FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Department of Computer Science 7

Computer Networks and Communication Systems

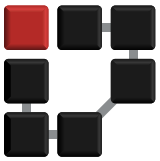
Moritz Schaffenroth

Decision Diagrams for the Efficient Representation of Automotive Configurations

Master's Thesis in Computer Science

24. April 2014

Please cite as:
Moritz Schaffenroth, "Decision Diagrams for the Efficient Representation of Automotive Configurations," Master's Thesis (Masterarbeit), University of Erlangen, Dept. of Computer Science, April 2014.



Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Rechnernetze und Kommunikationssysteme

Martensstr. 3 · 91058 Erlangen · Germany

http://www7.cs.fau.de/

Decision Diagrams for the Efficient Representation of Automotive Configurations

Master's Thesis in Computer Science

22. April 2014

Moritz Schaffenroth

Department of Computer Science 7
Computer Networks and Communication Systems

Friedrich-Alexander-Universität Erlangen-Nürnberg

supervisor: **Rüdiger Berndt**
Peter Bazan
Prof. Dr-Ing. Reinhard German

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties.
I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Moritz Schaffenroth)
Erlangen, 22. April 2014

Abstract

The increasing product variety in the automotive sector makes it harder to check the consistency in product configuration data and therefore increases sources of error. To check correctness of the sets in the product configuration context, MDDs are can be used. In this thesis an insight into the product configuration problems is given and the BDD-tool JINC is adapted to handle MDDs with variable number of branches. Further are *fully-* and *quasi-reduced* MDDs and *sparse representations* implemented and compared. A new branch compression technique that is adapted to the product configuration context is introduced and the advantages are shown. JINC features like parallelism and *multi operand apply* are adapted for MDDs and examined.

Die steigende Produktvielfalt im Automobilbereich erhöht die Schwierigkeit von Konsistenzprüfungen im Bereich der Produktkonfigurationen und damit auch die Anzahl von Fehlerquellen. Um die unterschiedlichen Mengen im Produktkonfigurations-Kontext zu prüfen, können MDDs verwendet werden. In dieser Arbeit wird ein Einblick in Problematik im Zusammenhang mit Produktkonfigurationen gegeben und das BDD-Tool JINC angepasst, damit es MDDs mit varibler Anzahl an Zweigen verarbeiten kann. Des Weiteren werden *voll-* und *quasi-reduzierte* MDDs und *spärliche Repräsentationen* implementiert und verglichen. Eine neue Kompressionstechnik zum Speichern der Zweige, welche für den Produktkonfigurations Bereich geeignet ist wird eingeführt und die Vorteile gezeigt. Unterschiedliche Merkmale der BDD-Bibliothek JINC, wie Parallelisierung und den sogenannten *multi operand apply* Algorithmus werden für MDDs angepasst und deren Leistungsfähigkeit untersucht.

Contents

Chapter 1

Motivation

Times when a Henry Ford could say “any customer can have a car painted any color that he wants so long as it is black” [1] are long gone. Today, customers demand custom products. The automobile became a way to express personal lifestyle, a status symbol or just a means of transportation a long time ago. Ecological awareness, recreation fun, the overcrowding of the city, or the freedom of the wilderness might be possible reasons to choose from a variety of cars. Manufacturers spare no effort to satisfy the customer’s every possible wish. Therefore, innovations are accelerating and automobiles are becoming more and more complex. Factors like national restrictions and new markets also influence the product variety. In the automotive industry the amount of possible product configurations increases constantly [2]. On the other hand, mistakes can delay the launch of new models, lead to defective assembly or delay the assembly because of missing parts. Product variety increases costs and the testing effort on the production side, but it is a deciding factor. To avoid mistakes in the product configuration data, it has to be checked. The number of possible configurations is by far higher than the estimated atom count of the visible universe (see Chapter 6). However there are efficient approaches to store and check valid product configurations based on decision diagrams. One of them is the usage of binary decision diagrams (BDDs) or as an extension multi-valued decision diagrams (MDDss). Decision diagrams are data structures that are able to encode functions compactly which are cross products of finite sets [3]. Donald Knuth is said to have called BDDs “one of the only really fundamental data structures that came out in the last twenty-five years” [4]. They are used in many different domains like logic sythesis [5] formal verification [6] and computer-aided design (CAD)

applications [7]. There are already commercial products that use BDDs in the field of product configurations [8]. The approach used here has similarities, but uses MDDs to represent the solution space. It is based on previous work by [9] and [9]. An initial compiling process is applied on given constraints to represent the configurations with MDDss compactly. Afterwards quick access to the resulting data structure results in a fast check time. To compile MDDs, the optimized BDD-tool JINC [10] was chosen. It is the product of the dissertation of Jörn Ossowski from the Friedrich-Wilhelms University of Bonn in 2009. The stated features are: It contains a framework for *higher-order weighted decision diagrams (HOWDDs)* which is said to allow implementing new decision diagram variants easily. Further is stated that the reordering framework applies the advantages of multithreading techniques to reordering algorithms. To resolve the problem of computed-tables not being as efficient in multi-threaded environments a new *multi operand apply* algorithm that eliminates the creation of temporary nodes was developed. Since JINC was not capable of working with MDDs that have a variable branch count, it was extended in this thesis. The extensions include two existing MDD variants and a new branch compression technique. Many algorithms had to be adjusted. Afterwards automotive use case algorithms were implemented and adjusted to exploit the multithreading capabilities of JINC. Also JINCs features like the *multi operand apply* and the implemented *genetic reordering* algorithm were evaluated for MDDs and the product configuration context.

Chapter 2

Background

In this chapter, an introduction to the automotive context and product configuration problems is given. Then decision diagrams are explained on the basis of binary decision diagrams. Typical operations and data structures that are used to implement BDDs are presented.

2.1 Product Configurations

The variant diversity information for configurations can be formalized according to [9] as follows: A product is defined by a set of *attributes* with $A := \{a_1, ..., a_n\}$. This set is partitioned in distinct sets of *features*, $F = \{F_1, ...F_m\}$, with $F_i \subseteq A$ and $\forall F_i, F_j \in F, F_i \neq F_j : F_i \cap F_j = \emptyset$ (partition property). A product configuration is a set of attributes that contains exactly one attribute per feature, $p \in F_1 \times ... \times F_m$. The set of all product configurations shall be $W = F_1 \times ... \times F_m$. Since not every configuration is producible, the manufacturer wants to reduce complexity or for other reasons, some configurations have to be prohibited. Therefore *constraints* are introduced. A *constraint* is a tuple

$$c := (P_1, ..., P_m), P_i \subseteq F_i \tag{2.1}$$

which prohibits the set of *configurations*

$$W(c) = P_1 \times ... \times P_m, W(c) \subseteq W \tag{2.2}$$

The set of valid configurations is the set of all configurations without the prohibited configurations. For a set $R = \{c_1, ..., c_k\}$ of *restrictive constraints*, the set of valid configurations is

$$\overline{W}(R) := W \setminus \bigcup_{1 \leq i \leq k} W(c_i). \tag{2.3}$$

Since in the first part of this thesis only two-valued operations are used, the rule has to be split up. There are several way to do this. The choice of the unification order influences the performance as more precisely explained in Chapter 4. For now the chosen rule is:

$$W(R_i) = W(R_{i-1}) \cup W(c_i) \tag{2.4}$$

with $W(R_0) = \{\}$ and

$$\overline{W}(R) = W \setminus W(R_k) \tag{2.5}$$

In Section 6.7, a multi-operand unification will be examined. In Section 4.3 other two-operand unification methods will be evaluated.

One possible validation with these sets is a check for unused parts. All parts of an automotive system are listed in the bill of materials. Not all parts are used for a single instance of a automobile. Whether a part is used or not is decided by checking if a part constraint is fulfilled. For a set B of all constraints in the Bill of Materials, the part constraints are consistent to the restrictive constraints if

$$\forall b \in B : b \cap W(R) \neq \emptyset \tag{2.6}$$

Table 2.1 shows an example for the different sets of the formalization. The example contains ten attributes that can be used to describe a car, it can be black, it can have a led, etc.. As easily can be seen it makes sense to organize this attributes. A car only can be black or orange, but not both. The attributes are categorized into three features. For example, the colors belong to the feature *car paint*. A car consists of exactly one attribute per feature, as the example car has the attributes white, electric and led. As an example for a restrictive constraint, a rule that prohibits orange or green electric cars is given. A customer would have to choose a colorless car if he wants to buy an electric car in this example. Diesel or gasoline cars would be available in all colors if no other rule is stated.

Most of the time a single constraint is intuitive and can be verified manually. But since automotive manufacturers have many restrictive constraints and even more part constraints, it is hard to check the correctness of the sets. Multi-valued decision diagrams are one way to handle these big sets with relatively low amount of memory since it is a data structure that works on compressed data.

2.2 Binary Decision Diagrams

In this section, the ideas behind decicion diagrams are explained, some operations on BDDs are described, and techniques that are used to implement BDDs and the specific realization in JINC.

2.2.1 Introduction to Binary Decision Diagrams

A simple data structure to represent a binary function $f(x_1, .., x_n) \rightarrow \{0, 1\}$ is a decision tree (see Figure 2.1). The data structure can be created by the following method: One *variable* is mapped onto the *root node* of the tree. The nodes can also be called *vertex*. For every possible assignment of this variable (*true* or *false* for binary decision trees) a branch is inserted, which leads to a node that represents the next variable. Following the branch for the false assignment can be done with the function *low(v)*, for the true assignment *high(v)*.

attributes A	{black,orange,green,white,diesel, gasoline,electric,halogen,xenon,led}
features F	{car paint,motor,headlight}
F_1 : "car paint"	{black,orange,green,white}
F_2 : "motor"	{diesel,gasoline,electric}
F_3 : "headlight"	{halogen,xenon,led}
configuration p "fictional car"	{white,electric,led}
constraint c_1 "no orange or green electric car"	{{orange,green},{electric},{halogen,xenon,led}}

Table 2.1 – Example for sets that define an automotive and the diversity information

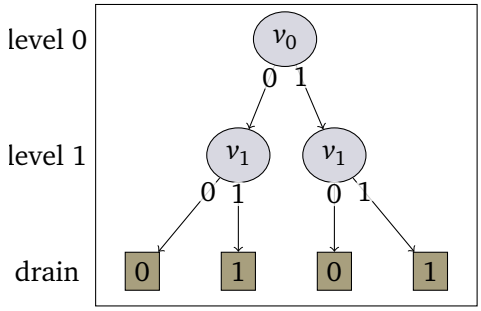


Figure 2.1 – Decision tree

When this has been done for every variable, a leaf (can also be called *drain*) is added, which represents the value onto which the functions maps the specific assignment. This tree originates in a Shannon expansion [11]. If unreduced, the node count for a binary tree is $2^{n+1}-1$ with the number of variables as n of the switching function.

Since a decision tree has exponential size, it can't be used to represent larger functions. To be able to reduce the size of the data structure, a binary decision diagram (BDD) can be used instead. A decision diagram is a *rooted directed acyclic graph (DAG)*. This means, multiple branches can point at a node. This data structure was first developed by Lee [12] and investigated for its full potential by Bryant [13]. Bryant introduced so called *reduced order decision diagrams (ROBDDs)*, which is usually meant when one refers to BDDs. An ROBDD is a binary decision diagram with two additional restrictions:

- it is ordered
- it is reduced

Ordered means that the order in which a variable appears is the same for each path. Bryant [13] defines the reduced property as follows:

"A function graph G is reduced if

- it contains no vertex v with *low(v)=high(v)*
- nor does it contain distinct vertices v and v $'$ such that the subgraphs rooted by v and v $'$ are isomorphic."

The first rule will in the following be referred to as *redundant node rule*. For an example of this rule see Figure 2.2. The node is removed, because both

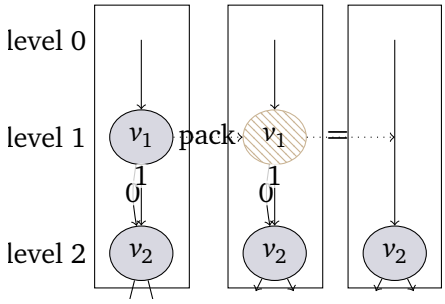


Figure 2.2 – Redundant node removal

branches point at the same node. Therefore no decision has to be taken and the node can be seen as redundant. For the second rule see Figure 2.3. In the example two nodes have the same subtree – they point to the same drains with the same values. One node can here be economized and instead the other node is referenced. Note that in the result of the example the redundant node rule would apply. For a fixed variable ordering, ROBDDs are a canonical representation for Boolean functions. Since functions are a "relation of inputs and a set of permissible outputs with the property that each input is related to exactly one output" [14] BDDs can also be used to store sets, if all elements in a set are exclusively connected with the same element of the output set. Although BDDs are in many cases a heavily compressed description for binary functions, the worst case size is still exponential.

Many variations to ROBDDs were developed for specific use cases to maximize the compression and speed-up calculations. One variation is referred as "shared BDDs". At this version, different BDDs share equal subgraphs.

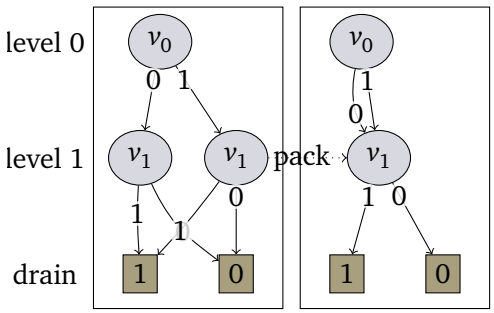


Figure 2.3 – Removal of nodes with isomorph subgraphs

This allows to reduce the size of the BDDs and speeds up some operations, e.g. comparing two BDDs for equality has the runtime complexity of O(1), since equal BDDs have an equal root node. Some variants handle BDDs with a high number of drain nodes.

One solution is BDDs with edge values (algebraic decision diagramss (ADDs)). Another variant allows inverting the drain value of the subgraph with a mark (complemented edges). This allows to reuse a subgraph that is equal except for the drain value.

2.2.2 Inserting a New Node

To ensure the ROBDD properties, one can use a *reduce* function that gives the graph the reduced property. Another way would be an *insert* function that guarantees that the BDD is reduced after every node insertion. To ensure that no two nodes with isomorphic subgraphs exist, there is a data structure called *unique-table* in JINC. Before an *insert* operation of a node is done, there are no distinct nodes with the same subgraph.

Therefore, to insert a new node, it simply has to be checked if a node with the same branches exists. If this is not the case, a new node can be created and inserted. Otherwise the existing node is given back. This way, before and after an insertion there are no two nodes with isomorphic subgraphs. To check if a node with the same branches exist, an associative array can be used. This is an abstract data structure that uses keys to reference values, where each key is unique. The unique table has branches as a key and the corresponding node as value. The node contains the branches (key), since a mapping from the branches of a node to the node should be realized. A hash table is well suited as implementation for the associative array because BDD algorithms need a lot of lookups. A hash table usually has a runtime complexity of O(1), but a worst case of O(n) where n is the number of entries. Therefore it is important to use a well fitted hashing function. This will be discussed later. There are different ways to implement the collision resolution, the most common is a linked list for every bucket. To identify the corresponding value to a given key in the linked list, key and value are saved together. In this case, the key does not have to be saved with the value, since the value (node) contains the key (branches). The collision resolution uses chaining, a bucket contains a node, which has a next pointer, and therefore acts as a linked list. JINC uses one unique table per variable. This approach offers

advantages when using multithreading and speeds up the garbage collection as it is implemented in JINC. The insertion function is named *findOrAdd* in JINC and guarantees that every node is unique for the given unique tables.

2.2.3 Logical Operators on Binary Decision Diagrams

To work with BDDs in binary function context, there should be operators like *AND* and *OR*. Operators for BDDs are often implemented with an abstract algorithm, which is often referred as *apply* algorithm that allows implementing different operations with minimal effort.

Apply is a recursive algorithm whose basic idea is to simultaneously go through all paths of the graphs of two BDDs and let the operator decide at every step, if the recursion is aborted. A so called *terminalCase* function also decides which node should be returned in the case of recursion abort. See Figure 2.5 for the rules of the *apply* algorithm.

Rule (2) and rule (3) are needed because of the redundant node rule, to ensure that *apply* is on both BDDs on the same level. All 16 binary two operand operations can be implemented by writing a *terminalCase* function. This operation is usually accelerated with the usage of a caching mechanism. This cache is referred to as “computed table” in JINC. It is a hash table that has as key two BDD nodes plus a static number for the operation as key and maps it to an already computed node. The value of the hash table is the two nodes and the resulting node (See Figure 2.6). The two nodes are saved as value for collision detection. The operation number is not saved, but included in the hash function. The implementation in JINCs uses the last bytes of a pointer to store the operation code. Therefore the operation

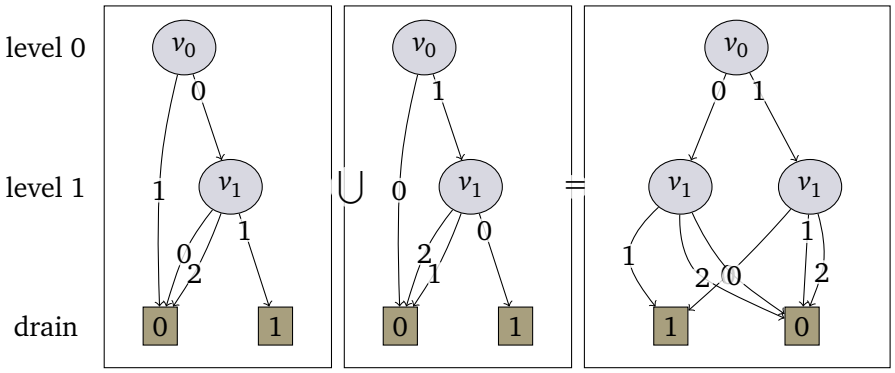


Figure 2.4 – MDD unification

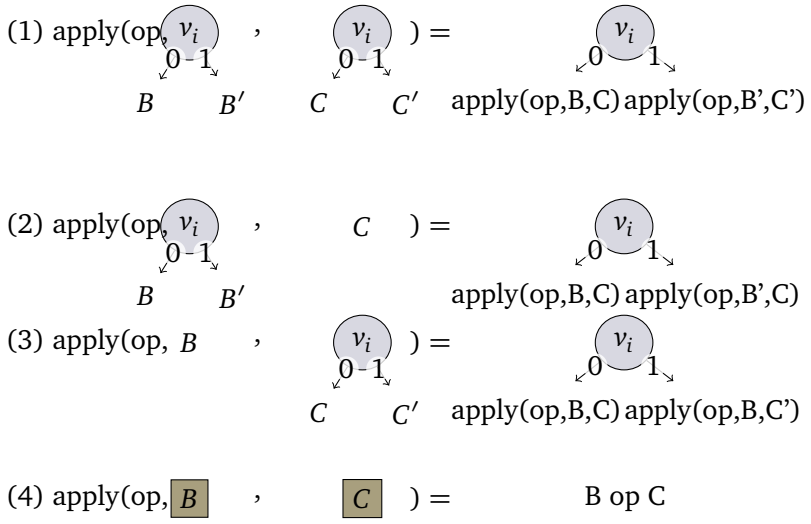


Figure 2.5 – Basic idea of the *apply* algorithm. Note: C in (2) and B in (3) can be drain or a node with higher level than the other

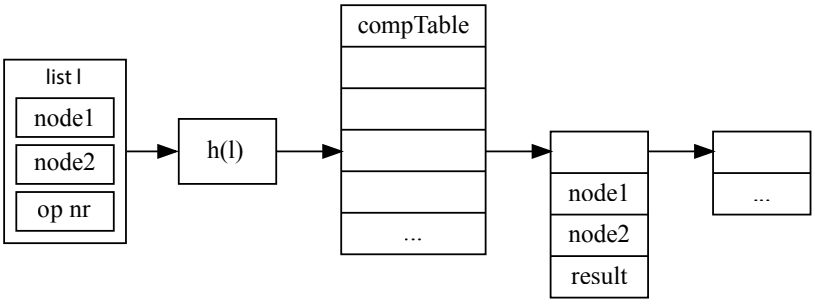


Figure 2.6 – Computed table

code must not be longer than a byte. JINC solves this by dividing operators into classes and provides a cache for every class. From the set perspective, the *OR* operation is a unification. A sample unification of two BDDs can be seen in Figure 2.4. The only other set operations needed in this thesis are *complement* and *intersection*, which correspond to *NOT* and *AND*.

2.2.4 Level Swap

ROBDDs have a fixed variable ordering (see $v_1 < v_2 < v_3$), which can be changed at runtime. That this is very import; why it is will be explained in Chapter 4. To change the variable ordering (see $v_2 < v_1 < v_3$) the partial

before swap	after swap
$A_1 \xrightarrow{0} B_1 \xrightarrow{0} C_1$	$A_1 \xrightarrow{0} B_3^* \xrightarrow{0} C_1$
$A_1 \xrightarrow{0} B_1 \xrightarrow{1} C_2$	$A_1 \xrightarrow{1} B_4^* \xrightarrow{0} C_2$
$A_1 \xrightarrow{1} B_2 \xrightarrow{0} C_3$	$A_1 \xrightarrow{0} B_3^* \xrightarrow{1} C_3$
$A_1 \xrightarrow{1} B_2 \xrightarrow{1} C_4$	$A_1 \xrightarrow{1} B_4^* \xrightarrow{1} C_4$

* new node if not already existing

Table 2.2 – Paths from a node A_1 before and after a level swap

paths between v_1 and v_2 have to be inverted. The partial path $(0,1)$ from a node A_1 at v_1 to a node at v_3 should now be $(1,0)$ but still be connected to the same node at v_3. The node at v_1 must be the same after the swap, since some node at a lower level might be connected to this node. The swapping will also not affect the nodes at the levels above (see [15],). For the new paths see Table 2.2. These rules are generally applicable, for every combination nodes and their branches can have in a BDD, but both reduction rules have to be considered. If $C_1 = C_2$ then C_1 is the zero successor of A_1 or if $C_3 = C_4$ then C_3 is the one successor of A_1 according to the redundant node rule. The insertion method guarantees the ROBDD properties after insertion of the new nodes.

Chapter 3

Implementing Multi-valued Decision Diagrams

In this chapter multi-valued decision diagrams are introduced, some variants are described, and the algorithms to implement the two MDD variants in JINC are showed.

3.1 Introduction

Multi-valued decision diagrams (MDDss) are usually defined as a extension of BDDs for a function $f : p^n \rightarrow p$ with $p = \{0, ..., n\}$ (see [16]). Since – in the configuration context – MDDs are used to encode subsets of the Cartesian product of different sets, the MDDs here encode a function $g(v_1, ..., v_n) : F_1 \times ... \times F_n \rightarrow \{0, 1\}$ with $v_i \in F_i$. Since the variables $v_i \in F_i$ have $|F_i|$ values, and a node needs to store a branch per value, the number of branches is variable per level of the MDD. Because of this difference, and due to the fact that JINC only supports a 4-valued MDD variant, so called *toggling algebraic decision diagrams (TADDs)*, some changes had to be made. To support a dynamic branch width for every variable, this information had to be stored additionally in the variable data structure. To support branch compression techniques, the branch width of a node should also be variable. This required a adjustment of the implemented algorithms, which was anyway necessary, as the algorithms partially only supported 4-valued MDDss. In the following, the reduced order multi-valued decision diagram (ROMDD) rules, level swap and insertion of nodes are described.

3.2 Reduction Rules

As reduction rules the BDD rules can be extended for more branches.

- No two nodes with isomorphic subgraphs may exist.
- A node is redundant if every branch contains an isomorph subgraph.

If the MDD is reduced then the second condition is fulfilled when all branches point to the same node.

3.3 Variants

MDDs have different properties than BDDs regarding the compression. Since MDDs have more branches, it can be assumed that the reduction rule has a lower probability for MDDs. Therefore, the implications of this rule are investigated. The reduction rule increases the complexity of the code, and also the amount of data, since a reference from the node to the level has to be saved. Therefore the influence of this rule was investigated for MDDs in the configuration context in this thesis. Gianfranco Ciardo calls MDDs without this rule *quasi reduced*, with the rule *fully reduced* [3]. Since the encoded functions map to $\{0, 1\}$, the probability of a branch pointing at a zero drain should be high in automotive context. For BDDs, this scenario can be efficiently handled with the usage of zero-suppressed decision diagrams (TADDs) [17]. Since this is not a feasible solution for MDDs, there is another solution. The approach has some similarities to sparse matrices [18] and is called *sparse representation* by Gianfranco Ciardo. Branches to the zero drain are not saved at this variant. Hence am additional index has to be saved for each branch so that gaps can be detected. See Figure 3.1 for a MDD, where only the equal subgraphs were removed. Figure 3.2 shows the same MDD but branches to the zero drain were removed. The remaining empty node on level one is also removed. Figure 3.2 shows the MDD with the *redundant node* rule applied. Finally, Figure 3.5 shows again the same MDD without redundant nodes and branches to the zero drain.

To implement a *sparse representation*, the *swap* and the *apply* algorithm of JINC had to be adjusted.
Since in the automotive use case there are nodes with around 100 branches and it has been observed that they often point the same node (see Figure 4.8),

it makes sense to compress the branch storage. For that reason, a new compression feature for the branches is introduced. It is an adapted run-length encoding [19] for the branches. For every branch, the number of following branches that point at the same node is saved. Thus the following branches do not have to be saved explicitly. No gaps between the indices are allowed if the MDD is a *sparse representation*. See Figure 3.5 for an example. Run-length encoding works well for MDDs, because it is not complicated to implement it and does not affect runtime performance negatively.

3.4 Inserting a Node

As the number of branches is dynamic, the unique table had to be modified to handle dynamic width instead of the width being a template parameter. The hash function used in the unique table was extended to be able to hash a set of branches and map it to a node. For a unique table access see Figure 3.6. To find a node based on a set of branches, the branches are hashed, the respective table is selected, and the position in the table is the hash value. In case of a collision the nodes are linked so that for a lookup, the branches of the node have to be checked for equality. The node insertion function searches in the unique table if the branches for the new node exists, if not a new node is created. See Algorithm 3.2 for the MDD insertion algorithm. Since the original hashing algorithm was not fitted for 64 bit it had to be adapted. Thereby also alternatives were tested. Since the hash function hashes pointers and the memory addresses change for every run with ASLR [20], different

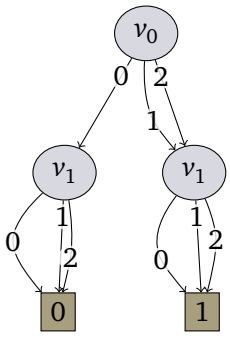


Figure 3.1 – MDD variant: nothing hiden (quasi reduced)

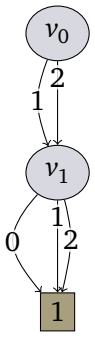


Figure 3.2 – MDD variant: hide branches (sparse representation)

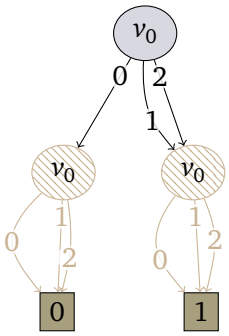


Figure 3.3 – MDD variant: hide nodes (fully reduced)

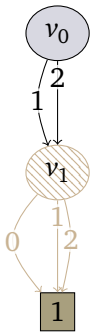


Figure 3.4 – MDD variant: hide nodes, hide branches (fully reduced sparse representation)

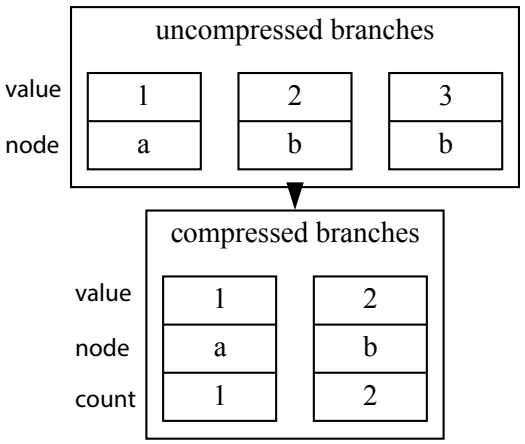


Figure 3.5 – Branch compression

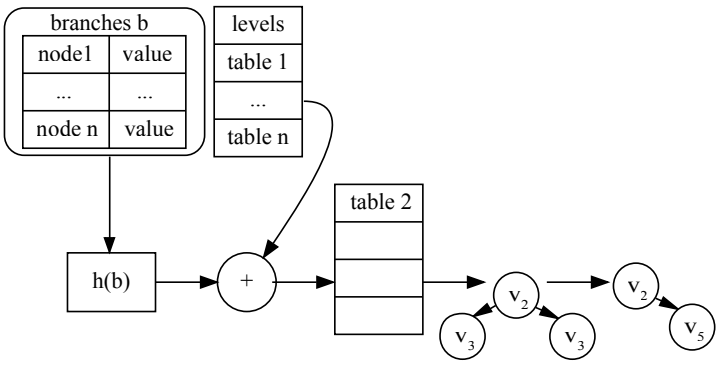


Figure 3.6 – Access to the unique table, part of *findOrAdd*

results for every run should be expected. Nevertheless after several test runs it was replaced by a hashing algorithm by Thomas Wang [21], who implemented a fast hashing algorithm specialized for 64 bit integers that is derived from general hashing algorithms. It is said to be *avalanche*. This is a criterion that demands that with the changing of one bit the output has to change by a chance of about 50%. Cryptographic hashing algorithms have to fulfill this criterion. For a comparison of different general purpose hash functions see [22]. Next to the Thomas Wang algorithm, the popular *Google CityHash* was tested. Google CityHash is also *avalanche*. CityHash is a state of the art general purpose fast hashing algorithm [23] and is based on MurmurHash. MurmurHash performs well in a random distribution of regular keys (see [24]). CityHash also offers an extension to use the CRC32 hardware acceleration capabilities of SSE4.2 to build upon it a fast hashing algorithm. Google CityHash performed slightly worse than the Thomas Wang algorithm (3% overall runtime performance), therefore Thomas Wang algorithm was chosen for all subsequent tests. The Thomas Wang hash function is public domain, Google CityHash is under the MIT-license.

3.5 Level Swap

For the MDD level swap, also the partial parts have to be inverted as it was shown for BDDs in Table 2.2. Now not only 0 and 1 paths have to be inverted but the paths for all values. If the MDD is not a sparse representation, the expense to swap the variable v_i is width(v_i)$\cdot$ width(v_{i+1}) inverting steps. Algorithm 3.1 shows the swapping algorithm form MDDss. It iterates over all nodes in a level (line 3), descends two levels (line 7,11) and saves the inverted paths in an array w. If no branch points to the level above, no swap for this node is necessary (line 10). This rule is important, as will be seen later. The created array is then taken, and for every value on level $i+1$ a new node is created if not already existing (line 27,23). To create a fully reduced MDD, it has to be checked if the *redundant node rule* applies before a new nodes is inserted (line 24). To avoid using a temporary hash table or other data structure, JINC inserts the newly created node in the hashtable of level i. Afterwards the swap operation it should be on level i+1. Inserting it to level i allows the algorithm to iterate over the node again, and since no branch points at level i+1, no swap is necessary. The nodes inserted nodes are step 1

in line 3.8. Now the root node is removed from the hashtable for level i, the branches and the linked subtree are deleted and branches to the new nodes are inserted (step 2 in Figure 3.9). If this has been done for all nodes on level i, the entries in the level list are swapped for the level i and i+1 (step 3 in Figure 3.9). Now the two levels are swapped (see Figure 3.10).

3.6 Operations on MDDs

Compared to the BDD *apply*, an extra condition has to be handled to support a sparse representation. See Figure 3.11 for different the conditions that can occur for the branches of two nodes. The grey boxes symbolize not saved branches and the zero drain the implicit node for this branch. The requirements for the MDD *apply* are for the two nodes n_1 and n_2 as following: If the level for a branch of one node is lower than the branch with the same value of other node, call *apply* recursively for these branches but descend only for the node with the lower level. If a branch exists for one node but not the other, call the operator with the zero drain instead. For saving of a newly created node applies: If a sparse representation should be created and a branch points at the zero drain, do not save it.

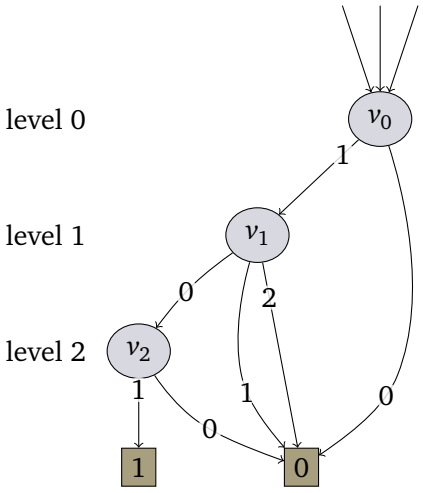


Figure 3.7 – Level swap step 1: original MDD

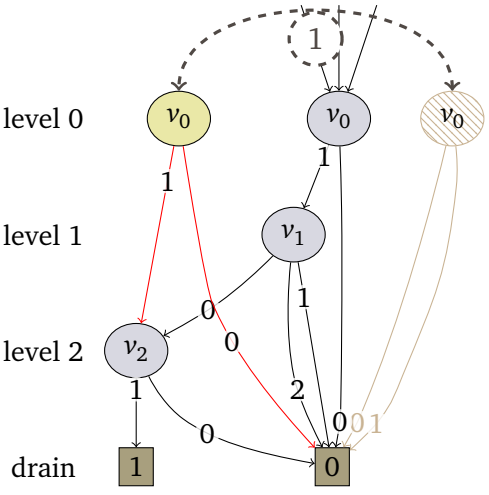


Figure 3.8 – Level swap step 2: calculation and insertion of new nodes

Require: level to swap in v_i
1: $h_1 \leftarrow$ hashtable(v_i)
2: $h_2 \leftarrow$ hashtable(v_{i+1})
3: **for all** $n \in h_1$ **do** {iterate over every node n in linked list of every bucket of h_1}
4: $w \leftarrow$ array$_{width(v_i)}$(list) {w saves the branches for the new nodes}
5: $r \leftarrow$ list {r saves the branches to the new nodes, which will be the new branches of n}
6: **if** $\exists n_{succ} \in$ branches(n) with level(n_{succ}) $= i+1$ **then** {at least one branch points to the level above}
7: **for all** $b_{succ} \in$ branches(n) **do**
8: $n_{succ} \leftarrow$ node(b_{succ})
9: $v_{succ} \leftarrow$ value(b_{succ})
10: **if** level(b_{succ}) $= i+1$ **then**
11: **for all** $b_{succsucc} \in$ branches(b_{succ}) **do**
12: $n_{succsucc} \leftarrow$ node($b_{succsucc}$)
13: $v_{succsucc} \leftarrow$ value($b_{succsucc}$)
14: w[$v_{succsucc}$].append(v_{succ}, $n_{succsucc}$)) {invert partial path, but keep $n_{sucsucc}$}
15: **end for**
16: **else** {node is hidden by redundant rule}
17: $n_{succsucc} \leftarrow$ node($b_{succsucc}$
18: **for all** $j \in \{0,...,\text{width}(v_i)-1\}$ **do**
19: w[j].append(v_{succ},$n_{succsucc}$)))
20: **end for**
21: **end if**
22: **end for**
23: **for all** $w_j \in w$ **do**
24: **if** every branch $\in w_j$ is equal **then** {redundant node rule applies, so don't create a new node, save branch to node in level above instead}
25: r.append(branch(j,w_j[0]))
26: **else** {add new node and save branch to node in r}
27: r.append(branch(j,findOrAdd(w_j)))
28: **end if**
29: **end for**
30: remove n from h_1
31: set branches of n to r and add to h_2
32: **end if**
33: **end for**
34: swap in level array v_i, v_{i+1}

Algorithm 3.1 – Swap algorithm

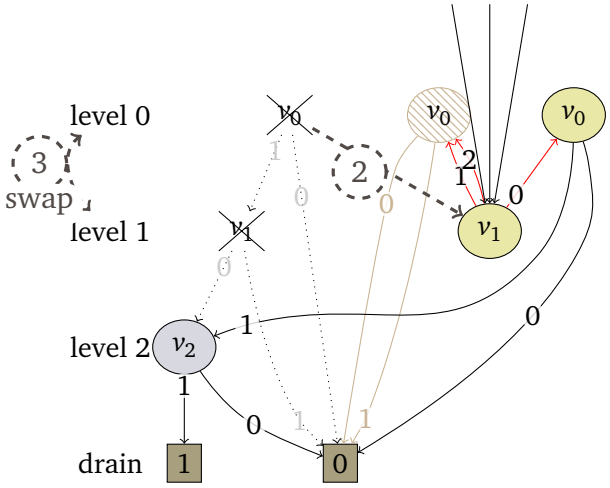


Figure 3.9 – Level swap step 3: move and reinitialize *root node* with branches to the new nodes (2) and swap level(3)

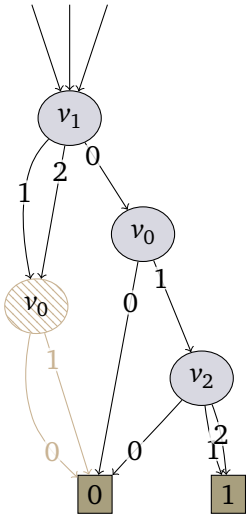


Figure 3.10 – Level swap step 4: resulting MDD

Require: Branches of the new node in set br
Ensure: Maintain reduced property of MDD and add new node if needed
1: index ← hash(br)
2: tempNode ← uniqueTable[index]
3: **while** tempNode **do**
4: tempBranches ← tempNode.branches
5: **if** tempBranches $\widehat{=}$ br **then**
6: **return** tempNode
7: **end if**
8: tmpNode ← tempNode.next
9: **end while**
10: newNode ← createNewNode(branches)
11: uniqueTable[index].insert(newNode)

Algorithm 3.2 – MDD findOrAdd

For the MDD *apply* algorithm see Algorithm 3.3. It should be mentioned that this algorithm always returns a zero drain if it operates on two branches that point at a zero drain, as for our use case no other behavior was needed, and the performance can be improved a bit this way. To support branch compression, *apply* should be able to handle variable branch width. It has to be considered that branches to the zero drain are not saved. In case that *apply* is called for a MDD with a hidden branch, the branch is for the recursion replaced by a pointer at the zero drain.

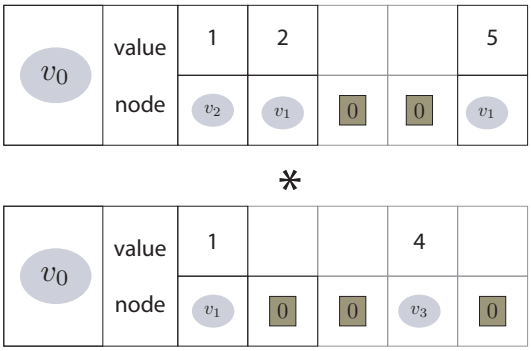


Figure 3.11 – Two nodes and their branches. Example for the different cases that MDD *apply* should be able to handle

Require: Two MDDs, represented by their root nodes n_1 and n_2
1: comp ← terminalCase(n_1,n_2) {operator specific termination criterium}
2: **if** comp ≠ 0 **then**
3: **return** comp
4: **end if**
5: **if** comp ← search in computed table(n_1,n_2) **then**
6: **return** comp
7: **end if**
8: **if** level of n_1 and n_2 is equal **then**
9: it_{n_1} ← iterator(branches(n_1))
10: it_{n_2} ← iterator(branches(n_2))
11: w ← list()
12: **while** (NOT atEnd(it_{n1})) OR (NOT atEnd(it_{n2})) **do**
13: **if** NOT atEnd(it_{n1}) **then**
14: v_{succ1} ← value(it_{n1})
15: **else**
16: $v_{succ1} \leftarrow \infty$
17: **end if**
18: same for value(it_{n2}) and v_{succ2}{...}
19: **if** $v_{succ1} \leq v_{succ2}$ **then**
20: n_{succ1} ← node(it_{n1})
21: **else**
22: n_{succ1} ← drain(0)
23: **end if**
24: vice versa (≥) for n_{succ2}{...}
25: c ← apply(n_{succ1},n_{succ2}) {recursive descend}
26: **if** NOT (sparse representation AND c = drain(0)) **then**
27: w.append(Branch(min(v_{succ1},v_{succ2}), c))
28: **end if**
29: **if** $v_{succ1} \leq v_{succ2}$ **then**
30: $++it_{n1}$
31: **end if**
32: vice versa (≥) for it_{n2}{...}
33: **end while**
34: **else**
35: { handle redundant node rule, recursive descend}{...}
36: **end if**
37: **return** c

Algorithm 3.3 – MDD *apply* algorithm supporting a sparse representation

3.7 From Constraints to MDDs

Mapping from the formal description of constraints to MDDss is done in this section. As reminder: a constraint is defined as $c := (P_1, ..., P_m), P_i \subseteq F_i$. From a constraint, a set $W(c) = P_1 \times ... \times P_n$ can be constructed. A function $m : m(p) = 1 \leftrightarrow p \in W(c) \wedge m(p) = 0 \leftrightarrow p \notin W(c)$ is used, to map the sets to a MDD, since MDDs can encode discrete finite functions. Every configuration $p \in W(R)$ should be mapped to one, every element not in the set should be mapped to zero. Therefore a constraint consists of maximal one node per level where all items not in P_i of a feature F_i point at the zero drain, the other branches point at the next node and finally to the one drain. For an example see Figure 3.12. The MDD representation for the constraint from Table 2.1 is shown. A *restrictive constraint* represents a set that should be subtracted from the allowed configurations, therefore the MDD in the example encodes the configurations that contain “orange” or “green” and “electric”.

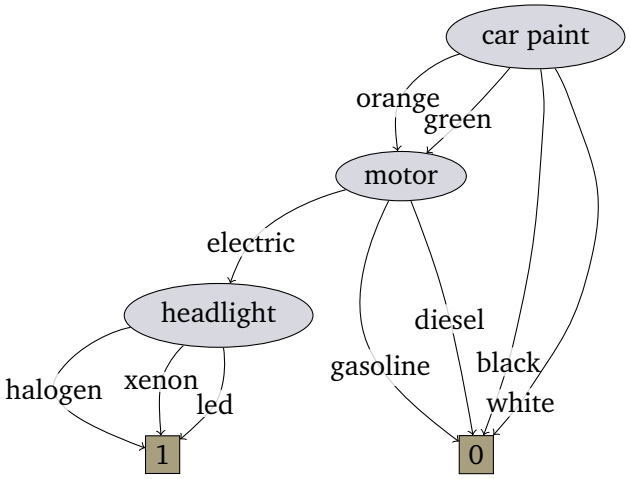


Figure 3.12 – Constraint “no orange or green electric car” as MDD

Chapter 4

Performance Optimization

In this chapter the factors that influence the performance when working with MDDs in general, and JINC specific factors, are stated and solutions are presented. The performance of the act of building $\overline{W}(R)$ and the verifying in the automotive context is influenced by several aspects. The most important aspects which will be explained in the following are the variable ordering and the constraint ordering. To improve performance parallel aspects will be discussed. Due to limited resources, it is important to optimize speed and memory size performance. The usual memory consumption curve in JINC for the automotive use case can be seen in Figure 4.1. The memory size for BDDs is usually measured by counting nodes, but since the MDDs in the product

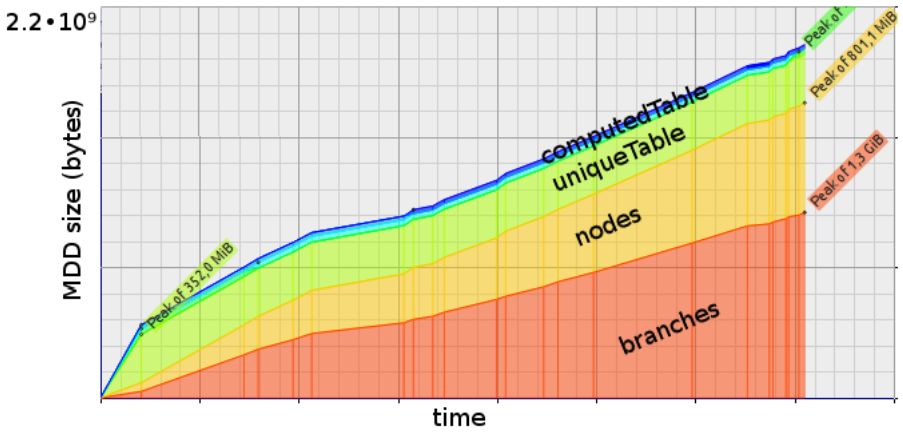


Figure 4.1 – Usual memory consumtion over time for building $W(R)$

configuration context have a varying branch count, it also has to be included in size calculation. The size of a MDD can be given in bytes, which can be calculated as following: $bytes_{MDD} = nodes \cdot size_{node} + branches \cdot size_{branch}$. Building $\overline{W}(R)$ for real world examples without any optimization is only possible with an extreme high amount of CPU and memory resources if not impossible.

4.1 Variable Ordering

It is well known that the size of a BDD/MDD heavily depends on the variable ordering. A bad fitting variable ordering can lead to an exponential sized MDD. The variable order can be improved, but it is *NP-hard* to find the best order [25]. Therefore a bunch of techniques were developed to optimize the variable ordering. There are static heuristics which use neighborhood relations between the variables to try to optimize the ordering offline; Heuristics like *force*, *distance* [26] or *mass spring relaxation (TADD)* [27]. It is also possible to dynamically optimize the variable ordering. For this purpose two methods are usually used: *window permutation* [28] and *sifting* [29]. Apart from that, other methods known from different optimization domains were adapted like *simulated annealing* and *genetic algorithms* [30] [31].

4.2 Constraint Ordering

The constraint ordering has a big influence on the size [32]. The constraint order is not commutative under performance aspects. For three different constraints c_1, c_2, c_3 the building process of a MDD for the set W = $(W(c_1) \cup W(c_2)) \cup W(c_3)$ is very likely to have different size and speed properties than another order $W(c_1) \cup (W(c_2) \cup W(c_3))$, since a different intermediate MDD will be created, which might even be bigger than the resulting MDD. See Figure 4.2 for the big influence, the constraint ordering can have.

This figure shows three memory consumption curves. One for an initial order which was too high to calculate the example, the next one was generated with a good variable order by a heuristic and the third with the initial variable order which seems to be not bad. The figure clearly shows that a bad constraint ordering can make it very hard to calculate $W(R)$, even if the final MDD would be small, the intermediate steps $W(R_i)$ are too big to fit into memory. A good

constraint ordering on the other hand leads to a monotonic growth of the MDD, as can be seen in the example.

To find a good constraint ordering for $\overline{W}(R)$ a *greedy algorithm* called *bestfit* was developed by [27]. The idea is to pick the best fitting constraint at every unification step $\overline{W}(R_i)$ which is when the resulting MDD is smallest. The constraints are then sorted by their relevance, as it is assumed that a good fitting constraint will also fit for the next intermediate MDD $\overline{W}(R_{i+1})$. To improve the speed two windows with the same window size are used and after every step the constraints are sorted after the resulting MDD size. The first window stays at the beginning, the second moves over the constraints so that every constraint is constantly checked but the first ones are checked every time, because of the mentioned assumption.

4.3 Multithreading

One characteristic of JINC is the shared-memory multithreading capability.

JINC offers multithreading on an operator level. This means that the calculation of a BDD operator like AND can be calculated in parallel with other operators. There already is a proof of concept implementation for a parallel “apply” with the usage of *cilk* [33] [34]. The implementations

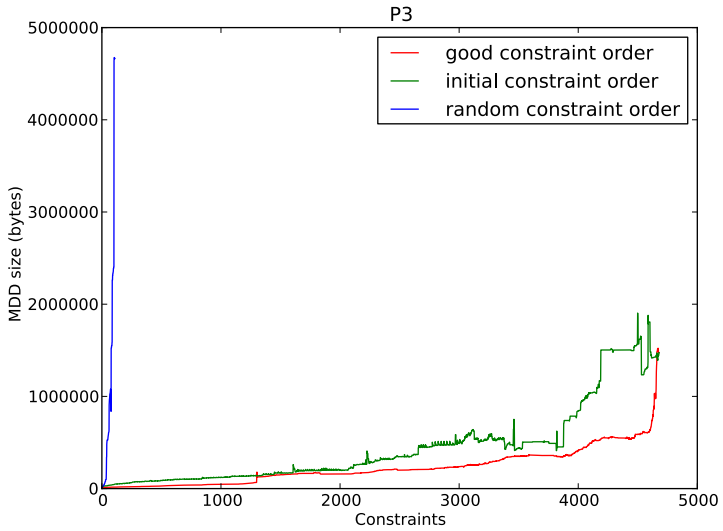


Figure 4.2 – memory consumption for different constraint orderings

let the *cilk* framework handle the thread management and take advantage of the integrated work stealing so that the recursive *apply* algorithm can be parallelized without a recursive increasing number of threads created, where most of them have to wait. Another implementation was realized with *wool* [35]. This kind of fine granular parallelism could also be implemented in JINC to use multithreading more easily. Therefore the thread local computed table would have to be thread-safe and shared across all threads since the thread management is done externally.

The advantages of the operand level type of multithreading in contrast to the sequential computing were explored for the automotive domain. Therefore *bestfit* and the check for not used parts were modified. To parallelize the execution of operators, a dependency between the operators executed in parallel must not exist.

The unification step can be reordered so that a partial parallel execution is possible (see Figure 4.3). If every unification would cost the same, the runtime complexity would be O(log(n)) for n constraints and processors. In praxis, the cost increases with every unification step since the MDDs grow. Therefore no big speedup should be expected. Also this approach can't take advantage of the *bestfit* method. To fix this issue, this method can be combined with the standard unification process. Small sets of constraints are unified in parallel and these sets are unified sequentially. See Figure 4.4 for the hybrid approach.

A better fitted algorithm for operator parallelization is *bestfit*. As described in the last section, it calculates for a MDD $W(R_i)$ the MDD size for a the unification of this MDD with respectively a constraint of a selection of constraints $H \subseteq R$. The constraint $h \in H$ where the resulting MDD for the operation $W(R_i) \cup W(h)$ is minimal is selected. There is no dependency between the

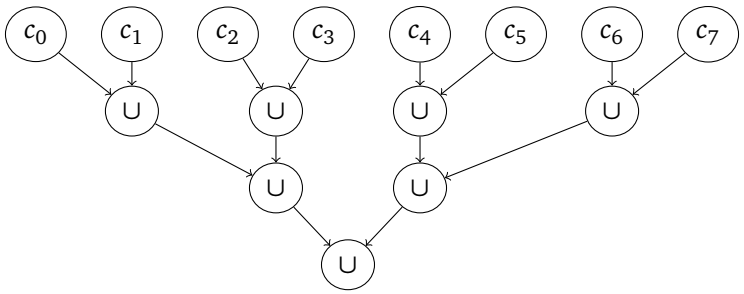


Figure 4.3 – Parallel unification

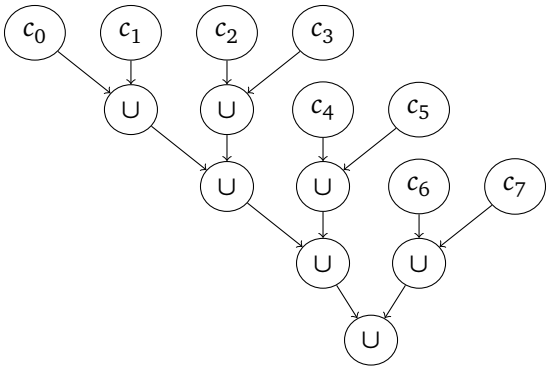


Figure 4.4 – Hybrid parallel sequential unification with a parallelism rank of two

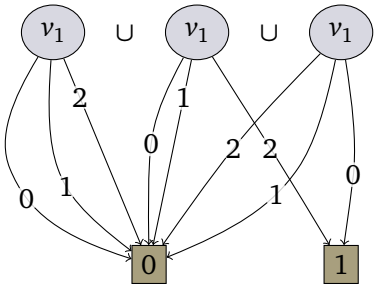


Figure 4.5 – Example MDD for multi operand apply

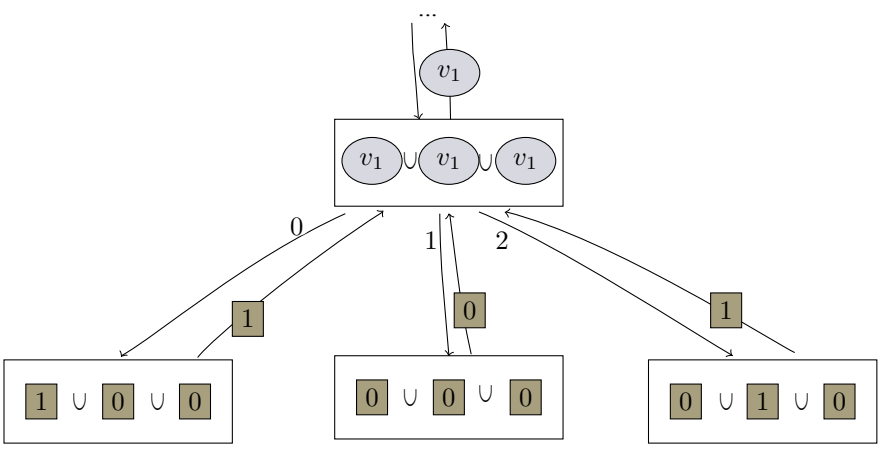


Figure 4.6 – *Multi operand apply* on MDD from Figure 4.5

different union operations, therefore they can be done in parallel. Results are that a speedup for *bestfit* by a factor 2.5-3 is possible. There are some possible approaches to improve the parallelism in JINC. One possibility would be using a shared computed table instead of using a thread local storage for the computed tables. Therefor the computed table would have to be thread-safe. A problem is the resizing operation. Another aspect that has to be improved is the *garbage collection*, since it doesn't scale well for our case. Garbage collection in JINC is divided into 5 operations and after each it has to be sequentialized. The parallelism is level-based, so cleaning up a single level is one job. Since in our case, the distribution of bytes to level is not uniformly (see Figure 4.7), and due to the high amount of sequencing points, garbage collection does not scale very well. For MDDs, the swapping of levels is a important task that costs a lot of computing time. It could be parallelized, but for JINCs a single level swap is atomic. Therefor it is just possible to use parallelism for reordering mechanisms like genetic reordering. Therefore genetic reordering was tried for our use case. The outcome can be seen in Figure 6.15 and Table 6.5.

Genetic optimization is not feasible in our case, even for a medium example the memory consumption and runtime are very high. Also the MDD size most often did not improve. Sifting and window permutation on the other hand offer a fast and reliable reduction for the MDDs with limited overhead. Sifting and window permutation are the standard techniques, implemented in all popular BDD packages. *BuDDy* offers sifting, window permutation with windows from 1-3 for blocks of variables (groups in JINC). CUDD offers random swapping (and checking for reduction), sifting (with limit for variables and BDD growth), symmetric sifting, group sifting, window permutation 1-4, *simulated annealing*, a *genetic algorithm* and *exact reordering* (finds guaranteed best order, but not recommended for more than 16 variables). The sifting implementation in JINC can be limited that the sifting of a variable is stopped at a too big growth of the MDD.

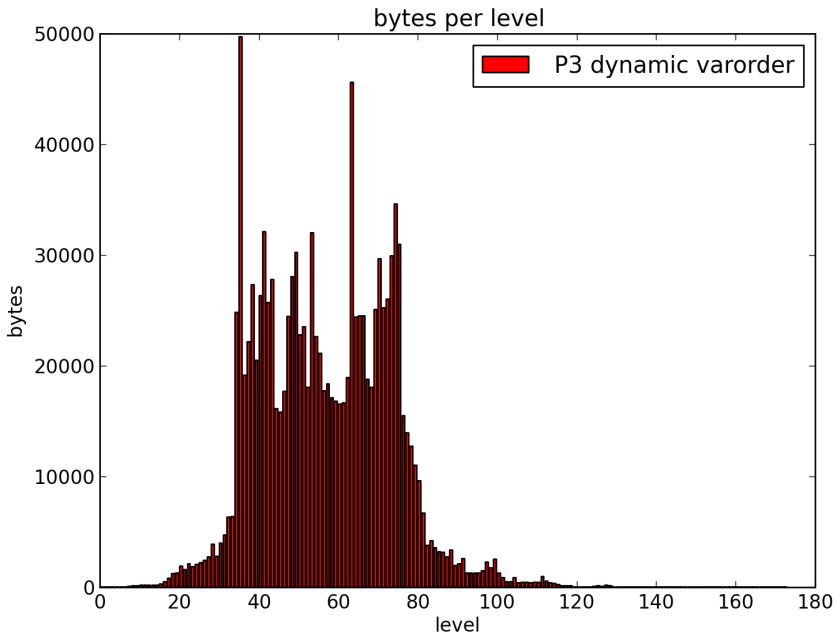


Figure 4.7 – Bytes per level for the testcase P3

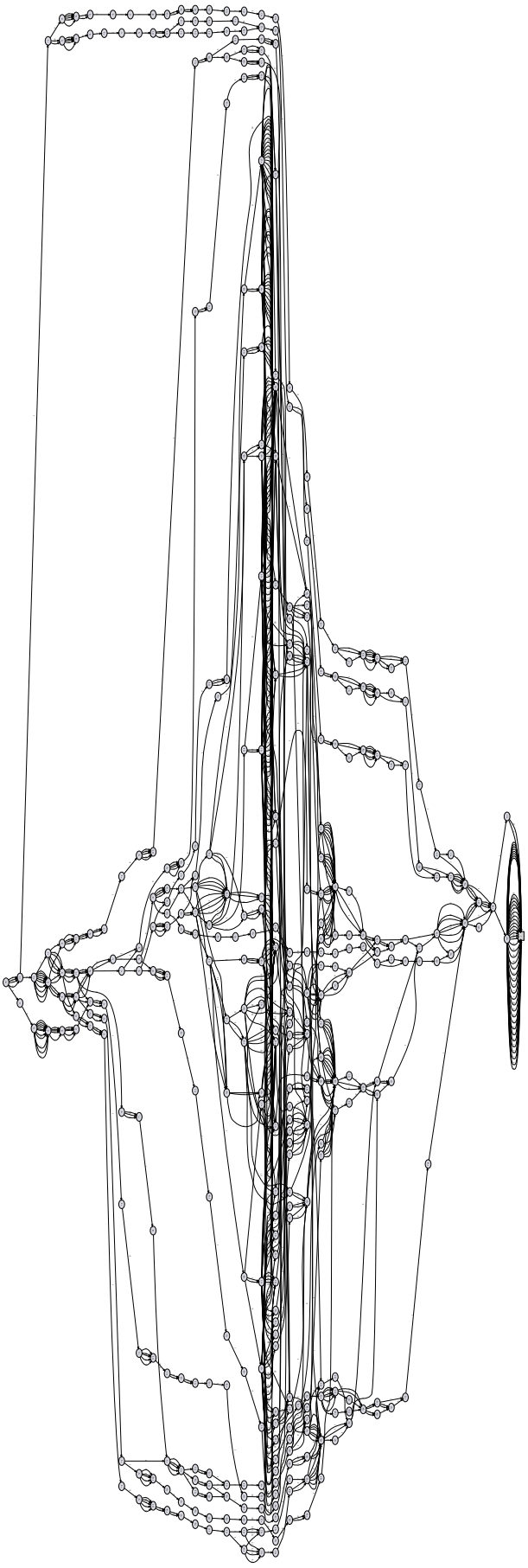


Figure 4.8 – Medium example, good variable order. The picture is 90° rotated to fit onto the page

Chapter 5

The Computation Process

In this section, the process from the given constraints to the part check is described and the used strategy to keep the MDD size small is explained. The process of compiling $W(R)$ is assembled from different parts (See Figure 5.1). It begins with a static variable ordering that is generated by the TADD method. The static variable ordering is quite good, but not good enough to compile all examples because of memory limitations. Therefore the in next step – the unification of the constraints – dynamic reordering is used if necessary. After every unification step, the MDD size is evaluated and checked against a threshold value t and a threshold factor t_f. If the MDD size is bigger than t

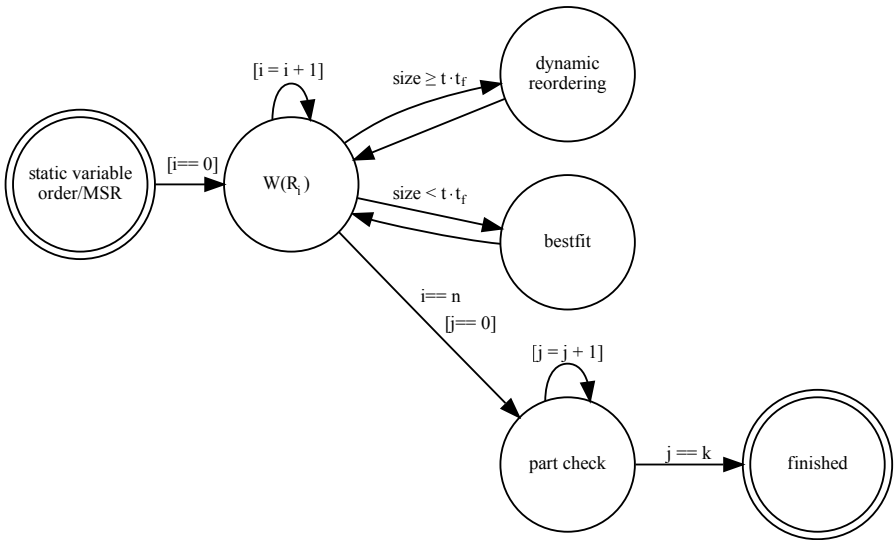


Figure 5.1 – Constraint unification process

· t_f reordering is activated. As reordering techniques window permutation with window size 2, window permutation window size 4 and sifting with maximum growth of 2 is used. The reordering is repeated until the size of the intermediate MDD is lower than the t or until no further improvement can be achieved. In that case the threshold is set to the new MDD size. The threshold together with the threshold factor creates a window in which the MDD size is tried to be fenced (see Figure 5.2). If a unification step is completed and reordering is done, the next constraint is selected. If bestfit is activated, the next constraint has to be computed first. If all constraints have been unified, the consistency check for the parts is done. The process is completed when every part has been checked.

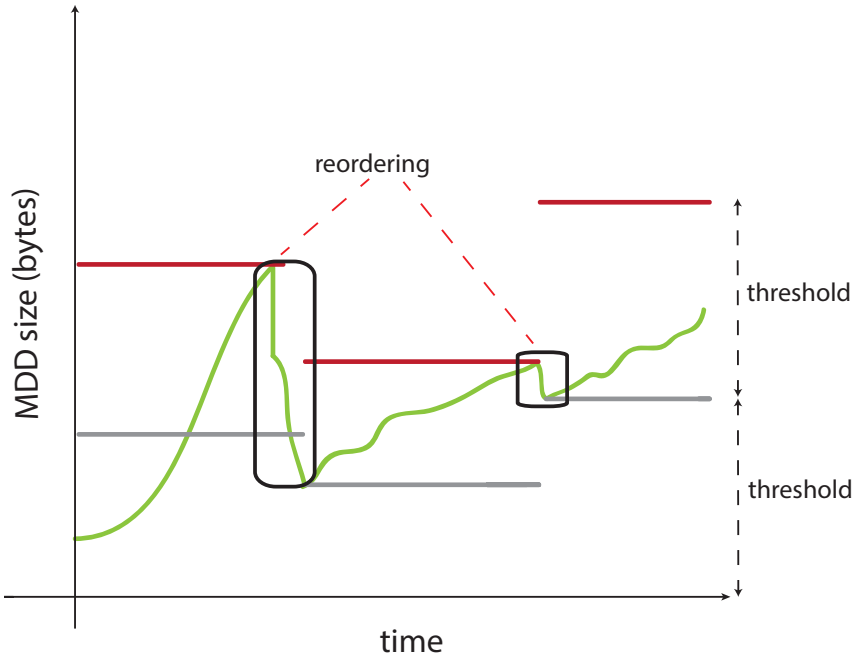


Figure 5.2 – Reordering strategy for threshold factor=2

Chapter 6

Performance Evaluation

6.1 Test Cases

To examine the performance of the MDD capable JINC, one medium sized example and 5 real world automotive examples(P1-P4,P6) were chosen. Due to the long runtime, the example P5 was skipped to be able to concentrate on other tests. Table 6.1 shows the properties of the test cases. As already mentioned in Section 2.1 $|A|$ is the number of attributes, $|F|$ the number of features, $|\mathrm{R}|$ the number of constraints, $|\mathrm{B}|$ the number of parts. $|\overline{W}(R)|$ is the size of the valid product configurations set and $|\mathrm{W}|$ the size of all product configurations. It is notable that $|\overline{W}(R)|$ is up to $5.17 \cdot 10^{71}(P4)$.

As comparison value: The number of worldwide available storage was estimated estimated around 2.9×10^{20} in 2007 [36]. $|\mathrm{W}|$ is up to 10^{105}, which is bigger than the estimated atoms of the visible universe that is 10^{80} atoms [37] by factor 10^{25}. For configurations sets of such size dimension can be handled by MDDs and compared rapidly. Also interesting is the achieved compression factor of $6.21 \cdot 10^{63}$ (P4) and shows the benefits of using decision diagrams very well. All tests with time declaration were executed on an "Intel(R) Xeon(R) CPU E5620 @ 2.40GHz" with 2 CPUs and 8 cores each and 20 GB Memory. Tests without time declaration are partially produced with 60GB memory. The average of 5 test runs was calculated. To find statistical outliers, the values were sorted by their distance to the arithmetic mean. Then iteratively, if the most distant value was, if more than %10 away from the arithmetic mean, dropped. It was made sure that the number of dropped

results was lower than 10% and not more than 2 values were dropped for a single test. Tests were repeated if necessary.

6.2 Calculating the Test Cases from a Statical Variable Ordering

For the memory usage while computing the different test cases with the static variable ordering generated by the TADD method see Figure 6.1 (P1), Figure 6.2 (P2), Figure 6.3 (P3), Figure 6.4 (P4), Figure 6.5 (P6). Notable is, that with *bestfit* the final MDD size is not always as good as without. Another interesting effect is that the intermediate MDD size close to the last constraint can be higher with bestfit than with the initial order. But it should also be considered, that the initial order is quite well as was shown in Figure 4.2. The results were also compared to a i7-chair prototype. The results can be seen in Table 6.2. Since JINCs nodes use 56 byte, the branches 16 byte and the prototype 72 bytes for the nodes and 24 byte for the branches two byte informations are given to be able to compare both. The prototype implements

test case	$\|A\|$	$\|F\|$	$\|R\|$	$\|B\|$	$\|\ \overline{W}(R)\|$	$\|\mathrm{W}\|$
medium	538	96	411	-	$5.58 \cdot 10^{46}$	$8.98 \cdot 10^{52}$
P1	989	168	2487	-	$1.06 \cdot 10^{66}$	$2.77 \cdot 10^{94}$
P2	1001	176	3040	-	$4.00 \cdot 10^{53}$	$3.96 \cdot 10^{97}$
P3	943	174	4679	19023	$6.06 \cdot 10^{53}$	$2.56 \cdot 10^{85}$
P4	1240	181	4413	21947	$5.17 \cdot 10^{71}$	$5.17 \cdot 10^{105}$
P6	781	157	2161	13644	$4.32 \cdot 10^{47}$	$6.79 \cdot 10^{76}$

Table 6.1 – The test cases and their properties

a quasi-reduced MDD, whereas JINC used the full-reduced MDD.

test case	MDD-tool	nodes	branches	bytes/i7	bytes/JINC	times
P1	i7-chair	306085	886731	43319664	30104116	
P1	JINC	267137	751939	37280400	25922148	101m
P1	JINC bf[1]	580952	1595674	80124720	55740288	59m
P2	i7-chair	203086	435072	25063920	17521624	
P2	JINC	202232	437193	25053336	17511152	108m
P2	JINC bf[1]	-	-	-	-	
P3	i7-chair	16828	12610	1514256	1076816	
P3	JINC	38850	26703	3438072	2447448	3m
P3	JINC bf[1]	11608	25815	1455336	1016656	7m
P4	i7-chair	369885	1001605	50670240	35259700	
P4	JINC	823378	2318424	114925392	79910440	28m
P4	JINC bf[1]	-	-	-	-	
P6	i7-chair	6827	14243	833376	582892	
P6	JINC	9226	25525	1276872	888152	2m
P6	JINC bf[1]	6064	16150	824208	573728	3m

Table 6.2 – Size comparison between the results of *JINC* after computing a dynamic variable order from a static MSR order to the dynamic order computed with the *i7-protoype*

[1] bestfit

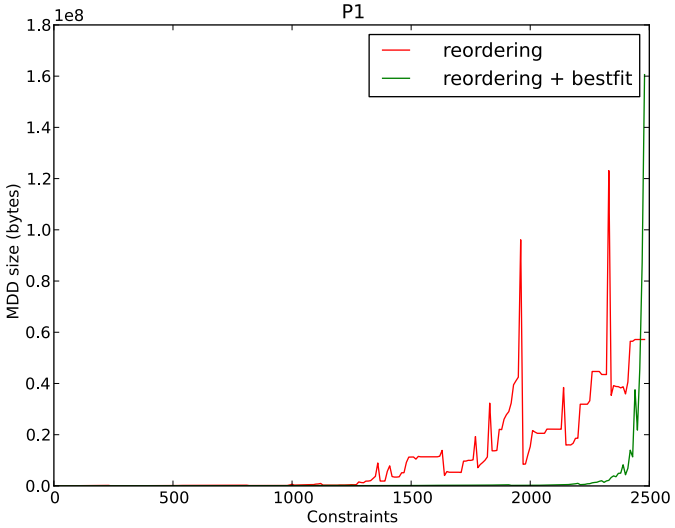


Figure 6.1 – Memory consumption for the test case P1 with TADD variable ordering, dynamic reordering and *node/branches hidden*

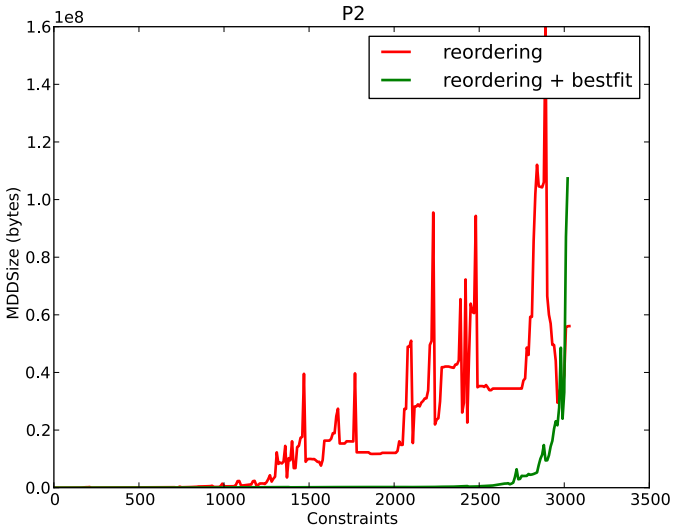


Figure 6.2 – Memory consumption for the test case P2 with TADD variable ordering, dynamic reordering and *node/branches hidden*

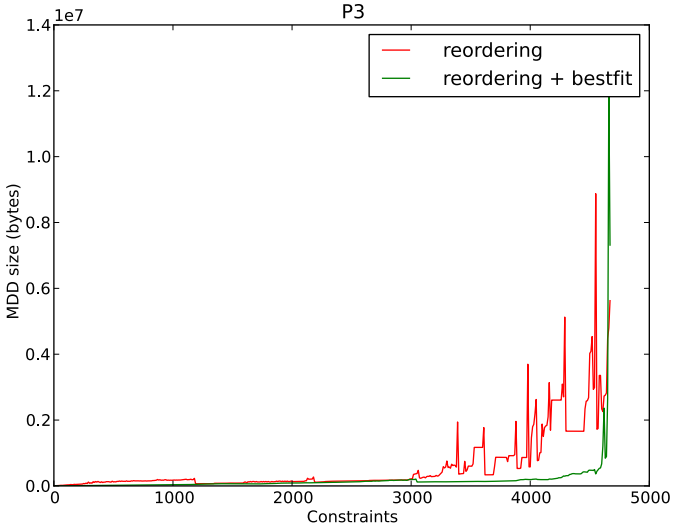


Figure 6.3 – Memory consumption for the test case P3 with TADD variable ordering, dynamic reordering and *node/branches hidden*

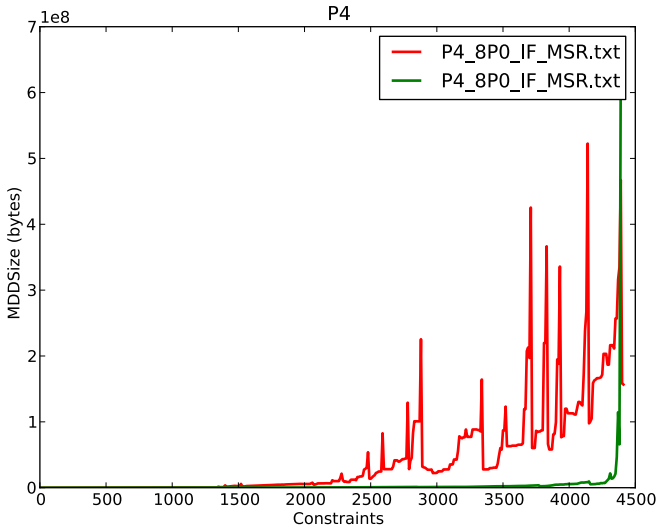


Figure 6.4 – Memory consumption for the test case P4 with TADD variable ordering, dynamic reordering and *node/branches hidden*

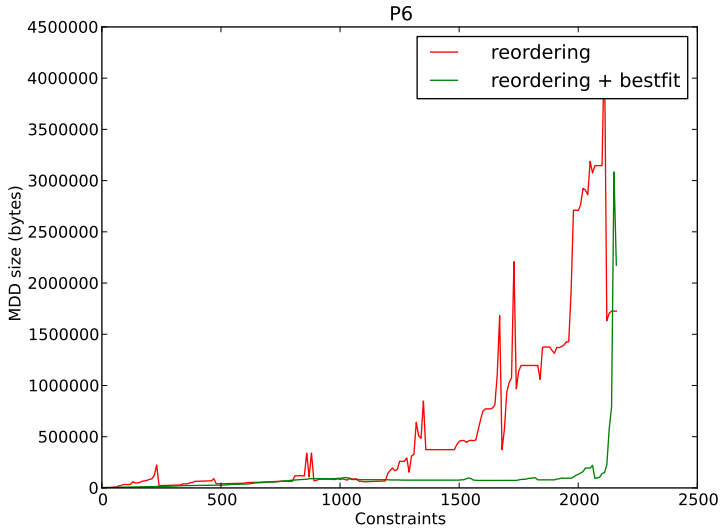


Figure 6.5 – Memory consumption for the test case P6 with TADD variable ordering, dynamic reordering and *node/branches hidden*

6.3 MDD Variants

The differences of the MDD variants can be seen in Table 6.3 for P6. It shows that for this example hiding of branches to a zero drain reduces the amount of needed branches for representing $W(R)$ by 53% when nodes are not hidden. When nodes are hidden, 57% of the branches can be saved. This reduction justifies the extra amount of memory that has to be used for saving an additional index, since a branch consists of an *integer* index(4 byte) and a pointer(8 byte). The compiler adds pad bytes (4 bytes) to get the alignment right. Therefore 3% (7%) of the bytes can be saved by using an index. For a quasi reduced MDDs additional bytes are saved, because nodes where all branches point to a zero drain disappear. Hiding redundant nodes reduces the amount of nodes by 14%. This effect is expected due to the low probabilities of such a case. The performance increases nevertheless with this rule, because the peak bytes count drops by 53%. The usage of the redundant node rule leads to a saving of 12% with branches not-hidden and 14% with branches-hidden. The redundant node rule did speed-up the building by around 30% with and without hiding branches. The overall speed-up from the worst to the best variant was 53%, which is mainly due to

the peak bytes dropping, which speed-up the building of $W(R)$. After this very detailed discussion for the two MDD variants now also branch compression is compared to the memory savings of the two variants. Times were not analyzed, because the implementation is prototypic and not as efficient as it could be. No big runtime differences should be expected, as only a few extra instructions for packing and unpacking are needed. There could be positive cache effects because the compression allows to store more information in the CPU caches. The memory usage for some examples with the *nothing hidden* and the *branches/nodes hidden* variant and *branches/nodes hidden + branch compression* variant can be seen in Figure 6.7 (P1) Figure 6.9 (P3). It can be seen that the byte usage drops by around 40% with branch compression. The effect reduces with an increasing number of constriants. Figure 6.11 (P4) and Figure 6.13 (P6).

node/branch	bytes	peak bytes	nodes	branches	times[1]
hide/hide	525312	1584656	6020	11762	15s (6s/9s)
-/hide	610256	3353816	6828	14244	22s (12s/10s)
hide/-	769816	1585736	6021	27042	18s (6s/12s)
-/-	875992	3355424	6985	30302	27s (13s/14s)

Table 6.3 – P6 with a precomputed dynamic variable ordering for the two MDD variants

[1] Unification / part check

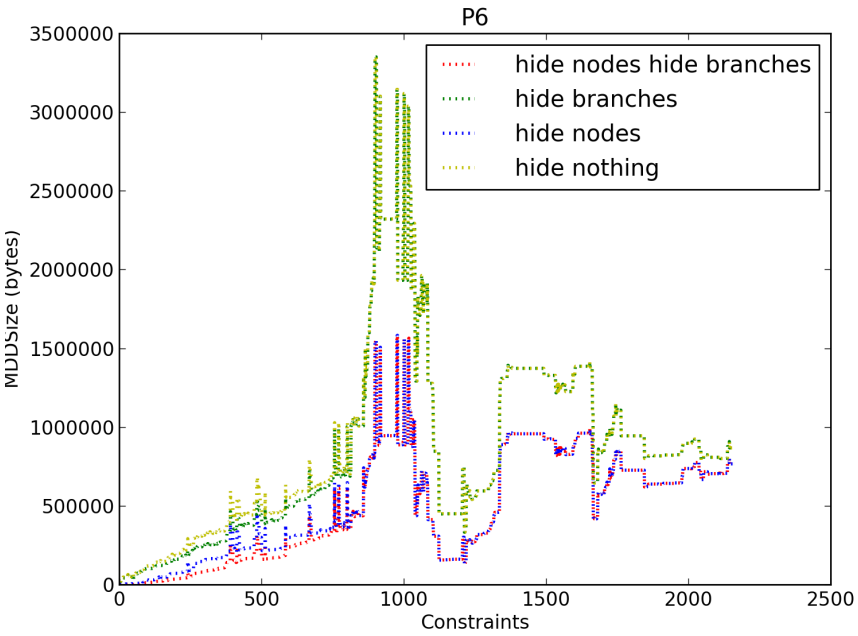


Figure 6.6 – Memory usage of the different variants for the test case P6 in comparison

node/branch	peak branches	peak bytes	branches end	bytes end
-/compress	51943	1952376	10514	550592
hide/compress	35668	780576	9706	492416
-/-	3353816	3353832	14244	610272
hide/-	86169	1584672	11762	525312

Table 6.4 – P6 with a precomputed dynamic variable ordering for node hiding/branch compression, branch hiding activated

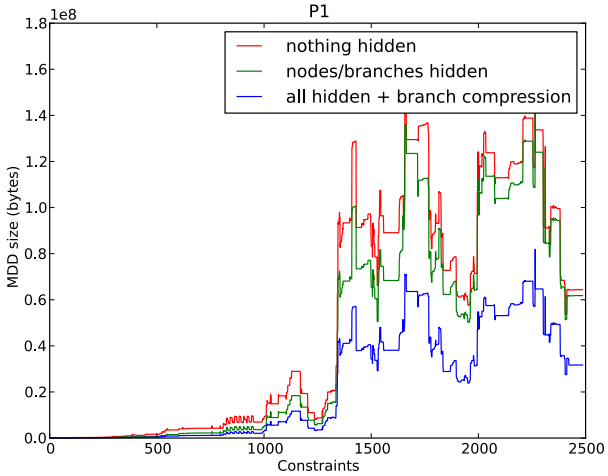


Figure 6.7 – Memory consumption for the test case P1 with precomputed dynamic variable ordering for the different MDD variants

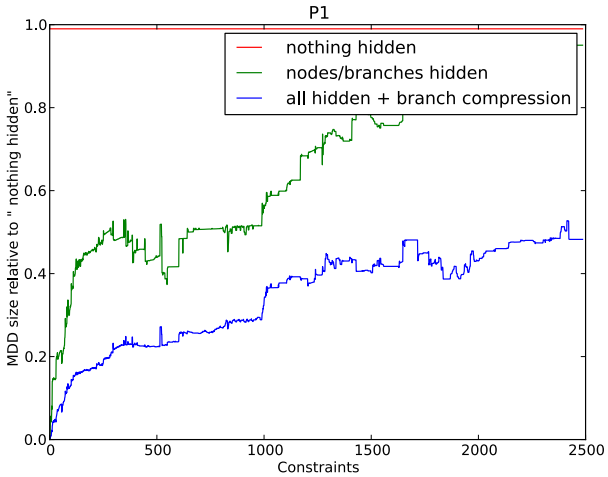


Figure 6.8 – Memory consumption for the test case P1 with precomputed dynamic variable ordering for the different MDD variants relative to the variant *nothing hidden*

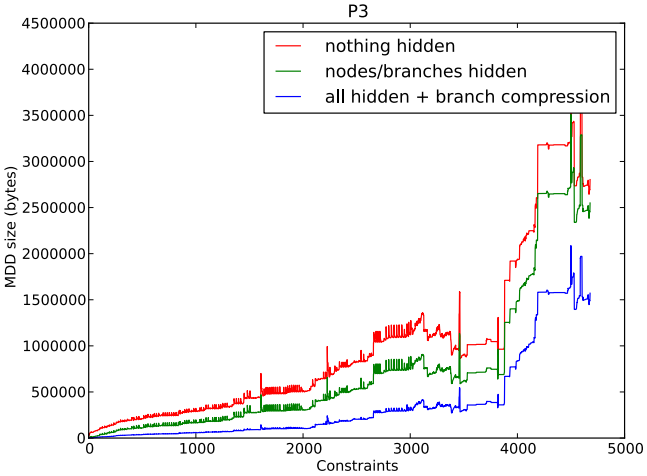


Figure 6.9 – Memory consumption for the test case P3 with precomputed dynamic variable ordering for the different MDD variants

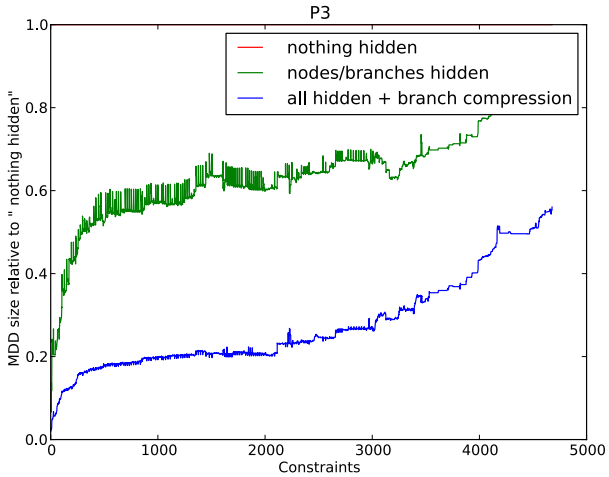


Figure 6.10 – Memory consumption for the test case P3 with precomputed dynamic variable ordering for the different MDD variants relative to the variant *nothing hidden*

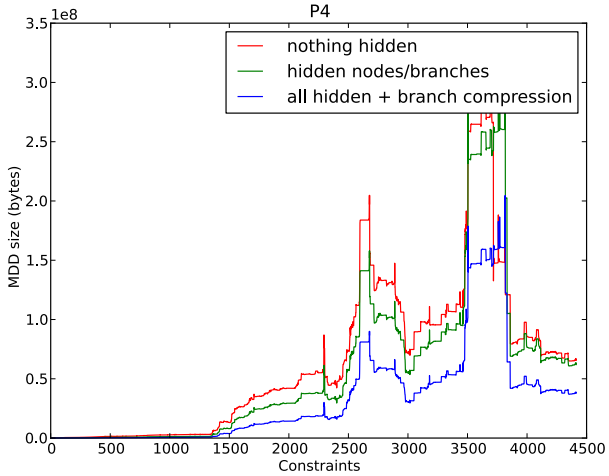


Figure 6.11 – Memory consumption for the test case P4 with precomputed dynamic variable ordering for the different MDD variants

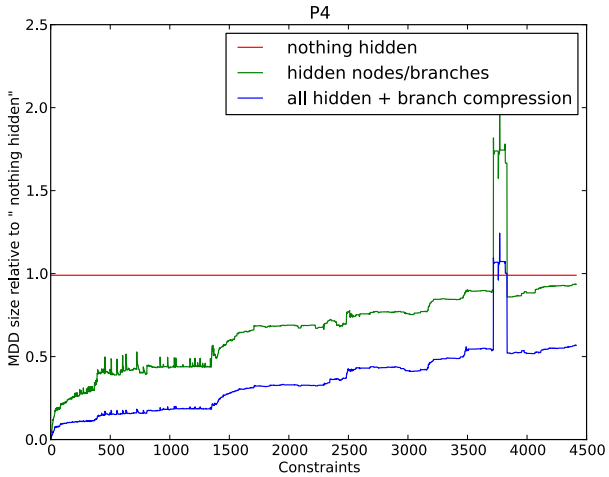


Figure 6.12 – Memory consumption for the test case P3 with precomputed dynamic variable ordering for the different MDD variants relative to the variant *nothing hidden*

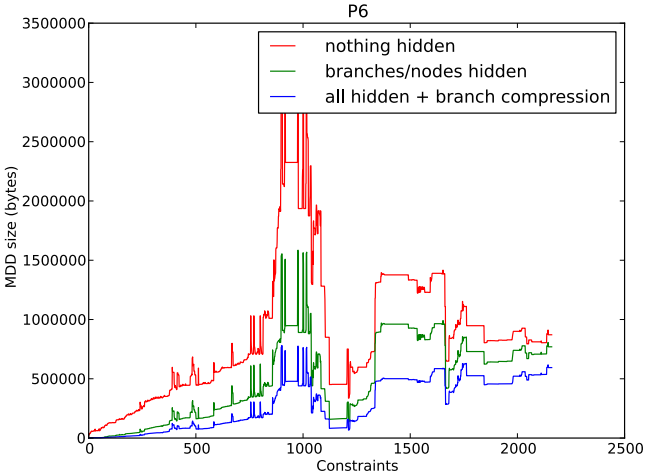


Figure 6.13 – Memory consumption for the test case P6 with precomputed dynamic variable ordering for the different MDD variants

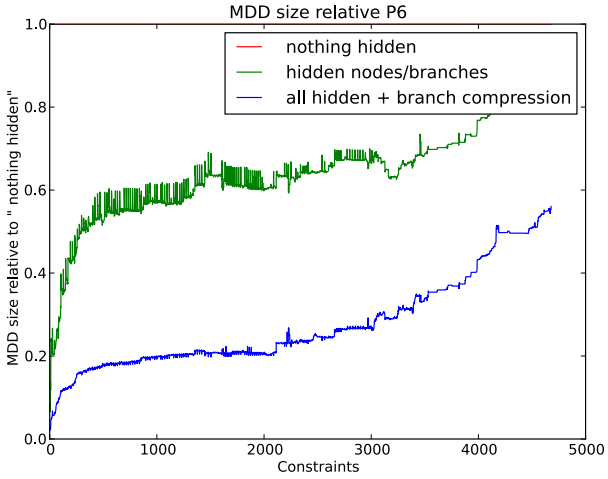


Figure 6.14 – Memory consumption for the test case P6 with precomputed dynamic variable ordering for the different MDD variants relative to the variant *nothing hidden*

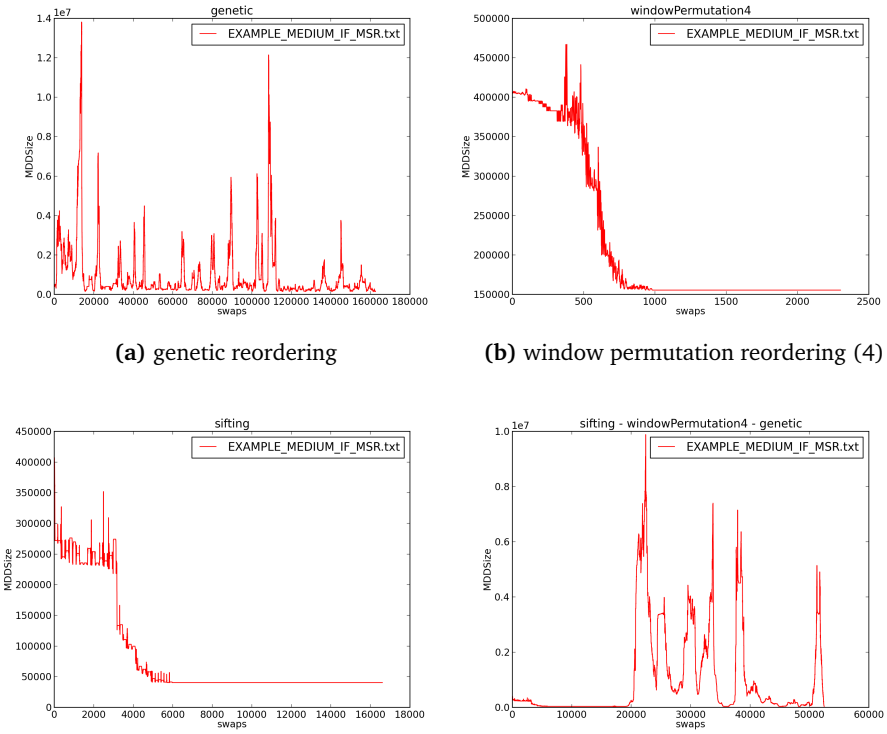
6.4 Reordering Techniques

As next performance factor, the implemented reordering techniques for JINCs were compared. For the memory usage after each swap see Figure 6.15. For

times and memory usage Table 6.5. It showed that sifting and window permutation requires a bit more memory at first than before the reordering, but then offer a reliable reduction and work fast. This is due to the fact that both algorithms are descent methods with different neighborhood relations. The straight line in the example is there due to the the reordering of empty levels, which does not influence the MDD size for fully reduced MDDs. Window permutation is a lot faster for bigger examples, since only window sizes up to 4 are available in JINC, but sifting gives the better results. Genetic reordering is not a descendant method, it is more like a scatter search. Therefore the search space is bigger. Local strategies like window permutation or sifting can get stuck in a local optimum. The genetic reordering in JINC was examined if it is applicable to the configuration problems. It showed that even for the medium example as well the runtime and also the memory consumption runtime exploded. It was not possible to apply genetic reordering on the bigger examples, as memory was exceeded. Also the resulting MDD size was sobering. The genetic algorithm rarely found better variable orderings than sifting or window permutation. The reason might be that evolutionary algorithms work well on problems where no knowledge about the structure of a system is available, but the MSR heuristic generates assumptions about the structure. The fundamental problem is the search space, as the genetic algorithm uses the complete search space, which is too big to handle in the product configuration context. There are variable orderings that won't fit into the memory. Therefore a approach with a static variable ordering fits better.

algorithm	bytes	peak bytes	nodes	branches	time
genetic	155640	13821448	1343	5028	70s
genetic(2nd run)	126560	13821448	1154	3871	59s
sifting	38520	407360	387	1054	2s
windowPermute4	60560	467240	572	1784	2s
all combined	38544	9888608	372	1108	58s

Table 6.5 – Medium testcase, different minimization techniques



(a) genetic reordering

(b) window permutation reordering (4)

(c) sifting reordering

(d) all techniques combined

Figure 6.15 – Different reordering techniques in comparison, memory usage after swaps

6.5 Parallel Bestfit

Figure 6.16 and Table 6.6 show the speed-up achieved with the concurrent bestfit implementation for the test case P6 with a precomputed dynamic ordering, so no reordering was necessary. Measured was only the bestfit execution, garbage collection was not included. See Figure 6.17 for bestfit with garbage collection times. Notable is, that even for one thread, the performance increases with more jobs. It is very likely that the reduction of garbage collection calls causes this effect, because garbage collection cleans the computed table. Nevertheless, garbage collection is absolutely necessary for bestfit, as a huge number of temporary nodes is produced, however it shows, that the garbage collection strategy could be improved. The best speed-up for this test case in comparison to the single threaded run with the same number of jobs was factor 2.7, which could be achieved with 5-6

threads with 64 jobs. Job sizes bigger than 8 did not increase the performance significantly. Concluded can be said that JINC does not scale for more than 6 processors. Different explanations for this behavior will be given later.

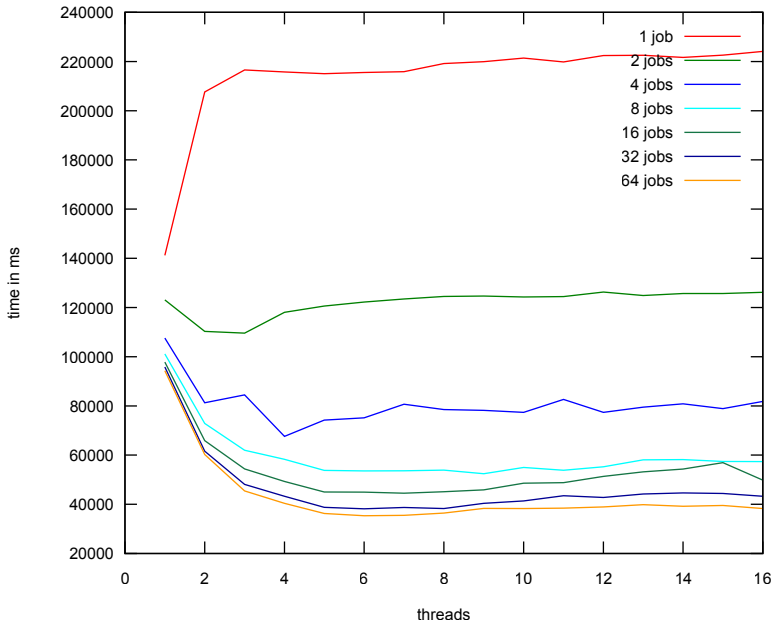


Figure 6.16 – Calculation time for the concurrent bestfit implementation for the testcase P6

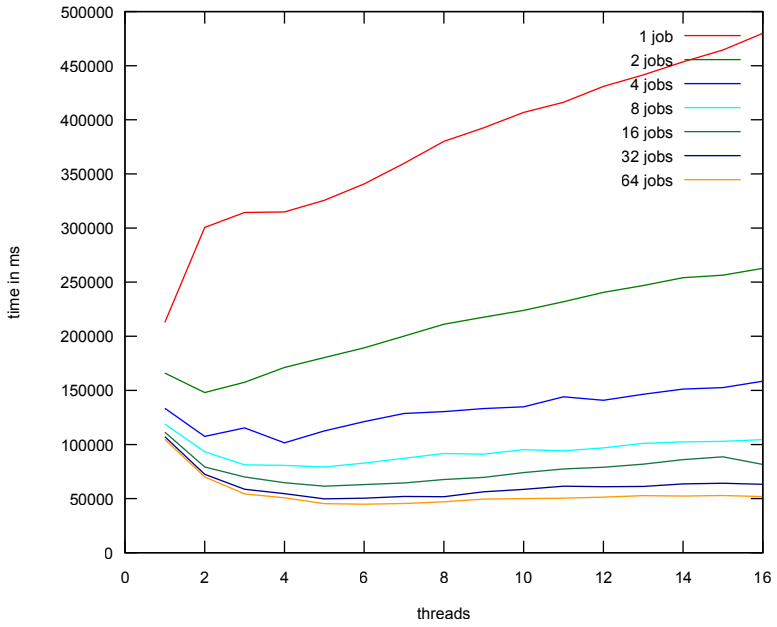


Figure 6.17 – Calculation time for the concurrent *bestfit* implementation with *garbage collection* included in the measurement of the test case P6

6.6 Parallel Parts Check

In this test, the speed-up for the parallel parts check was measured. As test cases the examples P3 and P6 were chosen. The tests were executed for 1-16 threads. The result can be seen in Figure 6.18 (P3) and Figure 6.19 and Table 6.6. The best speed-up for P3 was achieved for 5 threads and is 1.9 times the duration for one thread. For P6 the best speed-up was achieved with 6 threads and was also 1.9 times the duration for one thread. Garbage collection execution was included in the measuring, but should not have influenced the result significantly, as it is only executed after 1012 jobs were finished. Therefore also unique table clearing should not influence the performance much.

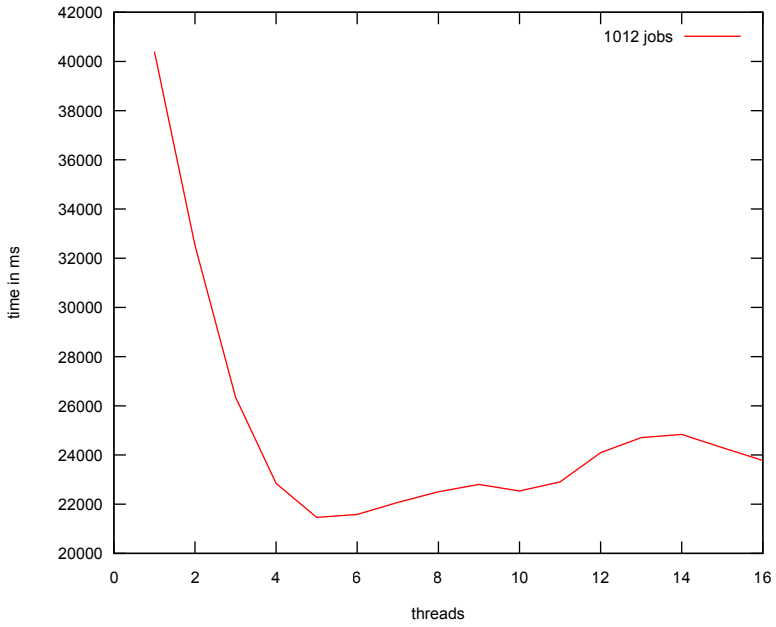


Figure 6.18 – Calculation time for the concurrent *parts check* for P3

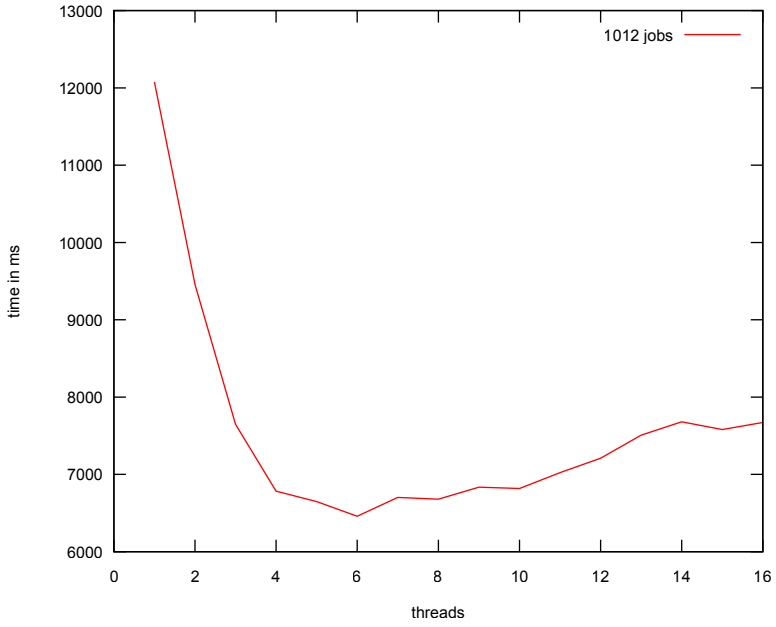


Figure 6.19 – Calculation time for the concurrent *parts check* for P6

#[1]	1 job	2 jobs	4 jobs	8 jobs	16 jobs	32 jobs	64 jobs
1	141	123	107	101	97	95	94
2	207	110	81	72	65	61	60
3	216	109	84	61	54	48	45
4	215	118	67	58	49	43	40
5	215	120	74	53	45	38	36
6	215	122	75	53	44	38	35
7	215	123	80	53	44	38	35
8	219	124	78	53	45	38	36
9	219	124	78	52	45	40	38
10	221	124	77	54	48	41	38
11	219	124	82	53	48	43	38
12	222	126	77	55	51	42	38
13	222	124	79	58	53	44	39
14	221	125	80	58	54	44	39
15	222	125	78	57	56	44	39
16	224	126	81	57	49	43	38

Table 6.6 – Times in millisecond for *bestfit* in parallel with P6 with and a precomputed dynamic variable ordering, for different number of threads and job sizes, without garbage collection

[1] Number of threads

6.7 Multi Operand Apply

Another feature of JINC is the *multi operand apply* algorithm which is, according to Ossowskis experiments [10], faster and more space efficient than the normal two operand *apply*. This method calculates the BDD for several BDD-operations in one step. Since there are no interim results, the number of created nodes is lower. Figure 4.5 shows three example MDDs that should be unified. Figure 4.6 shows how MDD multi operand apply works. Every variable configuration is followed, the appropriate nodes in the respective MDDs are chosen. In the example, the drain for every variable is reached, so that the operator can decide that for the 0-branch 1 should be returned as node, since 1 unified zero gives 1, equally for the other branches. Ossowski didn't use any computed table in his work. In this thesis, the multi operand

P3		P6	
threads	time	threads	time
1	40	1	120
2	32	2	94
3	26	3	76
4	22	4	67
5	21	5	66
6	21	6	64
7	22	7	67
8	22	8	66
9	22	9	68
10	22	10	68
11	22	11	70
12	24	12	72
13	24	13	75
14	24	14	76
15	24	15	75
16	23	16	76

Table 6.7 – Times in seconds for P3 and P6 for the concurrent parts check

apply algorithm was extended for MDDs. Since the runtime was extremely high, computed table entries were added. The resulting computation time for the medium example can be seen in Table 6.8 and memory costs in Figure 6.20. It shows, that multi operand apply is not a reasonable solution for this use case, standard procedures are faster.

multi operand apply	time
3 operands w/o caches	timeout
2 operands	1034ms
5 operands	351ms
15 operands	281ms
30 operands	925ms
60 operands	3638ms
120 operands	38857ms

Table 6.8 – Medium testcase, multi operand apply, different operand sizes

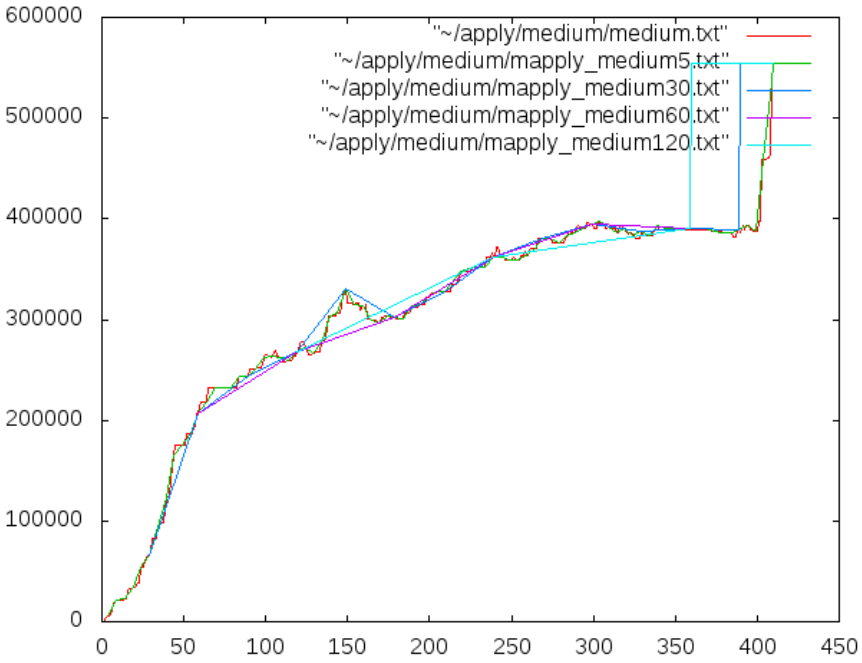


Figure 6.20 – Memory usage for *multi operand apply* on the medium example for different operand sizes

Chapter 7

Related & Future Work

In this section, some work is outlined that can be done in the future, and similar work is discussed. For the parallel computing of MDDs there are some approaches that could be examined. The results for the parallelization of not depended operators (*part check*, *bestfit*) showed that even for a big enough job count, the speed does not scale for more than 5 processors. There are two points in the current implementation where synchronization is necessary. The first point is, the refilling of the job queue (then also garbage collection is performed), and the second is unique table insertions. Experiments with a lock free algorithm did not show any significant improvement for the unique table synchronization. For the part check, the job queue contained 1012 jobs. Therefore it can be assumed, that the synchronization time is not relevant. Since the computed table is thread local, results achieved are not shared between the threads. Work may therefore be redundant. The probability of redundant work execution increases with the number of threads. Future work could be implementing a shared computed table for JINC to examine the amount of work that is done redundant. Another effect that has negative influence on the parallelization is the fact, that work, that is not yet inserted into a computed table is possibly done redundant in parallel. This effect occurs even for a shared computed table. It might make sense to consider techniques to prevent this. A shared computed table would also allow the usage of external thread management, for example *cilk*. Advantages of fine granular implementations with *cilk* are that parallelization can be used even for dependent operators. Also work stealing is implemented, so that no thread has to wait for another if work is available. For non dependent operators like the bestfit case, JINCs approach is probably more efficient, as the the work

packages are bigger and therefore the overhead is lower. Another task would be to consider horizontal parallelization for the swapping of levels, as for now only vertical parallel implementations exist. Vertical means in this context that a complete swap operation is done as one job. Swaps for two levels can be done in parallel if the distance is higher than one. A horizontal swap would be especially interesting for MDDss, as the expense of swapping of a single node can increase highly in comparison to BDDs. First investigations showed a critical section, but a correct implementation would be future work. Thomas van Dijk parallelized in his Master's thesis the BDD implementation "Sylvan" of the University of Twente. The work was done on a non-uniform memory access (NUMA) architecture, which differs from a symmetric multiprocessing (SMP) architecture, as some memory regions are on different buses and therefor access delays are not equal. The system had 48 cores. The speed-up was in the best cases up to $0.5 \cdot \#\text{workers}$, the maximum achieved speed-up around factor 18 and the test cases with the worst speed-up achieved 2-3 speed-up compared to one worker. In comparison to BuDDy the speed-up was between 0.4 and 8.

Lock free data structures were used and results from different threads were shared, but the computed table is *lossy*. To avoid the extra work for the lock free algorithm in case of a collision, the entry for the computed table is lost. Tom Dijk stated, that the redundant work is negligible in his case. His lock free algorithms for the unique table require extra work in case of a collision. The spinlock JINC uses for the unique table also may require extra work, but prevents an expensive context switch. It is also only necessary to synchronize concurrent writing access. It could also be replaced by a lock free algorithm, but experiments showed that no relevant speed-up can be achieved, probably because a concurrent hash bucket write access is not very likely and the used lock free algorithm required extra work in case of a concurrent access.
Ossowski did only present one speed-up comparison for the parallel execution with JINC. The test was a matrix power calculation. The parallel execution was divided in different steps, where each step doubled the concurrent matrices multiplication. Garbage collection was executed after every step. The results showed for 16 processors and 16 operations in parallel a speedup of factor 5.22. The results for less operators were around 2.5 for 8 operators, 1.25 for 4 operators. Therefore the results are better than the here achieved

speed-up. It could be examined if his test case in special is faster or generally BDD calculations. A possible explanation could be that in his case few redundant work is done, whereas for configuration problems the computed table is more important. Comparison to a i7-chair prototype showed, that the implemented reordering techniques in JINC are not optimal for the product configuration problems. Also the MDD specific algorithms, like the swapping could be implemented faster. The implementation developed in this thesis on the other hand offers data structures with higher data compression, which also can speed-up the calculation. Another possible task would be to consider an intelligent allocation strategy for the branches. Although this is not as important as the node allocation strategy, since branches are not reused, performance could possibly be improved and memory fragmentation reduced.
The unification process currently uses a lot of memory, because the intermediate MDDs have huge sizes. Nevertheless it was showed, that the resulting MDDs can compare with a prototype from the university chair, further optimizations could improve the resulting MDD. The size of the resulting MDD dependends on several factors, like the decision how to split the formula for $\overline{W}(R)$ and the reordering methods. To improve the building process, the reordering algorithms of JINC could be extended. For example, window permutation with a bigger window or sifting for a limited number of levels could be implemented. Fine tuning the different parameters, like the threshold or the sifting abortion criterion could also improve the performance.

Chapter 8

Conclusion

In this thesis, an insight into product configuration problems and the application of MDDs was given. Two MDD variants were compared, and it was showed that fully reduced MDDs give a noticeable speed-up whereas the size only reduces marginally, but intermediate sizes are noticeable reduced. Sparse storage on the other hand offers little improvement for fully reduced MDDs, the introduced branch compression is superior for the automotive context. Branch compression can be used together with the sparse representation without any extra effort and is easy to implement. It was shown that with the use of parallelism, the runtime of the used algorithms can be reduced, but also the limits of JINC were shown and basic approaches were given to solve the problems. It further was shown that the JINC features *multi operand apply* and the *genetic reordering* algorithm are not well fitted for configuration problems. When working with MDDs, the here performed tests might help to decide which methods should be chosen.

List of Acronyms

Acronyms

ADD algebraic decision diagrams

BDD binary decision diagram

CAD computer-aided design

DAG directed acyclic graph

HOWDD higher-order weighted decision diagram

MDDs multi-valued decision diagram

TADD mass spring relaxation

NUMA non-uniform memory access

ROBDD reduced order decision diagram

ROMDD reduced order multi-valued decision diagram

SMP symmetric multiprocessing

TADD toggling algebraic decision diagram

TADD zero-suppressed decision diagram

List of Figures

List of Tables

Bibliography

[1] H. Ford and S. Crowther, *My Life and Work*, ser. Library of American civilization. Doubleday, Page, 1922. [Online]. Available: http://books.google.de/books?id=4K82efXzn10C

[2] J. P. MacDuffie, K. Sethuraman, and M. L. Fisher, "Product variety and manufacturing performance: evidence from the international automotive assembly plant study," *Management Science*, vol. 42, no. 3, pp. 350–369, 1996.

[3] G. Ciardo, "Data representation and efficient solution: A decision diagram approach," in *Formal Methods for Performance Evaluation*, ser. Lecture Notes in Computer Science, M. Bernardo and J. Hillston, Eds. Springer Berlin Heidelberg, 2007, vol. 4486, pp. 371–394. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-72522-0_9

[4] Wikipedia, "Binary decision diagram — wikipedia, the free encyclopedia," 2013, [Online; accessed 15-April-2014]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Binary_decision_diagram&oldid=585184630

[5] K. O. Tinmaung, D. Howland, and R. Tessier, "Power-aware fpga logic synthesis using binary decision diagrams," in *Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays*, ser. FPGA '07. New York, NY, USA: ACM, 2007, pp. 148–155. [Online]. Available: http://doi.acm.org/10.1145/1216919.1216945

[6] R. Bryant, "Binary decision diagrams and beyond: enabling technologies for formal verification," in *Computer-Aided Design, 1995. ICCAD-95. Digest of Technical Papers., 1995 IEEE/ACM International Conference on*, Nov 1995, pp. 236–243.

[7] S.-i. Minato, *Binary Decision Diagrams and Applications for VLSI CAD*. Norwell, MA, USA: Kluwer Academic Publishers, 1996.

[8] [Online]. Available: www.configit-software.com/research

[9] R. Berndt, P. Bazan, and K.-S. Hielscher, "Mdd-based verification of car manufacturing data," in *Computational Intelligence, Modelling and Simulation (CIMSiM), 2011 Third International Conference on*, Sept 2011, pp. 187–193.

[10] J. Ossowski, "Jinc: a multi-threaded library for higher-order weighted decision diagram manipulation," Ph.D. dissertation, University of Bonn, 2010.

[11] Wikipedia, "Boole's expansion theorem — wikipedia, the free encyclopedia," 2014.

[12] C. Y. Lee, "Representation of switching circuits by binary-decision programs," *Bell System Technical Journal*, vol. 38, no. 4, pp. 985–999, 1959. [Online]. Available: http://dx.doi.org/10.1002/j.1538-7305.1959.tb01585.x

[13] R. Bryant, "Graph-based algorithms for boolean function manipulation," *Computers, IEEE Transactions on*, vol. C-35, no. 8, pp. 677–691, Aug 1986.

[14] Wikipedia, "Function (mathematics) — wikipedia, the free encyclopedia," 2014, [Online; accessed 16-March-2014]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Function_(mathematics)&oldid=598293407

[15] E. Dubrova, "Advanced logic design," p. 12, lecture 4. [Online]. Available: http://web.it.kth.se/~dubrova/LScourse/LECTURES/lecture4.pdf

[16] D. Miller and R. Drechsler, "On the construction of multiple-valued decision diagrams," in *Multiple-Valued Logic, 2002. ISMVL 2002. Proceedings 32nd IEEE International Symposium on*, 2002, pp. 245–253.

[17] S. Minato, "Zero-suppressed bdds for set manipulation in combinatorial problems," in *Design Automation, 1993. 30th Conference on*, June 1993, pp. 272–277.

[18] Wikipedia, "Sparse matrix — wikipedia, the free encyclopedia," 2014, [Online; accessed 29-March-2014]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Sparse_matrix&oldid=597834343

[19] ——, "Run-length encoding — wikipedia, the free encyclopedia," 2014, [Online; accessed 9-April-2014]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Run-length_encoding&oldid=594084869

[20] ——, "Address space layout randomization — wikipedia, the free encyclopedia," 2014, [Online; accessed 22-April-2014]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Address_space_layout_randomization&oldid=604914589

[21] T. Wang, "Integer hash function." [Online]. Available: http://burtleburtle.net/bob/hash/integer.html;http://web.archive.org/web/20071223173210/http://www.concentric.net/~Ttwang/tech/inthash.htm

[22] [Online]. Available: http://research.neustar.biz/2011/12/29/choosing-a-good-hash-function-part-2/,http://research.neustar.biz/2012/02/02/choosing-a-good-hash-function-part-3/

[23] Google, "Cityhash." [Online]. Available: https://code.google.com/p/cityhash/

[24] [Online]. Available: http://programmers.stackexchange.com/questions/49550/which-hashing-algorithm-is-best-for-uniqueness-and-speed

[25] B. Bollig and I. Wegener, "Improving the variable ordering of obdds is np-complete," *IEEE Trans. Comput.*, vol. 45, no. 9, pp. 993–1002, Sep. 1996. [Online]. Available: http://dx.doi.org/10.1109/12.537122

[26] M. Rice and S. Kulhari, "A survey of static variable ordering heuristics for efficient bdd/mdd construction," *University of California, Tech. Rep*, 2008.

[27] R. Berndt, P. Bazan, K.-S. Hielscher, and R. German, "Construction methods for mdd-based state space representations of unstructured systems," in *Measurement, Modelling, and Evaluation of Computing*

Systems and Dependability and Fault Tolerance, ser. Lecture Notes in Computer Science, K. Fischbach and U. Krieger, Eds. Springer International Publishing, 2014, vol. 8376, pp. 43–56. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-05359-2_4

[28] M. Fujita, Y. Matsunaga, and T. Kakuda, "On variable ordering of binary decision diagrams for the application of multi-level logic synthesis," in *Proceedings of the Conference on European Design Automation*, ser. EURO-DAC '91. Los Alamitos, CA, USA: IEEE Computer Society Press, 1991, pp. 50–54. [Online]. Available: http://dl.acm.org/citation.cfm?id=951513.951525

[29] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," in *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*. IEEE Computer Society Press, 1993, pp. 42–47.

[30] R. Drechsler, B. Becker, and N. Gockel, "Genetic algorithm for variable ordering of obdds," *Computers and Digital Techniques, IEE Proceedings -*, vol. 143, no. 6, pp. 364–368, Nov 1996.

[31] W. Lenders and C. Baier, "Genetic algorithms for the variable ordering problem of binary decision diagrams," in *Foundations of Genetic Algorithms*, ser. Lecture Notes in Computer Science, A. Wright, M. Vose, K. Jong, and L. Schmitt, Eds. Springer Berlin Heidelberg, 2005, vol. 3469, pp. 1–20. [Online]. Available: http://dx.doi.org/10.1007/11513575_1

[32] N. Narodytska and T. Walsh, "Constraint and variable ordering heuristics for compiling configuration problems," in *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, ser. IJCAI'07. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007, pp. 149–154. [Online]. Available: http://dl.acm.org/citation.cfm?id=1625275.1625298

[33] Y. He. [Online]. Available: http://software.intel.com/en-us/articles/multicore-enabling-a-binary-decision-diagram-algorithm

[34] J. Lind-Nielsen, "buddy." [Online]. Available: http://sourceforge.net/projects/buddy/

[35] T. van Dijk, A. Laarman, and J. van de Pol, "Multi-core {BDD} operations for symbolic reachability," *Electronic Notes in Theoretical Computer Science*, vol. 296, no. 0, pp. 127 – 143, 2013, proceedings the Sixth International Workshop on the Practical Application of Stochastic Modelling (PASM) and the Eleventh International Workshop on Parallel and Distributed Methods in Verification (PDMC). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066113000406

[36] M. Hilbert and P. López, "The world's technological capacity to store, communicate, and compute information," *Science*, vol. 332, no. 6025, pp. 60–65, 2011.

[37]