Patt[10] keeps track of the previous events in a history and analyzes patterns in the history with a prediction automata. The dynamic branch predictor consists of tree main components: prediction automata, history register and pattern history table.

Yeh and Patt[10] described various prediction automatas, this one performs best:

Figure 2: The Prediction Automata



The prediction automata has four states and s_0 is the starting state. The binary events on the arrows are used to analyze the last outcome and the binary events in the states are used to predict the new outcome. The prediction is pattern based but for this example assume there is only a prediction automata. On the first prediction the automata predicts 0. At the second prediction the last outcome was 1 and the automata analyzes this with a transfer to state s_1. Now the prediction automata predicts again 0. On the third prediction the last outcome was again 1 and the automata analyzes this with a transfer to state s_2. Now the prediction automata predicts 1.

To establish a pattern-based analysis the dynamic branch predictor uses a history register and a pattern history table. The history register is a fixed size sequence of 0 or 1 and keeps track of the last events. For example the following history 0101 with length 4 represents the alternating events of 0 and 1. For each possible value of the history register there is an entry in the pattern history table. An entry of the pattern history table stores a state of the prediction automata.

The prediction starts with a pre-prediction for the actual history (history register value). So when the predictor sees the next time the same history it knows what to predict. The pre-prediction step uses the history register value to extract out of the pattern history table the prediction automata's state. The prediction automata's state is updated with the new state, resulting from the old state and the last outcome. After the pre-prediction the history (history register value) is updated by shifting it with the last outcome to the left. The prediction uses the new history (history register value) to extract from the pattern history table another pre predicted state and the state determines the prediction.

This example shows a prediction with a 0 as the last outcome:

history register	011(3) ← 110(6)

	state		state
0	$s_0/0$	4	$s_3/1$
1	$s_1/0$	5	$s_1/0$
2	$s_1/0$	6	$s_2/1$
3	$s_3/1 \leftarrow s_2/1$	7	$s_0/0$

On the left is the history register (with length 3) and on the right is the pattern history table. The ← represents a value update. The pre-prediction updates the third entry of the pattern history table with the last outcome (0) and the previous state s_3. The history register is updated with the last outcome (0) and the sixth entry of the pattern history table determines the new prediction 1.

The space complexity of this dynamic branch predictor is exponential in the history register length, since for every history register value an entry in the pattern history table is needed. Each entry needs two bits to store the four states of the prediction automata.

3 Theory

3.1 Preliminaries

An ***r-uniform r-partite hypergraph*** for $r > 1$ is a tuple $G_r = (V, E)$, where $V = V_1 \cup \ldots \cup V_r$ are the vertices, separated in r components and $E = \{(v_1, \ldots, v_r) : v_1 \in V_1, \ldots, v_r \in V_r\}$ are the edges, connecting r vertices from separate components. Note that with $r = 2$ the r-uniform r-partite hypergraph has two components and each edge connects two vertices in the separate components. This corresponds to a bipartite graph.

Czech at al.[5] gives a definition of an ***acyclic*** r-uniform r-partite hypergraph: "An r-uniform r-partite hypergraph is acyclic if and only if some sequence of repeated deletion of edges containing vertices of degree 1 yields a graph with no edges."

3.2 The OPMPHM Algorithm

The core concept of this OPMPHM algorithm is the randomized search for acyclic r-uniform r-partite hypergraph G_r with $|E| = n$ and $|V| = cn$. The hypergraph is constructed for a given set of elements S with $|S| = n$. The constant c is the vertices factor that influences the number of vertices in the randomized r-uniform r-partite hypergraph. Each component V_i $1 \leq i \leq r$ of V has the same number of vertices $\lceil cn/r \rceil$.

The r-uniform r-partite hypergraph construction relies on r seeded hash functions $f_1, \ldots, f_r$. Each seeded hash function is defined as $f_i : S \to V_i$ for $1 \leq i \leq r$ and maps each element $s \in S$ to r vertices each one in a separate component of G_r. Each element of $s \in S$ in combination with all seeded hash functions $f_1, \ldots, f_r$ represents an edge $e \in E$ of G_r.

Should the constructed r-uniform r-partite hypergraph have cycles, new seeded hash functions $f_1, \ldots, f_r$ will be chosen until the r-uniform r-partite hypergraph is acyclic, this procedure is described by Botelho[2] as the mapping

step of the RAM algorithm. This approach makes this OPMPHM algorithm a randomized algorithm of the Las Vegas type.

After the mapping step when an acyclic r-uniform r-partite hypergraph is found, comes the assignment step. The assignment step assigns for every vertex an integer, stored in the function $g : V \to \{0, 1, \ldots, n-1\}$. During the assignment every edge's (representing an element) vertices $e = (v_1, \ldots, v_r), v_1 \in V_1, \ldots, v_r \in V_r$ are assigned to sum up to the predefined order of the element $order(s)$:

$$order(s) = (g(v_1) + g(v_2) + \cdots + g(v_r)) \ \ mod \ \ n$$

The search for an element uses the seeded hash functions $f_i(s) = v_i$ for $1 \leq i \leq r$ to calculate the vertices. The assigned values of those vertices are then used to calculate the predefined order of the element and conclude the search. The resulting mapping scheme is:

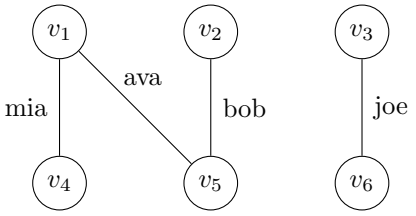
$$(g(f_1(s)) + g(f_2(s)) + \cdots + g(f_r(s))) \ \ mod \ \ n$$

The following example shows an OPMPHM of four elements and the corresponding r-uniform r-partite hypergraph with $r = 2$:

(a) Elements

S	Name	$order(s_i)$	f_1	f_2
s_1	bob	0	v_2	v_5
s_2	joe	1	v_3	v_6
s_3	ava	2	v_1	v_5
s_4	mia	3	v_1	v_4

(b) $G_2 = (\{v_1, v_2, v_3\} \cup \{v_4, v_5, v_6\}, E)$

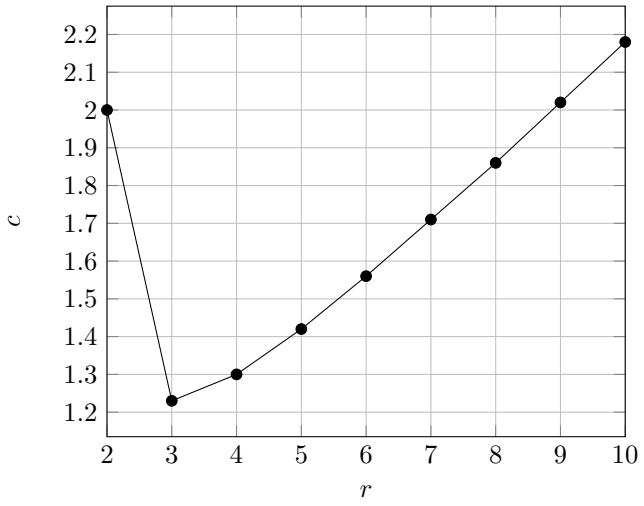


(c) $g()$

	v_1	v_2	v_3	v_4	v_5	v_6
g	2	0	3	1	0	2

A search for *joe* first calculates the vertices with the seeded hash functions $(g(v_3) + g(v_6)) \ \ mod \ \ 4$, than $g()$ gives the vertices integer values $(3 + 2) \ \ mod \ \ 4$ and the final result is the predefined order $5 \ \ mod \ \ 4 = 1$.

Botelho[2] gives a minimum c value for finding acyclic r-uniform r-partite hypergraph, to ensure that there are only $\mathcal{O}(1)$ r-uniform r-partite hypergraphs to construct. This minimum c value depends on r, the following function plot of $c(r)$ shows the values:



In this plot the x-axis is the number of components of the r-uniform r-partite hypergraph and the y-axis is the minimal c value to ensure a constant number of r-uniform r-partite hypergraph constructions.

The c value directly influences the space needed to store the OPMPHM, since the $g()$ function must be saved and $|V| = cn$. This plot of $c(r)$ shows that the space needed to store the OPMPHM is minimal at $r = 3$, since $c(3) \approx 1.23$.

The assignment of the vertices is known as the generalized perfect assignment problem. Czech et al.[5] described the ***generalized perfect assignment problem***: Given a r-uniform r-partite hypergraph where $|E| = n$, finding a function $g : V \to \{0, 1, \ldots, n-1\}$ such that the function $h : E \to \{0, 1, \ldots, n-1\}$ defined as:

$$h(e) = (g(v_1) + g(v_2) + \cdots + g(v_r)) \ \ mod \ \ n$$

is a bijection, where $e = (v_1, \ldots, v_r)$ and mod is the modulo operation. Only acyclic r-uniform r-partite hypergraphs assure a linear time complexity of the assignment. Because only for acyclic r-uniform r-partite hypergraphs it is guaranteed to have a linear time solution to the generalized perfect assignment problem.[5].

The assignment step is a generalized version of the assignment step described from Czech et al.[4] for bipartite graphs. The sequence of edges during the assignment cannot be arbitrary, since an arbitrary sequence could lead to the case that there is no assignable vertex for an edge. This can be the case when other edges assign the vertices before the edge is assigned. The assignment procedure uses the sequence of repeated deletion of edges, given by the acyclic r-uniform r-partite hypergraph definition. Since the r-uniform r-partite hypergraph is acyclic at this stage, there exists a sequence of repeated deletion of edges containing vertices of degree 1. The assignment step uses the reversed sequence of repeated deletion, thus it is ensured that every edge that will be assigned contains vertices of degree 1. The degree 1 vertices are not yet involved in other assignments of edges.

3.3 Related Work

Botelho[1] proposes a MPHM algorithm that constructs a MPHM through random cyclic undirected graphs with no self loops and multiple edges. The algorithm is not order preserving the order of the elements is not influenceable. The elements are mapped by the MPHM to a fixed number of buckets, through the missing order preserving property it possible that buckets stay empty. The algorithm consists of the steps: mapping, ordering and searching. The mapping step constructs with two hash functions the random graph and ensures by repeating that it contains no self loops and multiple edges. One element with the two hash functions represents an edge in the random graph. The ordering step divides the random graph in a cyclic and acyclic sub graph. The searching step first assigns the cyclic sub graph. This is done by traversing the cyclic sub graph in a breadth-first manner. The second step of the search is the assignment of the acyclic sub graph. This is done by traversing the acyclic sub graph in a deep-first manner. Due to the perfect assignment problem the assignment of the cyclic sub graph could lead to much empty buckets. These empty buckets are filled up with the assignment of the acyclic sub graph. It is possible that the fixed number of buckets is to small and the cyclic sub graph can not be assigned in this case the algorithm starts over and chooses other hash functions.

Fox et al.[6] proposes an OPMPHM algorithm that uses random cyclic bipartite graphs. The algorithm uses additionally a bit for each vertex to handle the vertices of the cyclic sub graph. The algorithm consists also of the steps: mapping, ordering and searching. The mapping step constructs with three hash functions the random graph. The first and second hash function define the edges the third hash function helps with the assignment of cyclic sub graph edges. The ordering step divides the random graph in a cyclic and acyclic sub graph. The searching step assigns randomly first the cyclic sub graph and then the acyclic sub graph. The bit for each vertex gives more possibilities to solve the perfect assignment problem, since it causes a mapping scheme change for that element. Also in this OPMPHM algorithm due to the perfect assignment problem it is possible that the assignment of the cyclic sub graph fails, if this happens the algorithm starts over.

Czech et al.[4] uses acyclic bipartite graphs. This algorithm is a special case of the in this thesis proposed algorithm with $r = 2$.

4 Implementation

4.1 Randomness

Every randomized algorithm needs a pseudo random function. This OPMPHM implementation uses a non cryptographic pseudo random number generator from Ray Gardner[4], it is based on Park et al.[7] and Carta[3]. The pseudo random number generator uses Schrage's method[9] to avoid overflow problems and is the function:

$$f(z) = 16807z \ \ mod \ \ 2^{31} - 1$$

The seeds are represented as `int32_t`. Due to the shape of this function and the datatype used for the seeds, the initial seed should not be 0 or bigger

[4] www8.cs.umu.se/ isak/snippets/rg_rand.c

than $2^{31} - 2$. Note that when the seed is 0 or $2^{31} - 1$ the pseudo random number generator generates a series of 0 and is useless. The initial seed is simply generated by the C standard library function `time(...)`, with parameter `NULL`. The `time(...)` returns with parameter `NULL` the passed seconds since a fixed date in the past, the main advantage of the `time(...)` is the portability. When the value returned by `time(NULL)` is bigger than $2^{31} - 2$ a modulo operation with $2^{31} - 1$ bring the initial seed back to range. In the most unlikely case that after the modulo operation the initial seed is 0 it will be set to 1.

4.2 Seeded Hash Function

The seeded hash function is `hashlitte(....)` from Bob Jenkins[5], it supports big and little endian systems. The seeded hash function first initializes three `uint32_t` variables with the sum of:

- the length of the string to hash
- the seed
- a hard-coded value

Then a loop over the string adds always three `uint32_t` parts of the string to the three variables. These three variables then are mixed by a procedure using xor, addition, subtraction and rotation. At the end of the string the third variable is returned. The hard-coded value is needed for situation where the passed seed is to small.

4.3 Recursive Acyclic Test

To test if a r-uniform r-partite hypergraph is acyclic Czech at al.[5] proposed a recursive $O(n)$ time algorithm, where n is the number of edges. The algorithm has two components:

1. loops over the vertices, invocate the recursive procedure for every vertex with degree 1
2. recursive procedure with a vertex as parameter:
    - remove the edge that connects the vertex from the r-uniform r-partite hypergraph
    - if other by that edge connected vertices have degree 1, invocate the recursive procedure

Since the recursion works like a stack the edges are processed in a first in last out manner. The recursive process handles a connected component of the r-uniform r-partite hypergraph, as long as there is a degree 1 vertex and the main loop ensures that all of the connected components are treated. When at the end of the main loop the r-uniform r-partite hypergraph is empty, then the r-uniform r-partite hypergraph was acyclic.

[5] http://burtleburtle.net/bob/c/lookup3.c

4.4 r-uniform r-partite Hypergraph

The following structures show how the r-partite r-uniform hypergraph is stored. One vertex of the r-uniform r-partite hypergraph is represented by the following structure:

```
typedef struct
{
    uint32_t firstEdge;
    uint32_t degree;
} OpmphmVertex;
```

Listing 1: Vertex of the r-uniform r-partite hypergraph

The vertex stores a list of edges that are connected to that vertex and saves in `firstEdge` the index of the first edge of the list and in `degree` the length of the list. The edge is represented by:

```
typedef struct
{
    uint32_t order;
    uint32_t * nextEdge;
    uint32_t * vertices;
} OpmphmEdge;
```

Listing 2: Edge of the r-uniform r-partite hypergraph

The `vertices` and `nextEdge` array have always r entries. One edge stores the $order(s)$ in `order`, this value is the predefined order of $s \in S$. The `nextEdge` entries are the next edges in the r lists, since each edge connects r vertices in r separate components, each edge is exactly in r lists. The `vertices` store the indices of the vertices connected by the edge. This field is an `uint32_t` because that is the return type of the seeded hash function. The seeded hash function decides to which vertices an edge is connected. This indexed edge and vertex representation allows an efficient traversal over edges and vertices connected by those edges, it also allows an efficient construction of the r-uniform r-partite hypergraph and deletion of edges. The whole r-uniform r-partite hypergraph is stored in:

```
typedef struct
{
    OpmphmEdge * edges;
    OpmphmVertex * vertices;
    uint32_t * removeSequence;
    uint32_t removeIndex;
} OpmphmGraph;
```

Listing 3: r-uniform r-partite hypergraph

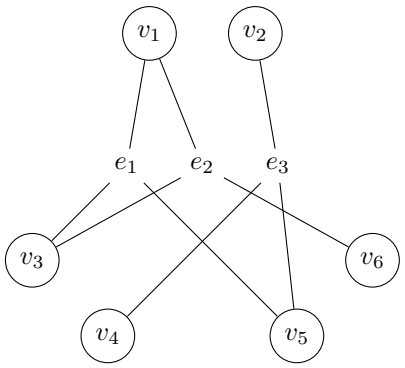
The `edges` is an array with n entries and holding all the edges. The vertices is also an array with $\lceil cn/r \rceil r$ entries and is holding all the vertices. In this representation the r-uniform r-partite hypergraph is considered empty when all `vertices` have a degree of 0. The `removeSequence` is an array with n entries. The recursive acyclic test fills the `removeSequence` with the help of the `removeIndex`. After a successful mapping step, when the r-uniform r-partite hypergraph is

acyclic the `removeSequence` contains the indices of the edges. The `removeSequence` is than used by the assignment step, to assign the edges in reversed sequence.

All three structures, the `OpmphmVertex`, the `OpmphmEdge` and the `OpmphmGraph` are disposable and only needed to assign the values of the $g()$ function. Storing the `OpmphmGraph` permanently would be a waste of memory.

The following example shows how an r-uniform r-partite hypergraph is stored with the `OpmphmGraph`. The first table represents the `vertices` array and the second the `edges` array.

(a) $G_3 = (V = \{v_1, v_2\} \cup \{v_3, v_4\} \cup \{v_5, v_6\}, E = \{e_1, e_2, e_3\})$



(b) Memory Image of OpmphmGraph from G_3

Vertices	v_1	v_2	v_3	v_4	v_5	v_6
firstEdge	e_2	e_3	e_2	e_3	e_3	e_2
degree	2	1	2	1	2	1

Edges			e_1			e_2			e_3
order			2			1			0
r	1	2	3	1	2	3	1	2	3
nextEdge				e_1	e_1				e_1
vertices	v_1	v_3	v_5	v_1	v_3	v_6	v_2	v_4	v_5

An iteration over the vertices that connect an edge uses the `vertices` array of the `OpmphmEdge` structure. An iteration over the edges that are connected to a vertex first looks in the `degree` to determine the iteration length. The starting edge is the `firstEdge`, the next edge is stored in the corresponding component id (r) of the edge's `nextEdge` array. The iteration over all edges of v_5 first gets to the edge e_3 and than from e_3's `nextEdge` array with the component id $r = 3$ to the edge e_1. Note that G_3 is acyclic, with the `removeSequence` $= \{e_3, e_1, e_2\}$.

4.5 The OPMPHM Build

The OPMPHM build is separated in two steps: mapping and assignment. When both steps succeed the OPMPHM is stored in the following structure:

```
typedef struct
{
```

```
    int32_t * hashFunctionSeeds; // r seeds for f_0,...,f_{r-1}
    uint8_t rUniPar; // actual value of r
    size_t componentSize; // ⌈cn/r⌉
    size_t * graph; // stores the g()
    size_t size;
} Opmphm;
```

Listing 4: The OPMPHM

The `hashFunctionSeeds` stores seeds used for the seeded hash functions. The `componentSize` stores the size of one component of the r-uniform r-partite hypergraph. The `size` is the size of the `graph` in bytes and is only needed in case the OPMPHM structure is serialized. The OPMPHM structure contains all information to perform a search operation with the OPMPHM.

4.5.1 The Mapping Step

The mapping step finds acyclic r-uniform r-partite hypergraphs. This pseudo code describes the mapping step of the RAM algorithm introduced by Botelho[2]:

```
 1: seed = generateInitialSeed ()
 2: do
 3: {
 4:     // generate seeds for f_0,...,f_{r-1}
 5:     for r_i in {0,...,r-1}
 6:     {
 7:         seed = pseudoRandomFunction (seed);
 8:         Opmphm.hashFunctionSeeds[r_i] = seed;
 9:     }
10:
11:     // construct G_r
12:     G_r = ∅
13:     for n_i in {0,...,n-1}
14:     {
15:         for r_i in {0,...,r-1}
16:         {
17:             // determine vertex with hash function
18:             G_r.edges[n_i].vertices[r_i] =
19:                 hashFunction (element[n_i].name,
20:                 Opmphm.hashFunctionSeeds[r_i]) mod ⌈cn/r⌉;
21:
22:             // calculate vertex index
23:             v = r_i⌈cn/r⌉ + G_r.edges[n_i].vertices[r_i];
24:
25:             // add edge to G_r
26:             G_r.edges[n_i].nextEdge[r_i] =
27:                 vertices[v].firstEdge;
28:             G_r.vertices[v].firstEdge = n_i;
29:             ++G_r.vertices[v].degree;
30:         }
31:     }
32: } while (isCyclic (G_r))
```

Listing 5: Pseudo Code of the Mapping procedure

The while loop (line 2-32) searches for acyclic r-uniform r-partite hypergraphs. The procedure only stops when the recursive acyclic test (line 32 `isCyclic (`G_r`)`) returns `false`. The start of the procedure (line 1) is the generation one random initial seed. The for loop (line 5-9) expands the single random initial seed with the pseudo random number generator (`pseudoRandomFunction(...)` to r random seeds. The r random seeds are stored in the `Opmphm` (line 8) because the final OPMPHM needs the seeds for lookup. The acyclic graph search iteration starts with an empty r-uniform r-partite hypergraph (`OpmphmGraph` line 12). In the acyclic graph search iteration the inner for loop (line 15-30) appends for every element of $|S| = n$ an edge in G_r. The edge is appended to the lists in the r vertices. The vertices are determined with the seeded hash function (`hashFunction (...)` in line 18). The seeded hash function uses the elements name (`element[`n_i`].name`) and the respective random seed (`Opmphm.hashFunctionSeeds[`r_i`]`). When the procedure stops the recursive acyclic test `isCyclic (`G_r`)` has filled the `removeSequence` with the sequence of repeated deletion of edges. The sequence of repeated deletion of edges is needed in the assignment step.

4.5.2 The Assignment Step

The assignment step assign for every vertex of G_r a value in the `Opmphm.graph[]` array. The `Opmphm.graph[]` array represents the stored $g()$ function. The following pseudo code describes the generalized version of assignment step from Czech et al.[4]:

```
 1: for v_i in {0,...,⌈cn/r⌉r}
 2: {
 3:     isAssigned[v_i] = 0;
 4: }
 5: // assign in reverse sequence of deletion
 6: for rS_i in reverse (OpmphmGraph.removeSequence[])
 7: {
 8:     edgeToAssign  = G_r.edges[rS_i]
 9:
10:     // (r index, Opmphm.graph[] value)
11:     (assignableVertex, assignedValue) =
12:         getAssignableVetex (edgeToAssign);
13:
14:     // calculate vertex index
15:     v =  assignableVertex * ⌈cn/r⌉ +
16:     edgeToAssign.vertices[assignableVertex];
17:
18:     // assign assignableVertex
19:     if (assignedValue >= n)
20:     {
21:         assignedValue = assignedValue mod n;
22:     }
23:     order = order(element[rS_i])
24:     if (assignedValue <= order)
25:     {
26:         Opmphm.graph[v] = order - assignedValue;
27:     }
28:     else
```

```
29:     {
30:         Opmphm.graph[v] = (n - assignedValue) + order;
31:     }
32:     isAssigned[v] = 1;
33: }
```

Listing 6: Pseudo Code of the Assignment procedure

The assignment procedure uses the array `isAssigned[]` to store the assignment state of every vertex of G_r. First the state of all vertices is set to not assigned (line 1-4). The assignment step iterates over the edges in the reverse sequence of deletion and assign the connected vertices (6-33). The assignment of an edge uses a sub procedure `getAssignableVetex (...)` (line 11). The sub procedure assign all vertices of the edge except one and sums up the values of the assigned vertices. `getAssignableVetex (...)` returns a tuple where the first entry is the r index of the not assigned vertex (`assignableVertex`) and the second entry is the sum (`assignedValue`) (but more to this sub procedure later). A modulo operation brings the `assignedValue` in the range between 0 and n (line 19-22). Depending on the $order(s_i)$ of the element and the `assignedValue` the last vertex is assigned to fulfill the mapping scheme of the OPMPHM (line 23-31). At the end of an edge assignment the last assigned vertex is set to assigned (line 32).

The sub procedure `getAssignableVetex (...)` is specified by the pseudo code:

```
 1: procedure getAssignableVetex (OpmphmEdge edgeToAssign)
 2: {
 3:     assignableVertex = r;
 4:     assignedValue = 0;
 5:     for r_i in {0,...,r-1}
 6:     {
 7:         // calculate vertex index
 8:         v = r_i⌈cn/r⌉ + edgeToAssign.vertices[r_i];
 9:         if (!isAssigned[v])
10:         {
11:             if (assignableVertex == r)
12:             {
13:                 // found first assignableVertex
14:                 assignableVertex = r_i;
15:             }
16:             else
17:             {
18:                 // assignableVertex already found
19:                 // just assign some value
20:                 Opmphm.graph[v] = 42;
21:                 assignedValue += graph[v];
22:                 isAssigned[v] = 1;
23:             }
24:         }
25:         else
26:         {
27:             // vertex is assigned
28:             assignedValue += graph[v];
29:         }
30:     }
31:     return (assignableVertex, assignedValue);
```

```
32: }
```

Listing 7: Pseudo Code of the getAssignableVetex procedure

The parameter of the sub procedure is an edge to assign. The sub procedure iterates over all vertices of that edge to assign (line 5-30). The first not assigned vertex is the `assignableVertex` (line 11-15), that will be assigned later by the assignment procedure. All not assigned vertices encountered after the `assignableVertex` is found are assigned with an arbitrary value (line 16-23). The vertices are set to assigned and the `assignedValue` is updated. Should the iteration encounter an assigned vertex only the `assignedValue` is updated. The reversed sequence of repeated deletion of edges created during the recursive acyclic test guarantees that there is always at least one not assigned vertex.

In the implementation the `isAssigned[]` array of the assignment step is not needed. The `Opmphm.graph[]` array contains also the is assigned information since $0 \ mod \ n = 0$ and $n \ mod \ n = 0$, the 0 value is used to represent not assigned and the n value is used to represent the 0.

The source code of the OPMPHM is in Elektra's GitHub[6] and designed to work also standalone.

4.5.3 Complexity

Botelho[2] states that the mapping step of the RAM algorithm is expected to run in $O(nr)$, if the c value is above the minimal value. The assignment step runs in $O(nr)$. Since r is assumed to be constant both time complexities and the overall time complexity of the OPMPHM build is $O(n)$.

The space complexity of the stored OPMPHM depends completely on the value of c, since the $g : V \to \{0, 1, \ldots, n-1\}$ function stored in the OPMPHM structure in the `graph` field has $\lceil cn/r \rceil r$ entries. During the build the r-uniform r-partite hypergraph must also be stored, the space needed is dominated by the number of vertices and therefrom detents also on the c value.

4.6 The Hybrid Search

Elektra's binary search takes due the always sorted keys $\mathcal{O}(\log_2(n))$ time, where n is the number of keys in the keyset. The OPMPHM search time is constant and therefore on long term faster, but the OPMPHM must be build before search. The OPMPHM build complexity is linear in the number of keys in the keyset and thus expensive compared to the binary search. The OPMPHM is build for a static keyset and must be rebuild only if a single key changes. On one hand is the binary search that is optimal for dynamic keysets and on the other hand is the OPMPHM that is the best for static keysets. The crucial points are the keyset alterations and the number of searches made between two alterations.

Let k be the number of searches between two alterations and n be the keyset size. There is a point where the binary search usage costs more than the OPMPHM usage because there exists a k such that $\mathcal{O}(k \log_2(n)) > \mathcal{O}(n)$. The OPMPHM usage could also be more expensive than the binary search usage, when the number of searches is too small. Therefore a sequence of searches

[6] https://master.libelektra.org/src/libs/elektra/opmphm.c

between two keyset alterations is or is not worth using the OPMPHM. The implementation relies on a benchmark based heuristic function that determines depending on n and k if it is worth using the OPMPHM or not.

The implemented dynamic branch predictor from Yeh and Patt[10] is modified to predicts if it is worth using the OPMPHM (1) or not (0). At the first search after a keyset alteration the predictor predicts if it will be worth using the OPMPHM or not.

The dynamic branch predictor is stored in a additional data structure:

```
typedef struct
{
    uint16_t history; // history register
    uint8_t * patternTable; // pattern history table
    size_t size;
    size_t lookupCount; // number of searches
    size_t ksSize; // keyset size
} OpmphmPredictor;
```

Listing 8: The Opmphm Predictor

The implementation limits the history register to 16 bits. To determine the last outcome with a heuristic function the data structure stores the `ksSize` and the `lookupCount`. The `ksSize` must be stored because the keyset's size is changed with an alteration of the keyset. The `lookupCount` keeps track of the number of searches made. The `size` is the size of the `patternTable` in bytes and is only needed in case the OpmphmPredictor structure is serialized. The source code of the modified branch predictor is in Elektra's GitHub[7].

4.7 Integration in Elektra

Elektra's keyset is extended to hold a pointer of an OPMPHM and modified branch predictor instance. Keyset alterations are crucial for both, not detected keyset changes entail wrong search results and non accurate predictions. The OPMPHM and the modified branch predictor need independent mechanisms to keep track of alterations. At an alteration the OPMPHM just gets freed. The modified branch predictor uses Elektra's keyset flags, that mark a keyset instance. A new keyset flag named `KS_FLAG_NAME_CHANGE` is created. Hooks are created to minimize the impact of the implementation to the API. The invalidation hook is invoked by all altering public and private API function invocations:

[7] https://master.libelektra.org/src/libs/elektra/opmphmpredictor.c

Name	Description
`ksClose (...)`	Resets a keyset and destroys all contained keys
`ksClear (...)`	Destroys all contained keys from a keyset, after this operation the keyset is empty
`ksAppendKey (...)`	Appends a single key to a keyset
`ksAppend (...)`	Appends a whole keyset to another keyset
`ksCopyInternal (...)`	Copies keys withing a keyset
`ksCut (...)`	Extracts the all keys from a keyset that are below a certain key in the hierarchy
`ksPop (...)`	Extracts the last key of the keyset

Table 1: Altering API Function

The API documentation[8] documents all public operations. The invalidation hook frees the OPMPHM and marks the keyset instance with the `KS_FLAG_NAME_CHANGE` flag.

The API has also coping functions, the coping hook is invoked by all coping API function invocations:

Name	Description
`ksDup (...)`	Duplicates a keyset but not the contained keys, there it is a flat copy
`ksDeepDup (...)`	Duplicates a keyset and the contained keys
`ksCopy (...)`	Replaces the content of a existing keyset with the content of another keyset

Table 2: Coping API Functions

The coping hook copies the OPMPHM instance and modified branch predictor instance.

The modified branch predictor decides what search to use, but the API user can overrule the predictor. Elektra's search options are extended with `KDB_O_OPMPHM` and `KDB_O_BINSEARCH`. When one of these options is set the modified branch predictor is overruled and not used for search. If the predictor is not overruled it uses the `KS_FLAG_NAME_CHANGE` flag to determine changes. When there is a change the modified branch predictor predicts what search to use, when there is no change the number of searches are counted for the heuristic function of the predictor. The modified branch predictor only takes actions above a certain keyset size, the `opmphmPredictorActionLimit` defines that limit.

[8] https://doc.libelektra.org/api/latest/html/group__keyset.html

5 Experiments

5.1 Benchmark Seeds

All benchmark data is generated randomly. The benchmark data is generated with the help of an initial seed and the pseudo random function. The statistical software R is the source of all random initial seeds. The function `runif(...)` generates uniformly distributed seeds in the range from 1 to $2^{31} - 2$.

5.2 Random Generated Keysets

The key names of a keyset influence the OPMPHM build and the binary search. The OPMPHM build relies on seeded hash functions that hash the key names and the binary search compares the key names. Always different key names are practically rare and do not evaluate the algorithms properly. The keysets for the benchmarks are generated by the highly adaptable `generateKeySet(...)` function, that outputs partial equal key names. With the `generateKeySet(...)` function it is not only possible the generate key names that start with an equal name, it is also possible to let them end with an equal name. The `generateKeySet(...)` takes an initial seed, the desired size and a keyset shape description. The generation constructs with the help of the pseudo random function a random hierarchical tree in a deep first manner and then translates the tree into a keyset. The ***keyset shape description*** has the following properties:

Characters type probability: The vertex names are generated randomly out of two pools, the alphabet and numbers pool and the special characters pool. This property influences the probability of a character being picked from the pools.

Vertex name length: A minimal and maximal vertex name length specifies the range of the name length, the generation randomizes the vertex name length between these values.

Number of children: Is specified by a function that defines the number of children for a vertex and will be invoked for every vertex in the hierarchical tree, during the deep first generation. The function has a variety of parameters to construct dynamic shapes. This function can also set a label for a sub tree and use it later in the generation to copy the whole labled sub tree at the used place. Labeling generates key names that end with an equal name.

Vertex probability: Influences the probability of a vertex being a key.

The following example keyset shape uses only alphabet and numbers characters. With a minimum vertex name length of 5, a maximum vertex name length of 9 and the vertex probability set to 0. The number of children function that uses the desired size and the actual level of the hierarchical tree:

$$numberOfChildren(\dots, size, level, \dots) = \begin{cases} 3 & \text{if } level < log_3(size) - 1 \\ 0 & \text{otherwise} \end{cases}$$

Generates the following keyset with size of 9 keys:

```
/plugin5/data0
/plugin5/data1
/plugin5/data5
/plugin46/data13
/plugin46/data63
/plugin46/data99
/plugin87/data41
/plugin87/data78
/plugin87/data84
```

Every vertex in the hierarchical tree of this generated keyset has 3 children as long as the level is under 2. Otherwise the number of children is 0. The vertex probability is 0, therefore only the leafs of the hierarchical tree are keys. With the labeling it would be possible to have equal sub trees for all `plugin*` vertices and would look like:

```
/plugin5/data0
/plugin5/data1
/plugin5/data5
/plugin46/data0
/plugin46/data1
/plugin46/data5
/plugin87/data0
/plugin87/data1
/plugin87/data5
```

All designed keyset shapes are defined in the `getKeySetShapes ()` function of the OPMPHM benchmark file[9].

5.3 Hardware

The time measuring benchmarks are executed on these tree systems (all data from the linux command `lshw`):

[9] https://master.libelektra.org/benchmarks/opmphm.c

i7-6700K	
CPU	Intel Core i7-6700K @ 4 GHz
L1 cache size	256 KB
L2 cache size	1 MB
L3 cache size	8 MB
RAM	16 GB DDR4 @ 2133Mhz
i7-3517U	
CPU	Intel Core i7-3517U @ 1.9 GHz
L1 cache size	128 KB
L2 cache size	512 KB
L3 cache size	4 MB
RAM	4 GB DDR4 @ 1600Mhz
1800X	
CPU	AMD Ryzen 7 1800X @ 3.6 GHz
L1 cache size	768 KB
L2 cache size	4 MB
L3 cache size	16 MB
RAM	64 GB DDR4 @ 2400 Mhz

Table 3: Hardware

Elektra is compiled with gcc version 6.3. The `cmake` is configured to `Release` what is equivalent to `-O 3`. Logging and debug is disabled.

5.4 Runtime of the OPMPHM Build

The OPMPHM build has two steps the mapping step and the assignment step. The runtime of the assignment step is fixed, but the runtime of the mapping step depends on number of r-uniform r-partite hypergraph to construct. The constuction of an acyclic r-uniform r-partite hypergraph depends on the two constants:

r the number of components in the r-uniform r-partite hypergraph

c the vertices factor that influences the number of vertices in the hypergraph

Botelho[2] defines the function $c(r)$ that specifies the minimal c values depending on r, to ensure that the number of r-uniform r-partite hypergraphs to construct is constant. Beside the minimal c value function from Botelho[2] there is no information about practical r and c values. The main task of this research question is the construction and evaluation of two heuristic functions `optimalR(n)` and `optimalC(n)`. Both depend on the number of keys in the keyset n and aim to minimize the runtime of the OPMPHM build.

RQ (i) Is the runtime of the OPMPHM build minimal?

The runtime of the OPMPHM build is minimized by minimizing the theoretical $\mathcal{O}$ time complexity of the mapping step.

5.4.1 Method

The huge time complexity of the combination from keysets, initial seeds, values for r and values for c forces the use of three benchmarks. All benchmarks measure the number of r-uniform r-partite hypergraphs to construct until the mapping step is successful. The mapping step is successful when the r-uniform r-partite hypergraph is acyclic.

The mapping benchmark is used to develop approximate versions of the two heuristic functions `optimalR(n)` and `optimalC(n)`.

The optimal mapping benchmark transforms the approximate versions of the two heuristic functions to the final heuristic functions. Evaluates also if the two heuristic functions lead to a minimal runtime of the OPMPHM build.

The all seeds mapping benchmarks gives a blink to the behaviour of the heuristic function over all possible seeds.

The mapping benchmark gives an overview how the mapping procedure behaves for different keyset sizes (n), r-partite r-uniform hypergraphs (r) and vertices factors (c). The benchmark uses the sequence $a_n = (10, 15, a_{n-2} * 2, a_{n-2} * 2, \ldots, 1280)$ for the keyset sizes, previous evaluations indicated that 1280 as maximum is suitable. The previous evaluations had the same setup as this benchmark beside different keyset sizes. The hypergraphs have from 2 to 7 components and the vertices factors are in a range from $c(r) + 0.1$ to $c(r) + 1.5$, with a step size of 0.1, $c(r)$ is the minimum vertices factors function. One population consists of a set of random initial seeds and different random generated keyset categorized by their size (n), multiple populations are used and different hypergraph types (r) use different populations. All keyset shape are used in this benchmark.

One population has 160 keysets per keyset size and 10000 random intial seeds. The $r = 2$ r-uniform r-partite hypergraph used only one population because a previous evaluation showed that the $r = 2$ could not compete with the other r values. Also the minimal c value $c(2) = 2.0$ is much higher in comparison to the other values. This disqualifies the $r = 2$ r-uniform r-partite hypergraph from the evaluation. All other r values used 3 populations.

The benchmark for one population takes each single keyset from the population and goes through all vertices factors (c) and measures with all initial seeds from the population. Each measurement is counted and categorized by the keyset size (n), the number of components of the hypergraph (r) and vertices factor (c). The multiple results of multiple populations from one hypergraph type (r) are united. The number of r-uniform r-partite hypergraphs to construct until success are summarized with the $x_{n,r,c,0.995}$ quantile.

The approximate heuristic function `optimalR(n)` is developed first. The target is to create a function that returns for all keyset sizes ($n \in \mathbb{N}$) the optimal number of components in the r-uniform r-partite hypergraph (r). The evaluation of the punctual results defined by sequence of keyset sizes $a_n = (10, 15, a_{n-2} * 2, a_{n-2} * 2, \ldots, 1280)$ is done first. The evaluation minimizes the theoretical $\mathcal{O}$ time complexity of the mapping step with the following cost function:

$$n \cdot r \cdot x_{n,r,c,0.995}$$

So the best number of components in the r-uniform r-partite hypergraph (r) is chosen by fixating each punctual keyset size (n) and minimizing the theoretical $\mathcal{O}$ time complexity cost function. The vertices factors (c) are relevant for the evaluation of the cost function but not for the resulting approximate `optimalR(n)` heuristic function.

The punctual domain of the approximate `optimalR(n)` function is extended for all $n \in \mathbb{N}$, by transforming the points to intervals. The evaluation of the individual numbers of components in the r-uniform r-partite hypergraph (r) show a monotone property of the results. The result for all values of r, c and n stays the same or gets better when increasing the n, above a small c value to eliminate fluctuations. With the knowledge of the monotone property the punctual domain is extended to intervals without a increasing of the theoretical $\mathcal{O}$ time complexity costs. Each punctual result is therefore extended to the next higher point. The number of components value for each interval comes from the extended point. The highest point is transformed to infinity and the lowest point is also extended to $n = 1$.

The approximate `optimalR(n)` function specifies the number of components in the r-uniform r-partite hypergraph for the intervals. The target of the approximate `optimalC(n)` function is to return for all intervals the optimal vertices factor. The neglected values of the vertices factor (c) during the construction of the approximate `optimalR(n)` function are now relevant. The vertices factors (c) are obtained by creating a linear interpolation for every interval, using the start and end vertices factors. The vertices factor value for the highest interval is constant.

The optimal mapping benchmark uses the two approximate heuristic functions, thus the variation in the number of components (r) and the number of vertices factor (c) is not needed. This gives space for more keyset sizes (n) and bigger populations. This benchmarks transforms the approximate versions of the two heuristic functions to the final heuristic functions. This benchmark also evaluates the final heuristic functions and uses therefore an increased number of different keyset sizes.

The keyset sizes (n) are $\{2, 3, 4, \ldots, 38, 39, 44, 49, \ldots, 239, 240, 259, 279, 299, \ldots, 1279, 1280\}$, the gaps have a step size of 1, 5 and 20. One population consists of a set of random initial seeds and different random generated keysets categorized by their size (n). The benchmarks uses multiple populations. One poulation has 560 keysets per keyset size and 20000 random intial seeds. This benchmark used 5 populations. All keyset shape are used in this benchmark.

The benchmark for one population takes each single keyset from the population and uses all initial seeds form the population. Each measurement is counted and categorized by the keyset size (n), all results from the populations are united and summarized by the $x_{n,0.995}$ quantile and the maximum.

The two approximate heuristic functions will repeatedly altered and measured, until the $x_{n,0.995}$ quantile stabilizes at the values obtained by the mapping benchmark.

The all seeds mapping benchmark uses multiple populations consisting of one random keyset and all possible seeds in the range from 1 to $2^{31} - 2$. The

tested keysets have the following sizes (n): $9, 29, 49, 69, 89, 109, 129$. The benchmark for one population takes the keyset and counts the number of r-uniform r-partite hypergraphs to construct until it is acyclic for all seeds. Due to the time complexity only one keyset shape with short key names was used.

All results where obtained with the commit `9d397a5a3afb8cc6e84f76f7143e4de672182e6d` of Elektra's git repository[10] and the initial seeds from the rawdata repository[11].

5.4.2 Results

The two heuristic functions are developed by minimizing the theoretical $\mathcal{O}$ time complexity of the mapping step with the following cost function:

$$n \cdot r \cdot x_{n,r,c,0.995}$$

The minimization used the result from the mapping benchmark. Figure 6 depicts the result for $r = 3$. In this result all constructed r-uniform r-partite hypergraph have three components.

Figure 6: Result: Mapping Benchmark with $r = 3$

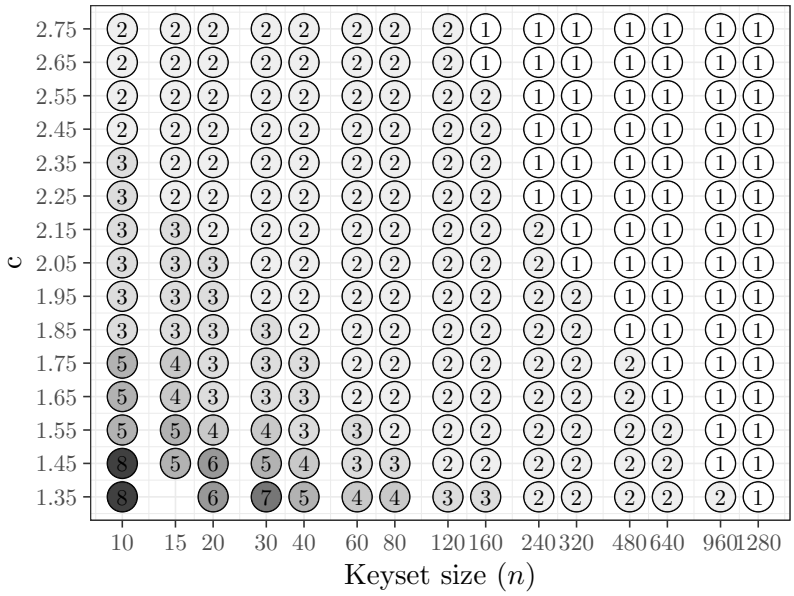


Figure 6 shows how many r-uniform r-partite hypergraphs need to be constructed until an acyclic hypergraph is found. The x-axis is the keyset size. The y-axis is the vertices factor in the r-uniform r-partite hypergraph (c). The dots are the number of r-uniform r-partite hypergraphs to construct until an acyclic hypergraph is found. Each dot is the $x_{n,3,c,0.995}$ quantile of $4,800,000$ measurements. The missing point is an inaccurate measurement, where at least one measurement reached the implementation limit of 10 tries to find an acyclic

[10] git.libelektra.org
[11] https://github.com/ElektraInitiative/rawdata/tree/master/OPMPHM

r-uniform r-partite hypergraph. The other results (also the disastrous $r = 2$) are in the appendix section A.

The two resulting heuristic functions are:

$$optimalR(n) = \begin{cases} 6 & \text{if } 1 \leq n < 15 \\ 5 & \text{if } 15 \leq n < 30 \\ 4 & \text{if } 30 \leq n < 240 \\ 3 & \text{otherwise} \end{cases}$$

$$optimalC(n) = \begin{cases} 3 & \text{if } 1 \leq n < 15 \\ 2.45 \rightarrow 1.95 & \text{if } 15 \leq n < 30 \\ 2.35 \rightarrow 1.45 & \text{if } 30 \leq n < 240 \\ 2.25 \rightarrow 1.35 & \text{if } 240 \leq n < 1280 \\ 1.35 & \text{otherwise} \end{cases}$$

The $\rightarrow$ represents the endpoints of a linear interpolation and for example in the second case the linear interpolation starts at 2.45 for $n = 15$ and goes to 1.95 for $n = 29$.

The two heuristic functions are used and where iteratively measured and improved by the optimal mapping benchmark. Figure 7 display the final result of the optimal mapping benchmark.

Figure 7: Result: Optimal Mapping Benchmark

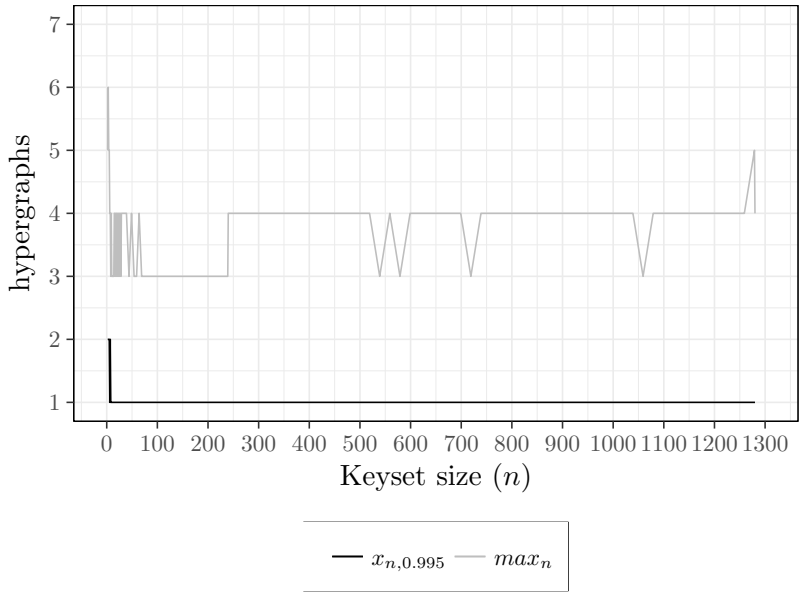


Figure 7 shows in detail how many r-uniform r-partite hypergraphs need to be constructed until an acyclic hypergraph is found. The x-axis is the keyset size.

The y-axis is the number of r-uniform r-partite hypergraphs to construct until an acyclic hypergraph is found. The black line is the $x_{n,0.995}$ quantile and the gray line is the max_n maximum of $56,000,000$ measurements.

All the results until now had random initial seeds the result in figure 8 is obtained by using all possible initial seeds.

Figure 8: Result: All Seeds Mapping Benchmark

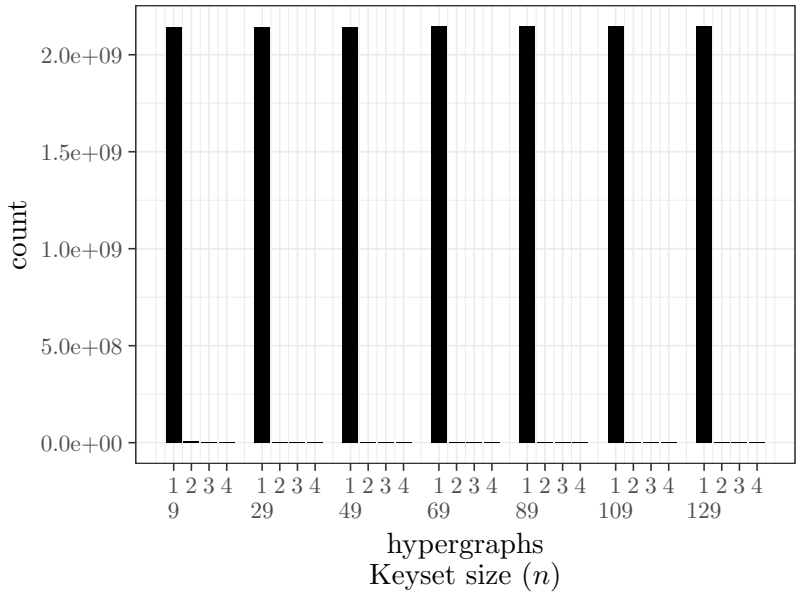


Figure 8 shows the count of how how many r-uniform r-partite hypergraphs need to be constructed until an acyclic hypergraph is found. The first row of the x-axis is the number of r-uniform r-partite hypergraphs to construct until the hypergraph is acyclic. The second row of the x-axis groups the number of r-uniform r-partite hypergraphs to construct by their keyset size. The y-axis is the count. Each keyset size result is obtained from only one keyset.

5.4.3 Discussion

All mapping benchmark results except the disastrous result for $r = 2$ (figures 6,16,17,18 and 19) show the monotone property above a the vertices factor c of $c(r) + 0.5$. The disastrous mapping benchmark result with $= 2$ in figure 15 displays even an inverted monotone property. The results stay the same or get worse when increasing the keyset size.

The mapping benchmark and the optimal mapping benchmark have a maximal keyset size of 1280. The mapping benchmark result with $r = 3$ in figure 6 comes close to the 1280 limit because $x_{1280,3,1.35,0.995} = 1$ and $x_{960,3,1.35,0.995} = 2$. The mapping benchmark results in the appendix section A (figures 16,17,18 and 19) show a much earlier encounter of the minimum $x_{n,r,c,0.995} = 1$ quantile. But with the monotone property it is sufficient for the minimization of the theoretical $\mathcal{O}$ time complexity of the mapping step.

The minimization of the theoretical $\mathcal{O}$ time complexity of the mapping step with the cost function lead to a minimization of the $x_{n,r,c,0.995}$ quantile. Both heuristic functions `optimalR(n)` and `optimalC(n)` return only values that lead to a $x_{n,r,c,0.995}$ quantile that is 1. Therefore the minimization of the theoretical $\mathcal{O}$ time complexity of the mapping step implicates a minimization of the number of r-uniform r-partite hypergraphs to construct until the hypergraph is acyclic.

The optimal mapping benchmark result in figure 7 confirms that the $x_{n,0.995}$ quantile is always except for really small keyset sizes 1. Also for the keyset sizes that where not measured with the mapping benchmark. The maximum max_n depicts that there are bad initial seeds that can take on average up to 4 r-uniform r-partite hypergraphs to construct until an acyclic one is found. For small keyset sizes these bad initial seeds are not perceptible in the runtime of the OPMPHM build. However an increasing keyset size will make bad initial seeds perceptible in the runtime. The $x_{n,0.995} = 1$ result of the optimal mapping benchmark states that the two heuristic functions `optimalR(n)` and `optimalC(n)` return in 99.5% of the cases optimal values.

The all seeds mapping benchmark results in figure 8 indicate that there are not many bad initial seeds. The result is obtained from only one keyset per keyset size and is therefore only a blink to the distribution of the number of r-uniform r-partite hypergraphs to construct until the hypergraph is acyclic. The distribution shows that the most initial seeds lead to only one r-uniform r-partite hypergraphs to construction.

5.5 OPMPHM Build vs Hsearch Build

Elektra's OPMPHM build time is compared with the hsearch build time to answer the second research question.

RQ (ii) How is the time performance of Elektra's OPMPHM build in comparison to the hsearch[12] build?

5.5.1 Method

This benchmark measures the build time of the OPMPHM and the hsearch. The keyset sizes start at 50 and increase to 19550 in steps of 500. For each keyset size 5 different keysets are used. Each measurement was repeated 7 times.

Previous evaluations had shown that hsearch has a problem with too long key names, thus the keysets come from a keyset shape that produces only short names. This does not affect the comparison, since both get same shaped keysets. It is not in the scope of this thesis to find out why hsearch has a problem with too long key names. All data is aggregated by the median ($x_{0.5}$).

The OPMPHM has in this benchmarks a fixed pool of 51 random initial seeds. For every keyset and initial seed the build time was measured. The data is first aggregated by the repeats, then by the initial seeds and at last by the keysets.

The hsearch data is aggregated by the repeats and at last by the keysets. The hsearch build benchmark measures different loads (α) $0.25, 0.5, 0.75, 1$.

The result of both is one value successive aggregated per keyset size. This benchmark is executed on all systems.

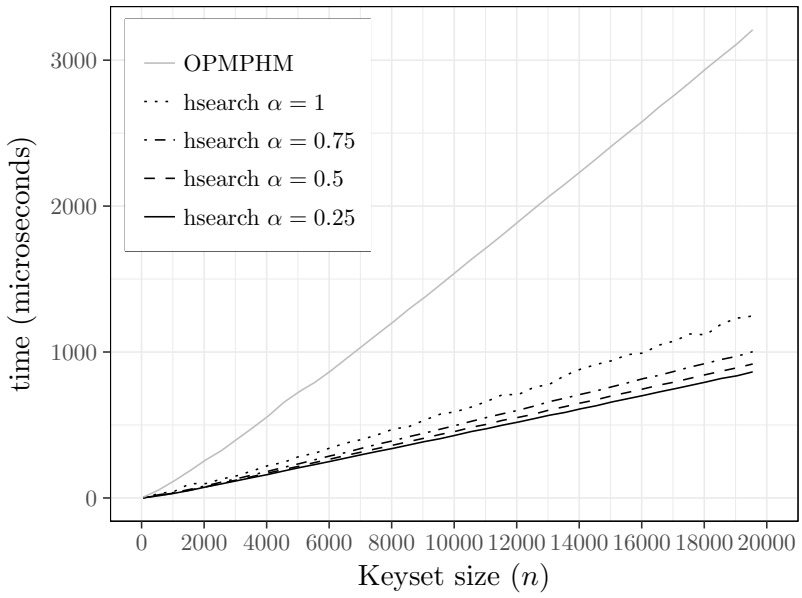
[12]Standard C library function http://linux.die.net/man/3/hsearch

All results where obtained with the commit `9d397a5a3afb8cc6e84f76f7143e4de672182e6d` of Elektra's git repository[13] and the initial seeds from the rawdata repository[14].

5.5.2 Results

The chart in figure 9 shows the result of the comparison between Elektra's OPMPHM build and the hsearch build. This result is from the i7-6700K hardware the result from the other hardwares is in the appendix section B.

Figure 9: Result: OPMPHM vs Hsearch on i7-6700K



The x-axis is the keyset size. The y-axis is the time of the build in microseconds. The gray line is the OPMPHM build time and the black line from dotted to solid are the hsearch build times with the different loads.

5.5.3 Discussion

The results depict that the OPMPHM build time is linear, this corresponds to the theoretical time complexity. The keyset size was extended to 29550 for the OPMPHM on the i7-3517U hardware (figure 20). Because the range from 50 to 19550 indicated a non linear runtime.

The results made on the i7-6700K and 1800X hardware (figures 9 and 21) show that the hsearch build time increases with the load. This result was expected, though the results made on the i7-3517U hardware does not conform with that expectation.

Comparing the worst hsearch build time with the OPMPHM build time shows that the hsearch build is on average $\approx 60\%$ faster than the OPMPHM

[13] git.libelektra.org
[14] https://github.com/ElektraInitiative/rawdata/tree/master/OPMPHM

build. The hsearch build saves on average $\approx 60\%$ of the time compared to the OPMPHM build.

5.6 The Hybrid Search

The performance of the hybrid search relies on:

heuristic function that determines if a sequence of k searches between two keyset alterations (keyset size n) is or is not worth using the OPMPHM.

history register length how many past events are remembered by the hybrid search.

action limit that defines the minimum keyset size for the hybrid search usage.

The task for the third research question splits in two parts. The first part is finding a heuristic function and the second part is the evaluation of the hybrid search with the heuristic function and different settings. The actual values for the history register length and the action limit result from this evaluation.

RQ (iii) How much superior and how much inferior is the hybrid search time compared to the standalone binary search time in random cases?

5.6.1 Method

The target of the heuristic function is to define how many searches (k) are necessary that the OPMPHM is faster than the binary search. Since the binary search time and the OPMPHM build time depend on the keyset size the heuristic function also depends on the keyset size. The benchmark measures for a keyset size n the time spent to make k searches with the binary search and the time spent to make k searches with the OPMPHM including the OPMPHM build time. The first k where the OPMPHM usage time is fast than the binary search time defines the heuristic function. The measurement was made with all keyset shapes except one that had unnaturally long key names. Because previous evaluation had show that the results with that keyset shape where unusable.

The OPMPHM search time consists of build time and search time. Due to time complexity reasons the benchmark was split up in build time and search time. The OPMPHM build time benchmark has the same settings as in the OPMPHM compare with hsearch.

Both search time benchmarks used a keyset size that starts at 50 and increase to 19550 in steps of 500. For each keyset size 3 different keysets are used, the measurements where repeated 7 times. The data was aggregated first by the repeats and then by the keysets always with the median ($x_{0.5}$). The different keyset shapes are then aggregated by the mean. The number of searches starts at 500 and increases to 32000 in steps of 500. Each search was made randomly. For both the result is a matrix where one dimension is the keyset size the other the number of searches. The OPMPHM build time was added to the OPMPHM search time. The heuristic function is created by going through all keyset sizes (n) and find that number of searches (k) where the OPMPHM search time is smaller than the binary search time. Through this result the heuristic function defines what sequence of search is or is not worth using the OPMPHM.

The final benchmark evaluates the hybrid search, by measuring the time of random patterns. A pattern consists of 66 random sequences, each sequence has a random number of searches. In between the sequences an alteration is simulated, by setting the `KS_FLAG_NAME_CHANGE` flag of the keyset and by clearing the OPMPHM. Each sequence has a random length between 1 and twice the heuristic function. Therefore the random number of searches are half in the not worth using the OPMPHM area and half in the worth using the OPMPHM area of the heuristic function.

The benchmarked keyset sizes are defined by the series $a_n = (100, 200, \ldots, 1000, 1200, \ldots, 5000, 6000, \ldots, 10000)$ and history register lengths from 5 to 11 bits in steps of 2. The i7-6700K system used 999 pattern per keyset size, the other systems use 503. The number of patterns for the other systems must be restricted to reduce the runtime of the final benchmark. The benchmark rotates through all keyset shapes, except the one with the unnaturally long key names. The hybrid search time and the standalone binary search time is measured. During the measurement all searches were random and each measurement is repeated 5 time and the median ($x_{0.5}$) is taken. The OPMPHM if used, used for each pattern different 66 random initial seeds. The two evaluation are made for every history register length individually.

For each keyset size the measurements are divided in two sets where the hybrid search time is faster or the standalone binary search is faster. The first evaluation calculates a ratio out of the cardinality of the two sets. The ratio represents the percentage of measurements where the hybrid search is faster than the standalone binary search. The second one evaluates each sets individually, by calculating for each element the percentage of how much the hybrid search time is faster or how much the standalone binary search time is faster. All the resulting percentages are then aggregated by the mean. This calculation represents the average percentage how much superior and how much inferior is the hybrid search time compared to the standalone binary search time.

The results of the first part where obtained with the commit `9d397a5a3afb8cc6e84f76f7143e4de672182e6d`. The results of the second part where obtained with the commit `3858e2dd02738cec84c9c10cec3901978246fecf`. Both of Elektra's git repository[15]. The initial seeds are from the rawdata repository[16].

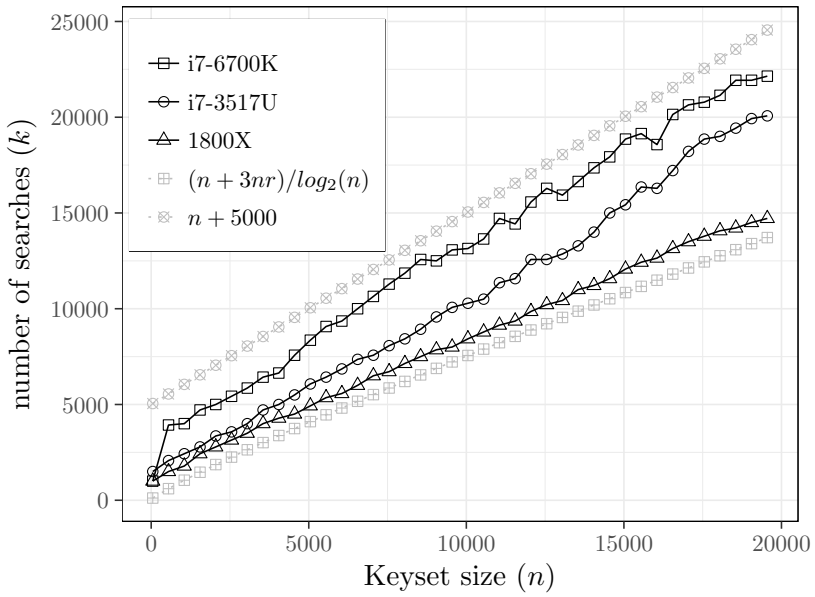
5.6.2 Results

The results from the first part are used to find the heuristic function. Figure 10 depicts how many searches are necessary to make the OPMPHM usage worth.

[15] git.libelektra.org

[16] https://github.com/ElektraInitiative/rawdata/tree/master/OPMPHM

Figure 10: Result: Heuristic Function



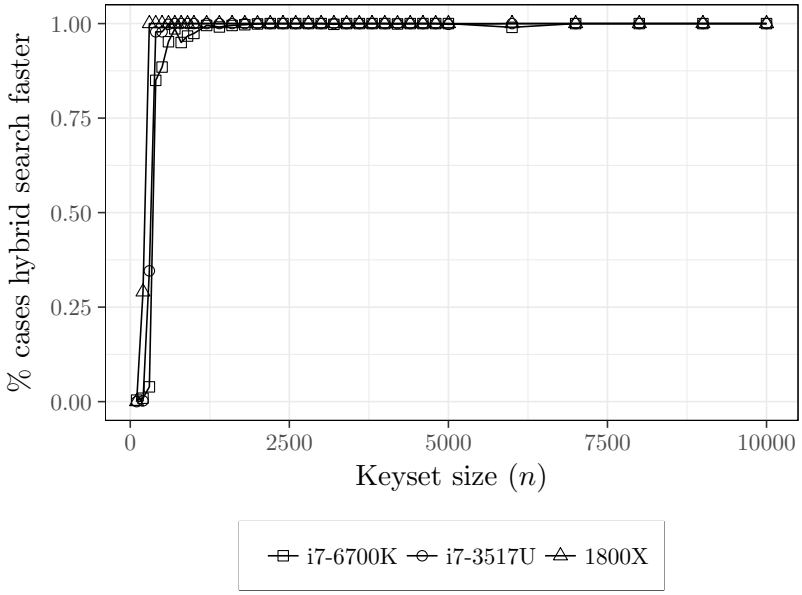
The x-axis is the keyset size and the y-axis is the number of searches. The black lines are the results of the different hardwares and show above what number of searches the OPMPHM usage is faster than the binary search usage. For example with a keyset size of 10000 and on the i7-6700K hardware are $\approx$ 12500 searches needed to make the OPMPHM usage worth. The function $(n + 3nr)/log_2(n)$ is the theoretical heuristic function resulting from both theoretical $\mathcal{O}$ time complexities. The final heuristic function is:

$$n + 5000 = h(n) > k$$

The function $n + 5000$ has the advantages that is fast computable and does not underestimate the measured values. This heuristic function determines how many searches (k) are necessary to make the OPMPHM usage worth. When the number of searches is bigger than $h(n)$ it is worth using the OPMPHM.

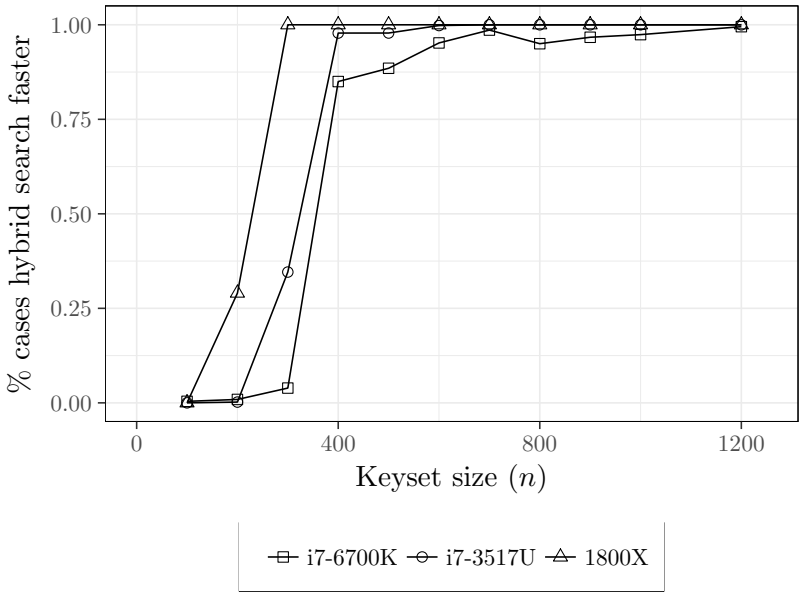
The found heuristic function is used to obtain the final results. Figure 11 shows the percentage of random cases where the hybrid search with a 9 bit history register length is faster than the standalone binary search.

Figure 11: Result: Cases Hybrid Search faster with 9 Bit History Register



The x-axis is the keyset size and the y-axis is the percentage of random cases were the hybrid search is faster than the stand alone binary search. The three black lines are the different hardwares. Figure 12 is a enlargement of the keyset range from 0 to 1200.

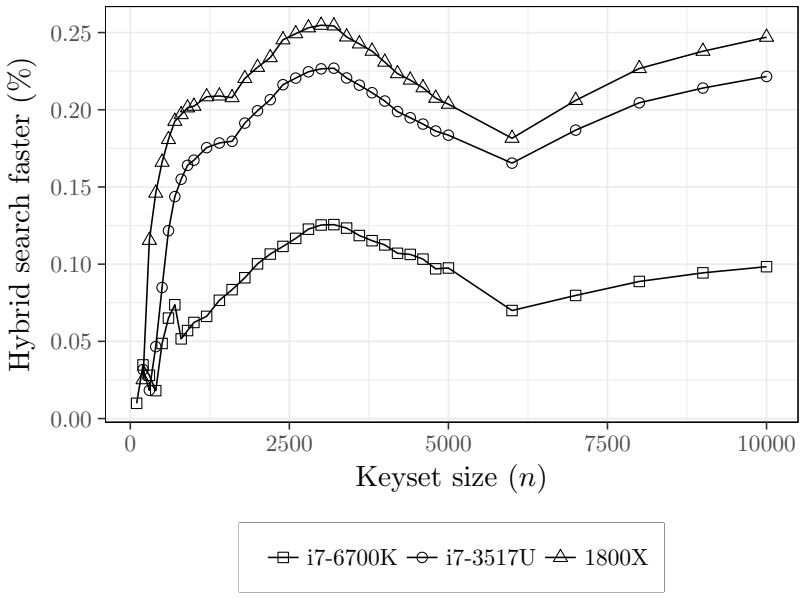
Figure 12: Result: Cases Hybrid Search faster with 9 Bit History Register, Enlarged



Here also x-axis is the keyset size and the y-axis is the percentage of random cases were the hybrid search is faster than the standalone binary search. The three black lines are the different hardwares. The other history register lengths are in the appendix section C, subsection cases hybrid search faster.

Figure 13 examines the random cases where the hybrid search is faster than the standalone binary search. This figure exhibits the average percentage how much faster is the hybrid search compared to the standalone binary search.

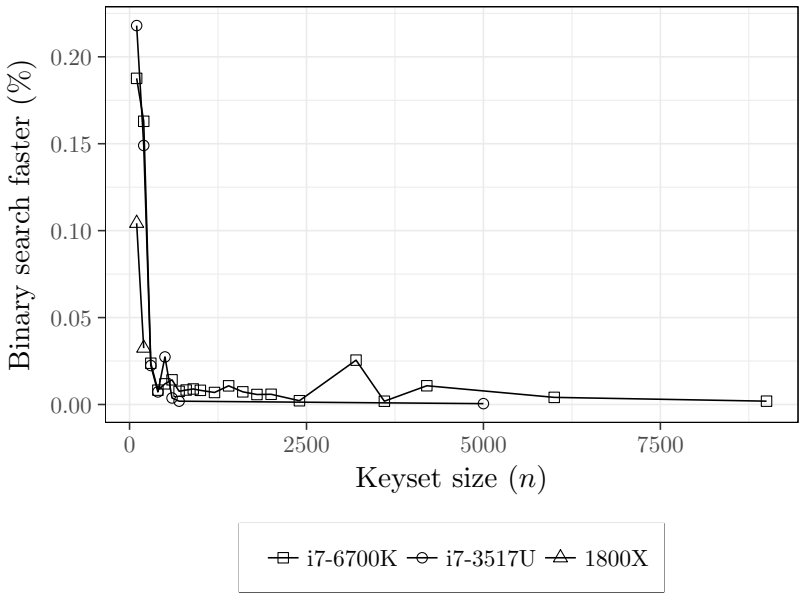
Figure 13: Result: Hybrid Search Superior 9 Bit History Register



The x-axis is the keyset size. The y-axis is the average percentage how much faster the hybrid search is compared to the standalone binary search. The black lines represent the different hardwares. For example with a keyset size of 10000 and on the i7-6700K hardware the hybrid search is $\approx 10\%$ faster than the standalone binary search. In this example the hybrid search saves on average $\approx 10\%$ of the time compared to the standalone binary search.

Figure 14 examines only the random cases where the standalone binary search is faster than the hybrid search.

Figure 14: Result: Hybrid Search Inferior 9 Bit History Register



The x-axis is the keyset size. The y-axis is the average percentage how much faster the standalone binary search is compared to the hybrid search. The black lines represent the different hardwares. There are missing points in this figure, these keyset sizes had no inferior random cases. For example with a keyset size of 3100 and on the i7-6700K hardware the standalone binary search is $\approx 2.5\%$ faster than the hybrid search. In this example the standalone binary search saves on average $\approx 2.5\%$ of the time compared to the hybrid search. The other superior and inferior results for the other history register lengths are in the appendix section C, subsection hybrid search superior and inferior results.

5.6.3 Discussion

Figure 10 shows that each hardware has another number of search necessary to make the OPMPHM usage worth. The 1800X hardware is really close and the i7-6700K is far away from the theoretical heuristic function. This made the choice for the good heuristic function difficult. Choosing a heuristic function that overestimates the real costs is choosing the lesser evil. Since both over- and underestimation of the real costs by the heuristic function is not optimal, but the underestimation is much worst than the overestimation. Since the OPMPHM would be used in cases where the binary search is faster. Most of the OPMPHM usage time is spent in the build, since the search works in $\mathcal{O}(1)$. Compared to a small number of searches with the binary search the OPMPHM build is much more expensive. Therefore it is saver to build the OPMPHM only in cases where it is ensured that it is profitable.

The cases hybrid search faster results (figures 22,11,23 and 24) depict that the number of cases where the hybrid search is faster grow when the history register length is increasing. The hybrid search superior results (figures 25,13,26 and 27) show also a general improvement of the hybrid search when the history

register length is increasing. Even the hybrid search inferior results (figures 28,14,29 and 30) exhibit that the standalone binary search is getting slower when the history register length is increasing. Thus the performance of the hybrid search is increasing with the length of the history register. Though a long history register length comes at high costs, since the space complexity of the modified branch predictor is exponential in the history register length.

The history register length of 9 bit represents a good tradeoff between performance and memory consumption. The enhancements of the results from 9 bit history register length to 11 bit history register length are not big and the memory consumption is with a 9 bit history register length only 128 byte compared to the 11 bit history register length with 512 byte.

The cases hybrid search faster result in figure 11 show that except for small keyset sizes the hybrid search is in almost all random cases faster than the standalone binary search. The enlarged cases hybrid search faster result in figure 12 shows that above a keyset size of 599 the percentage of random cases where the hybrid search is faster grows over 90%. This makes 599 to a good action limit, that defines the minimum keyset size for the hybrid search usage.

The enlarged cases hybrid search faster result in figure 12 depicts also that the hybrid search is weak at a small keyset size. The hybrid search inferior results in figure 14 confirm that at small keyset sizes performance is lost.

An average calculation over the superior results shows that the hybrid search is on average $\approx 8.53\%$ to $\approx 20.92\%$ faster than the standalone binary search. Depending on the hardware $\approx 8.53\%$ to $\approx 20.92\%$ of the time is saved by using the hybrid search.

An average calculation over the inferior results shows that the standalone binary search is on average $\approx 2.49\%$ to $\approx 6.83\%$ faster than the hybrid search. Depending on the hardware $\approx 2.49\%$ to $\approx 6.83\%$ of the time is saved by using the standalone binary search.

6 Conclusion

The goal of this thesis was the extension of Elektra's search operation with an OPMPHM, but without any change of the Elektra API. In this thesis three heuristic functions where developed and measured to avoid any change of the Elektra API and to gain performance. Two of them have the target to minimize the OPMPHM build runtime. The first research question evaluates there quality:

RQ (i) Is the runtime of the OPMPHM build minimal?

This question was evaluated with benchmarks that used random cases. The results had shown that the runtime of the OPMPHM build is in 99.5% of the random cases minimal. Since the OPMPHM algorithm is a randomized algorithm there is no guarantee that the runtime of the OPMPHM build in all cases minimal.

The evaluation of the second research question, showed how the OPMPHM performs compared to a common hash map.

RQ (ii) How is the time performance of Elektra's OPMPHM build in comparison to the hsearch[17] build?

[17]Standard C library function http://linux.die.net/man/3/hsearch

The time performance of Elektra's OPMPHM build in comparison to the hsearch build is bad, the hsearch build is on average $\approx 60\%$ faster than the OPMPHM build and therefore saves $\approx 60\%$ of the time.

The third heuristic function is used by the hybrid search, the aim of the hybrid search is to achieve the goal of this thesis. The third research question evaluated the hybrid search and therefore all the implemented components.

RQ (iii) How much superior and how much inferior is the hybrid search time compared to the standalone binary search time in random cases?

The results showed that the percentage of random cases where the hybrid search is faster than the standalone binary search is for a keyset sizes larger than 599 almost always 100%. How much superior and how much inferior the hybrid search time is compared to the standalone binary search time strongly depended on the hardware used to measure. In the hybrid search superior random cases the hybrid search is on average from $\approx 8.53\%$ to $\approx 20.92\%$ faster. Using the hybrid search instead of the standalone binary search saved in this cases $\approx 8.53\%$ to $\approx 20.92\%$ of the time. In the hybrid search inferior random cases the standalone binary search time is on average from $\approx 2.49\%$ to $\approx 6.83\%$ faster. Using the standalone binary search instead of the hybrid search saved in this cases $\approx 2.49\%$ to $\approx 6.83\%$ of the time.

6.1 Further Work

The OPMPHM algorithm can only handle a limited keyset size, since the seeded hash function returns only an `uint32_t`. The seeded hash function maps an element to the vertices in each component of G_r. This implicates that one component of G_r has at maximum of $2^{32} - 1$ vertices and one component size is defined as cn/r. The heuristic values from `optimalR(n)` and `optimalC(n)` for a great n placed in the equation $2^{32} - 1 = cn/r$, result in $n = 9544371767$. The 9544371767 represents the limit of the keyset size. This problem can be resolved by replacing the actual seeded hash function with another seeded hash function that returns an `uint64_t`. The replacement forces the reevaluation of the first research question, including the reconstruction of the heuristic functions `optimalR(n)` and `optimalC(n)`.

The general performance on not so powerful hardware could be improved by hard-coding the number of components in the r-uniform r-partite hypergraph r. This would force a choice for an optimal value of r and would make the heuristic function `optimalR(n)` obsolete. The major work of the algorithm is done in loops over r and a hard-coded value would allow the compiler to optimize the code more. This optimization could be combined with the replacement of the modulo operation by the logic *and* operator. The replacement would limit the component sizes of G_r to $2^x - 1$.

The OPMPHM algorithm implements only the mapping step from Botelho's RAM algorithm[2]. The here presented assignment step could be replace with the assigning and ranking step of the RAM algorithm from Botelho[2]. This would lead to a more space efficient OPMPHM.

The heuristic function that determines how many searches are necessary to make the OPMPHM usage worth, is developed and evaluated only on the hardware used in this thesis. The heuristic function could be custom made for every

hardware with an automatic benchmark that runs on the first Elektra execution. Based on the in this thesis developed heuristic function this automatic benchmark could define a custom made heuristic function. The custom made heuristic function could be stored in Elektra.

One search in Elektra can trigger a series of searches. The length of this series of searches is tiny compared to the length of the benchmarked search sequences in this thesis. An investigation could tell more about the performance of the tiny series of searches when done with the OPMPHM.

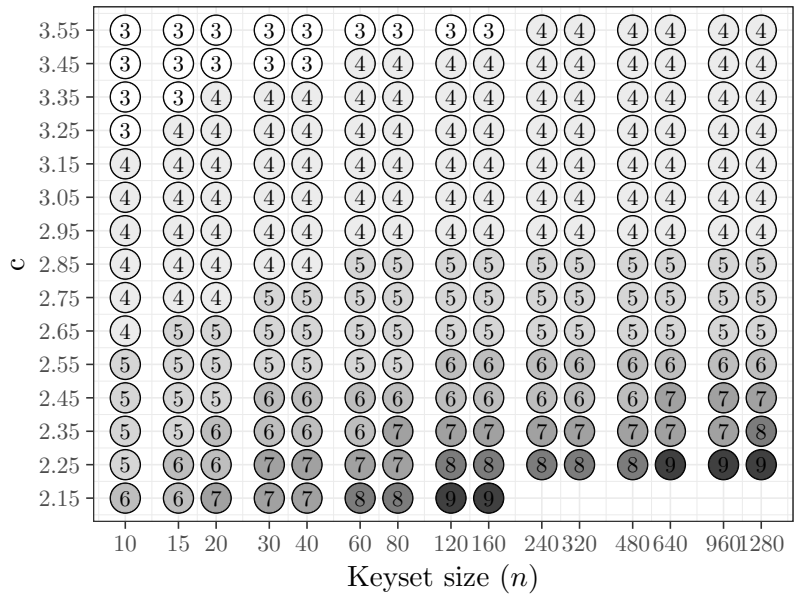
7 Appendix

7.1 Appendix A

All of these charts show the individual number of components in the r-uniform r-partite hypergraph (r) results. They depicts how many r-uniform r-partite hypergraphs need to be constructed until an acyclic hypergraph is found. The x-axis is the keyset size. The y-axis is the number of vertices factor in the r-uniform r-partite hypergraph (c). The missing points are inaccurate measurement, where at least one measurement reached the implementation limit of 10 tries to find an acyclic r-uniform r-partite hypergraph.

This is the disastrous result of $r = 2$:

Figure 15: Result: Mapping Benchmark with $r = 2$



Each dot is the $x_{n,2,c,0.995}$ quantile of $1,600,000$ measurements.

The following chart depict the other number of components in the r-uniform r-partite hypergraph (r) results. Each dot is the $x_{n,r,c,0.995}$ quantile of $4,800,000$ measurements.

Figure 16: Result: Mapping Benchmark with $r = 4$

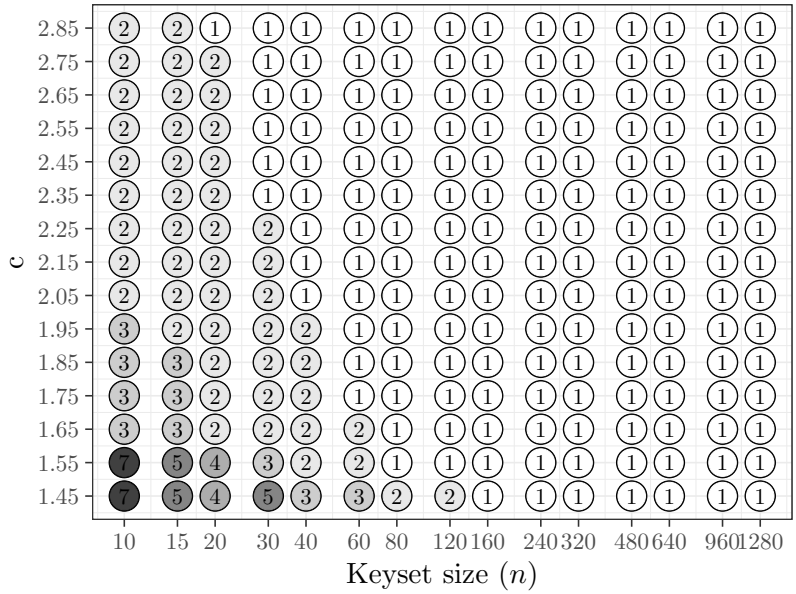


Figure 17: Result: Mapping Benchmark with $r = 5$

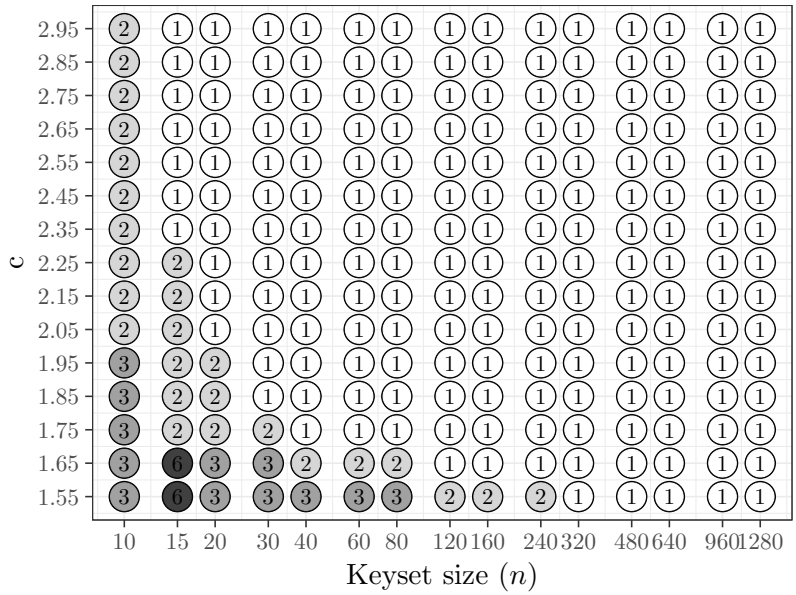


Figure 18: Result: Mapping Benchmark with $r = 6$

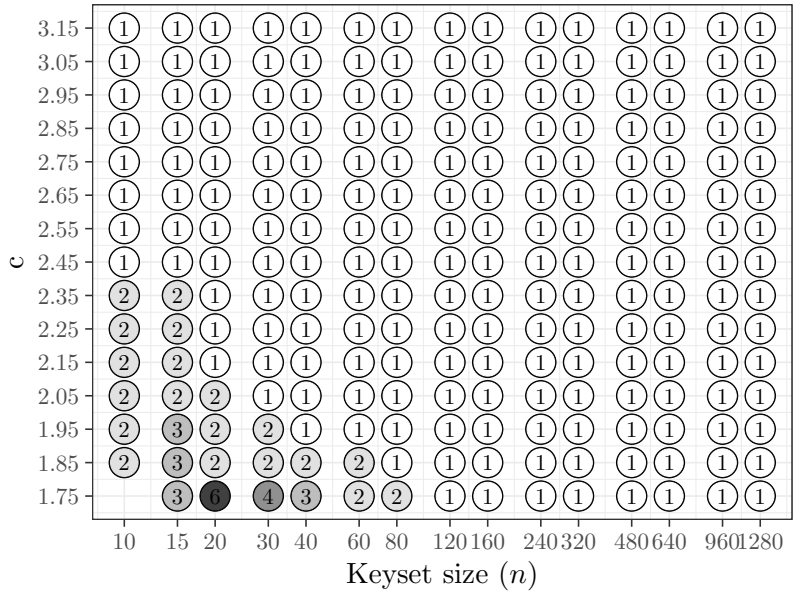
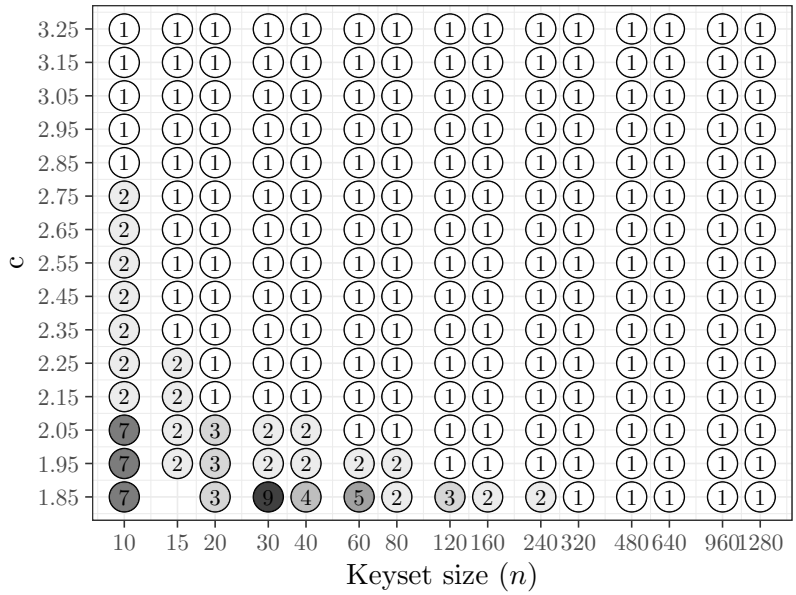


Figure 19: Result: Mapping Benchmark with $r = 7$



7.2 Appendix B

The x-axis is the keyset size. The y-axis is the time of the build in microseconds. The gray line is the OPMPHM build time and the black line from dotted to solid are the hsearch build times with the different loads.

Figure 20: Result: OPMPHM vs Hsearch on i7-3517U

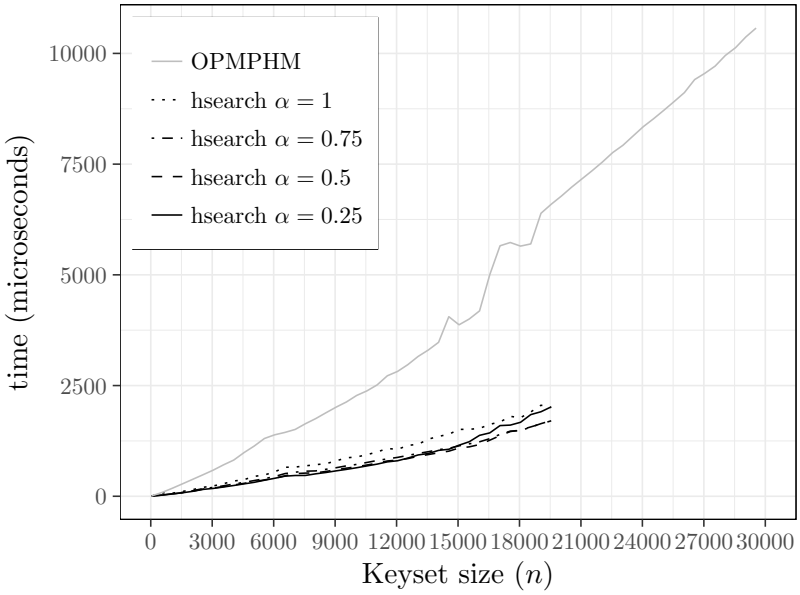
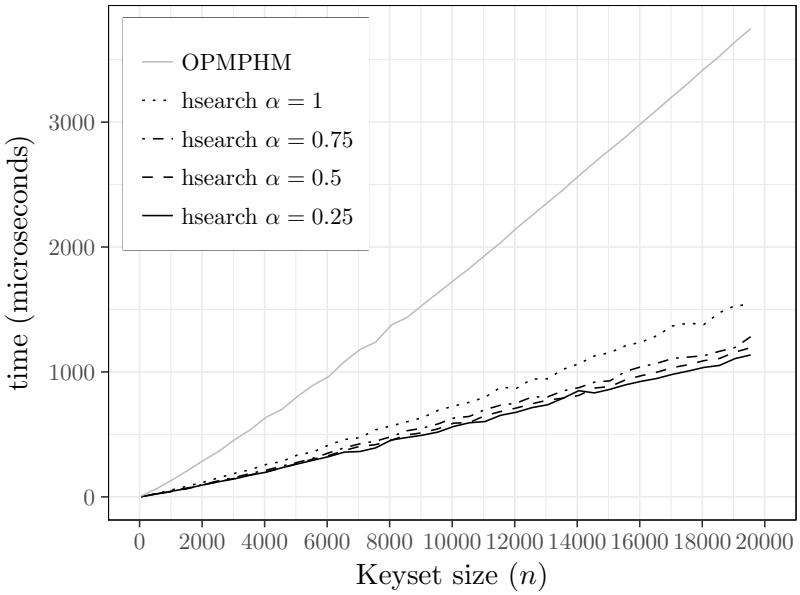


Figure 21: Result: OPMPHM vs Hsearch on 1800X



7.3 Appendix C

7.3.1 Cases Hybrid Search Faster Result

The following charts depict the percentage of random scenarios where the hybrid search is faster than the standalone binary search on different hardware. The x-axis is the keyset size and the y-axis is the percentage of random scenarios were the hybrid search is faster than the stand alone binary search. The three black lines are the different hardwares.

Figure 22: Result: Cases Hybrid Search with 11 Bit History Register

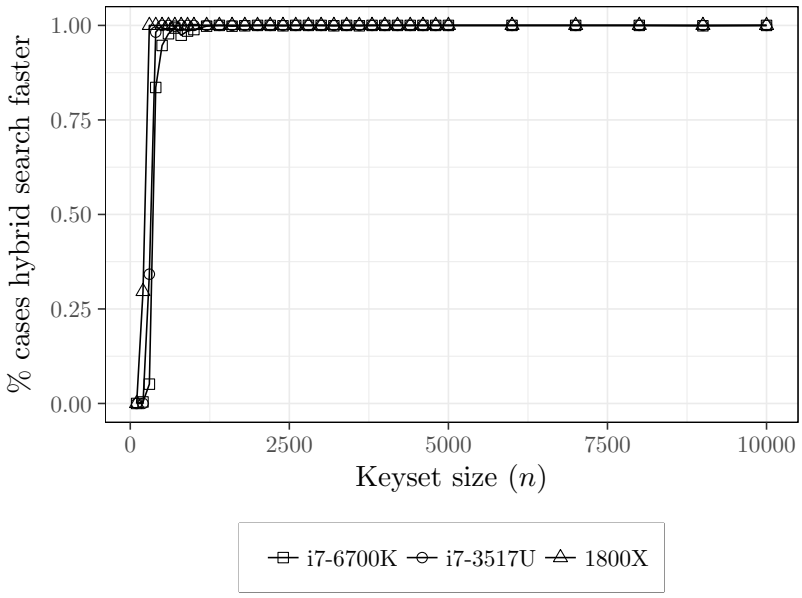


Figure 23: Result: Cases Hybrid Search with 7 Bit History Register

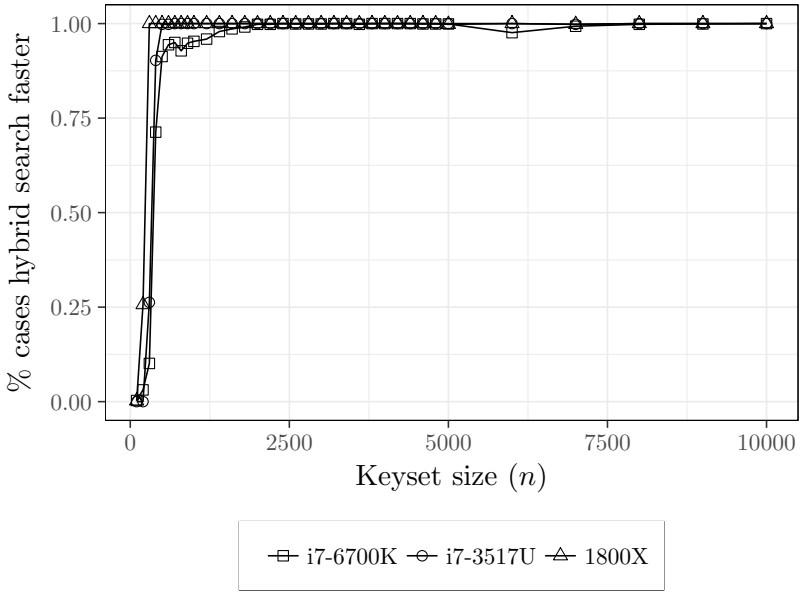
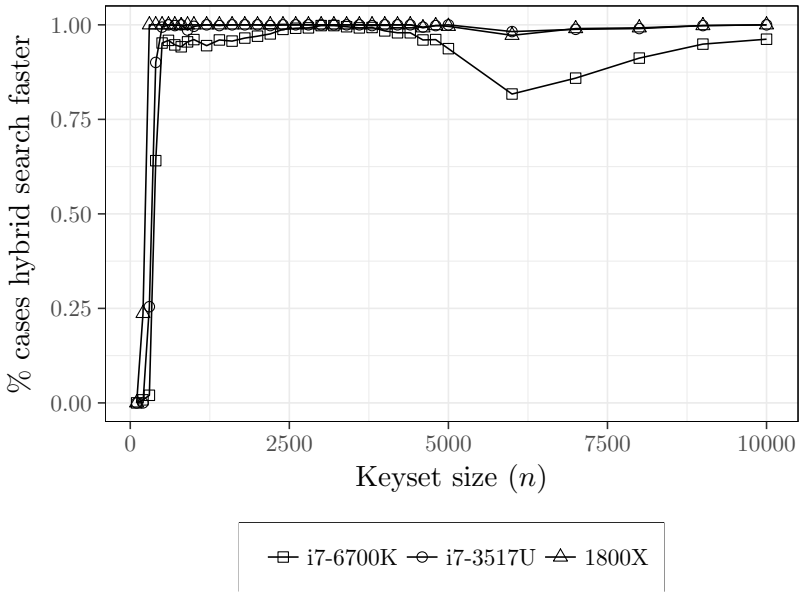


Figure 24: Result: Cases Hybrid Search with 5 Bit History Register



7.3.2 Hybrid Search Superior and Inferior Results

The following charts show the average percentage how much faster is the hybrid search compared to the standalone binary search. The x-axis is the keyset size. The y-axis is the average percentage how much faster the hybrid search is

compared to the standalone binary search in the random scenarios. The black lines represent the different hardwares.

Figure 25: Result: Hybrid Search Superior 11 Bit History Register

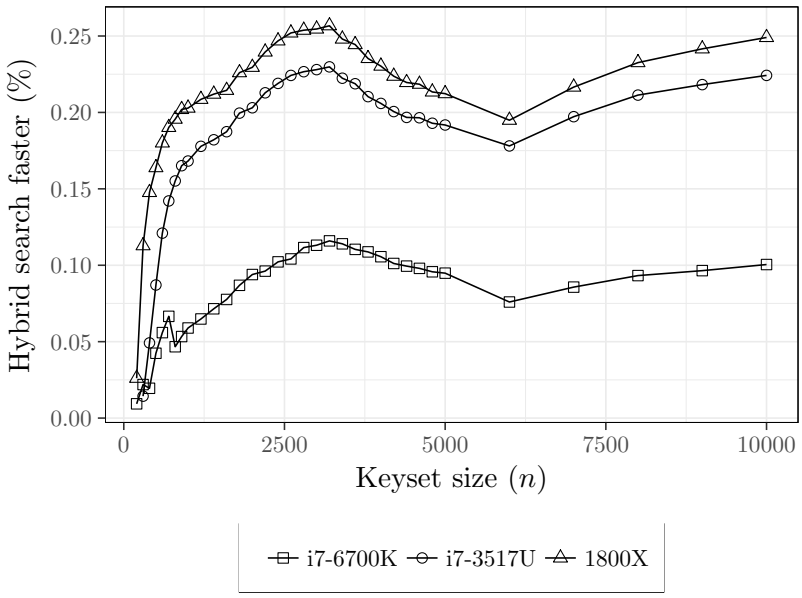


Figure 26: Result: Hybrid Search Superior 7 Bit History Register

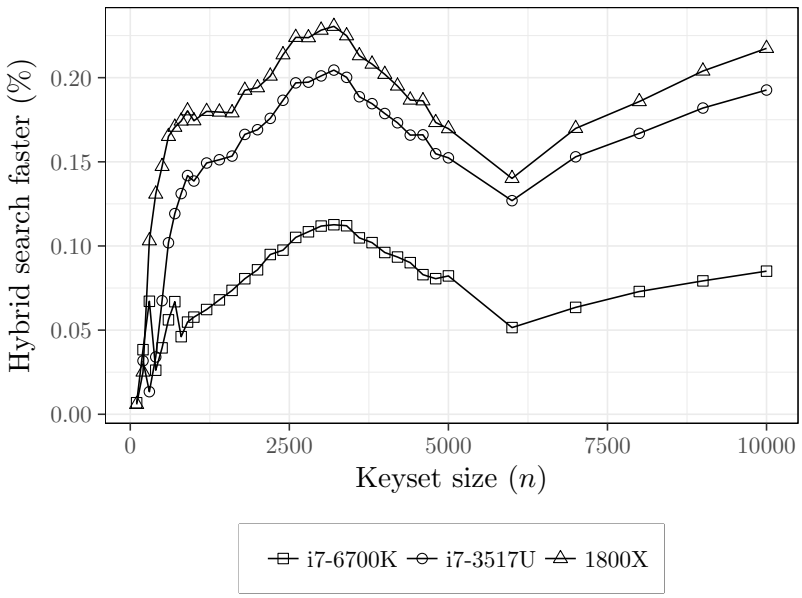
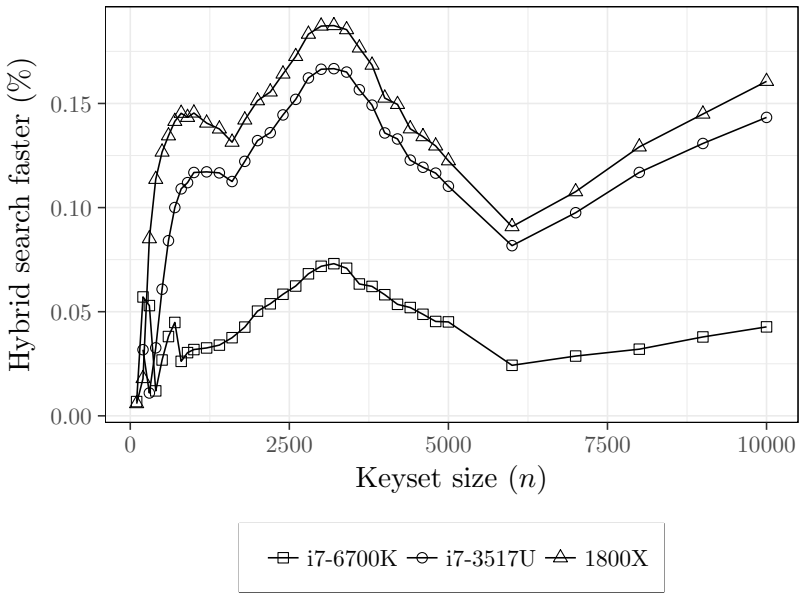


Figure 27: Result: Hybrid Search Superior 5 Bit History Register



The following charts exhibits the average percentage how much faster is the standalone binary search compared to the hybrid search. The x-axis is the keyset size. The y-axis is the average percentage how much faster the hybrid search is compared to the standalone binary search in the random scenarios. The black lines represent the different hardwares. There are missing points in this figures, these keyset sizes had no inferior random cases.

Figure 28: Result: Hybrid Search Inferior 11 Bit History Register

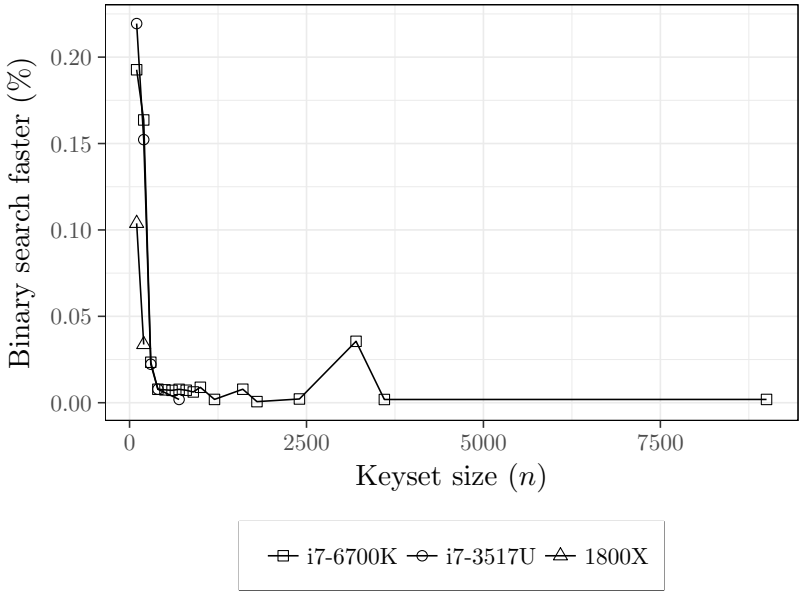


Figure 29: Result: Hybrid Search Inferior 7 Bit History Register

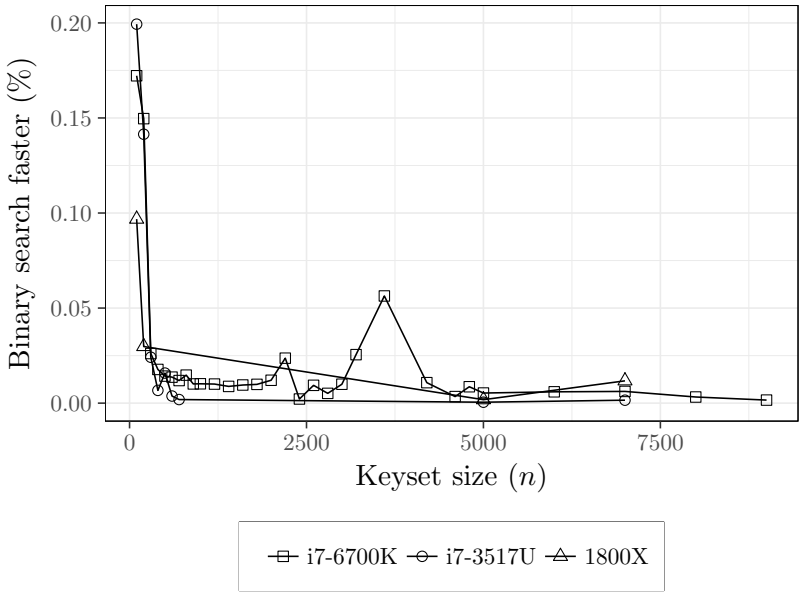
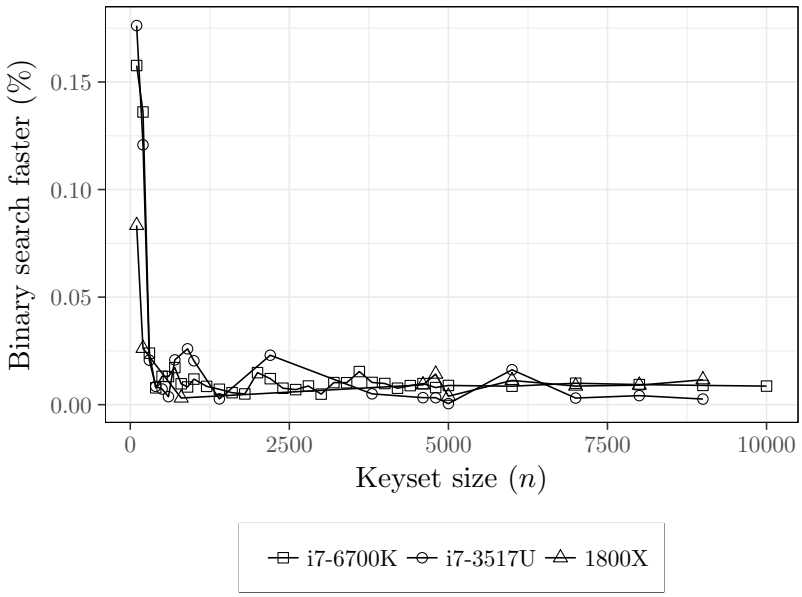


Figure 30: Result: Hybrid Search Inferior 5 Bit History Register



References

[1] Fabiano C. Botelho, David M. Gomes, and Nivio Ziviani. *A New Algorithm for Constructing Minimal Perfect Hash Functions*. Jan. 2008.

[2] Fabiano C. Botelho and Nivio Ziviani. *Near-Optimal Space Perfect Hashing Algorithms*.

[3] David F. Carta. "Two Fast Implementations of the "Minimal Standard" Random Number Generator". In: *Communications ACM* 33.1 (Jan. 1990), pp. 87–88. ISSN: 0001-0782. DOI: 10.1145/76372.76379. URL: http://doi.acm.org/10.1145/76372.76379.

[4] Zbigniew J. Czech, George Havas, and Bohdan S. Majewski. "An Optimal Algorithm for Generating Minimal Perfect Hash Functions". In: *Information Processing Letters* 43 (1992), pp. 257–264.

[5] Zbigniew J. Czech, George Havas, and Bohdan S. Majewski. "Perfect Hashing". In: *Theoretical Computer Science* 182.1-2 (1997), pp. 1–143. DOI: 10.1016/S0304-3975(96)00146-6. URL: https://doi.org/10.1016/S0304-3975(96)00146-6.

[6] Edward A. Fox et al. "Order-preserving Minimal Perfect Hash Functions and Information Retrieval". In: *ACM Transactions on Information Systems* 9.3 (July 1991), pp. 281–308. ISSN: 1046-8188. DOI: 10.1145/125187.125200. URL: http://doi.acm.org/10.1145/125187.125200.

[7] S. K. Park and K. W. Miller. "Random Number Generators: Good Ones Are Hard to Find". In: *Communications ACM* 31.10 (Oct. 1988), pp. 1192–1201. ISSN: 0001-0782. DOI: 10.1145/63039.63042. URL: http://doi.acm.org/10.1145/63039.63042.

[8] Markus Raab. “A modular approach to configuration storage”. In: *Master's thesis, Vienna University of Technology* (2010).

[9] Linus Schrage. “A More Portable Fortran Random Number Generator”. In: *ACM Transactions on Mathematical Software* 5.2 (June 1979), pp. 132–138. ISSN: 0098-3500. DOI: 10.1145/355826.355828. URL: http://doi.acm.org/10.1145/355826.355828.

[10] Tse-Yu Yeh and Yale N. Patt. “Alternative Implementations of Two-level Adaptive Branch Prediction”. In: *Proceedings of the 19th Annual International Symposium on Computer Architecture*. ISCA '92. Queensland, Australia: ACM, 1992, pp. 124–134. ISBN: 0-89791-509-7. DOI: 10.1145/139669.139709. URL: http://doi.acm.org/10.1145/139669.139709.