Preprint - please cite: C. Vidal Silva, A. Felfernig, J. Galindo, M. Atas, and D. Benavides, A Parallelized Variant of Junker's QuickXPlain Algorithm, 25th International Symposium on Methodologies for Intelligent Systems, Graz, Austria, pp. 457–468, LNAI, 12117, 2020.

A Parallelized Variant of Junker's QUICKXPLAIN Algorithm

Cristian Vidal Silva[1], Alexander Felfernig[2],
José A. Galindo[1], Müslüm Atas[2], and David Benavides[1]

[1] Catholic University of the North cristian.vidal@ucn.cl
[2] University of Sevilla {benavides, jagalindo}@us.es
[3] Graz University of Technology {afelfernig, muatas}@ist.tugraz.at

Abstract. Conflict detection is used in many scenarios ranging from interactive decision making to the diagnosis of potentially faulty hardware components or models. In these scenarios, the efficient identification of conflicts is crucial. Junker's QUICKXPLAIN is a divide-and-conquer based algorithm for the determination of preferred minimal conflicts. Motivated by the increasing size and complexity of knowledge bases, we propose a parallelization of the original algorithm that helps to significantly improve runtime performance especially in complex knowledge bases. In this paper, we introduce a parallelized version of QUICKXPLAIN that is based on the idea of predicting and executing parallel consistency checks needed by QUICKXPLAIN.

1 Introduction

Conflict detection is used in many applications of constraint-based representations (and beyond). Examples thereof are *knowledge-based configuration* [13] where users define requirements and conflict detection is in charge of figuring out minimal sets of potential changes to the given requirements in order to restore consistency (if the configurator is not able to identify a solution), *recommender systems* [3,10], and many other applications of model-based diagnosis [9]. Especially in interactive settings, there is often a need of identifying preferred conflicts [7,11], for example, users of a car configurator or a camera recommender who have strict preferences regarding the upper price limit, are more interested in relaxations related to technical features (e.g., related to the availability of a skibag in a car or a wide-aperture lens in a pocket camera).

Conflict detection helps to find combinations of constraints in the knowledge base that are responsible for an inconsistency. QUICKXPLAIN is such a conflict detection algorithm which is frequently used and works for constraint-based representations, description logics, and SAT solvers [7]. The algorithm is based on a divide-and-conquer approach where consistency analysis operations are based on the division of a constraint set $C = \{c_1..c_m\}$ into two subsets $C_a = \{c_1..c_k\}$ and $C_b = \{c_{k+1}..c_m\}$ assuming, for example, $k = \lfloor \frac{m}{2} \rfloor$. If C_b is *inconsistent*, the consideration set C can be reduced by half since C_a must not be analyzed anymore (at least one conflict exists in C_b). Depending on the QUICKXPLAIN variant, either C_a or C_b is checked for consistency.

Conflict detection is typically applied in combination with conflict resolution which helps to resolve all existing conflicts. In this context, the minimality (irreducibility) of conflict sets is important since this allows to resolve each conflict by simply deleting one of the elements in the conflict set. The elements to be deleted to restore global consistency are denoted as *hitting set* and can, for example, be determined on the basis of

a hitting set directed acyclic graph [9]. Due to the increasing size and complexity of knowledge bases, there is an increasing need to further improve the performance of solution search and conflict detection / hitting set calculation [2,4,5,8]. Parallelizations of algorithms in these scenarios have been implemented in different contexts. Approaches to parallelization have, for example, been proposed on the *reasoning level* [2] where the determination of a solution is based on the idea of identifying subproblems which can be solved to some degree independently by the available cores. Due to today's multi-core CPU architectures, such parallelization techniques become increasingly popular in order to be able to better exploit the offered computing resources.

A similar motivation led to the development of parallelization techniques in model-based diagnosis [9]. J. Marques-Silva. et al [6] propose a parallelization approach for hitting set determination where Reiter's approach to model-based diagnosis is parallelized by a level-wise expansion of a breadth-first search tree with the goal of computing minimal (cardinality) diagnoses. On each level, (minimal) conflict sets are determined in parallel, however, the determination of individual conflict sets is still a sequential process (based on QUICKXPLAIN [7]). In diagnosis search, the efficient determination of minimal conflicts is a core requirement [6]. Especially in constraint-based reasoning scenarios, the identification of minimal conflict sets is frequently based on QUICKXPLAIN [7]. Compared to iterative approaches of removing elements from inconsistent constraint sets [1], QUICKXPLAIN follows a divide-and-conquer strategy that helps to reduce the number of needed consistency checks. Although the algorithm is often used in interactive settings with challenging runtime requirements, up-to-now no parallelized version has been proposed. In this paper, we propose an algorithm that enables efficient parallelized minimal conflict detection and thus helps a.o. to significantly improve the runtime performance of conflict detection in interactive applications.

The contributions of this paper are the following. *First*, we show how to parallelize conflict detection with look-ahead strategies that scale with the number of available computing cores. *Second*, we show how to integrate our approach with the QUICKXPLAIN algorithm that is often used in constraint-based applications. *Third*, we show the applicability and improvements of our approach on the basis of performance evaluations. Finally, we point out in which way the proposed approach can help to improve the performance of existing diagnosis approaches. The remainder of the paper is organized as follows. In Section 2, we introduce the basic idea of Junker's QUICKXPLAIN using a working example. Thereafter, in Section 3 we introduce a parallelized variant of the algorithm. In Section 4, we analyze the proposed approach and report the results of a performance evaluation which shows significant improvements compared to standard QUICKXPLAIN. The paper is concluded with Section 5.

2 Calculating Minimal Conflicts

In the remainder of this paper, we introduce our approach to parallelized conflict detection on the basis of constraint-based knowledge representations [14]. A *conflict set* can be defined as a minimal set of constraints that is responsible for an inconsistency, i.e., a situation in which no solution can be found for a given constraint satisfaction problem (CSP) (see Definitions 1–2).

Definition 1. *A* Constraint Satisfaction Problem *(CSP) is a triple (V,D,C) with a set of variables* $V = \{v_1..v_n\}$*, a set of domain definitions* $D = \{dom(v_1)..dom(v_n)\}$*, and a set of constraints* $C = \{c_1..c_m\}$.

Definition 2. *Assuming the inconsistency of C, a* conflict set *can be defined as a subset $CS \subseteq C : CS$ is inconsistent. CS minimal if $\neg\exists CS' : CS' \subset CS$.*

Examples of a CSP and a conflict set are the following (see Examples 1–2).

Example 1. An example of a CSP for car configuration is the following: $V = \{cartype, fuel, pdc, color, skibag\}$, $D = \{dom(cartype) = [s, c], dom(fuel) = [p, d], dom(pdc) = [y, n], dom(color) = [b, r, g], dom(skibag) = [y, n]\}$, and $C = \{c_1 : fuel = d, c_2 : skibag = y, c_3 : color = b, c_4 : cartype = c, c_5 : pdc = y, c_6 : skibag = y \rightarrow cartype = s, c_7 : cartype = c \rightarrow color = b\}$.

Note that PCD refers to the Park Distance Control implemented in recent vehicles It is convenient to distinguish between a *consistent background knowledge* B of constraints that cannot be relaxed (in our case, $B = \{c_6, c_7\}$) and a *consideration set* C of relaxable constraints (in our case, requirements $C = \{c_1..c_5\}$).

Example 2. In Example 1, there is one minimal conflict which is $CS = \{c_2, c_4\}$, since $\{c_2, c_4\} \subseteq C$ and inconsistent (CS). Furthermore, CS is minimal since there does not exist a CS' s.t. $CS' \subset CS$. The execution trace of QUICKXPLAIN for this working example is depicted in Figure 1.

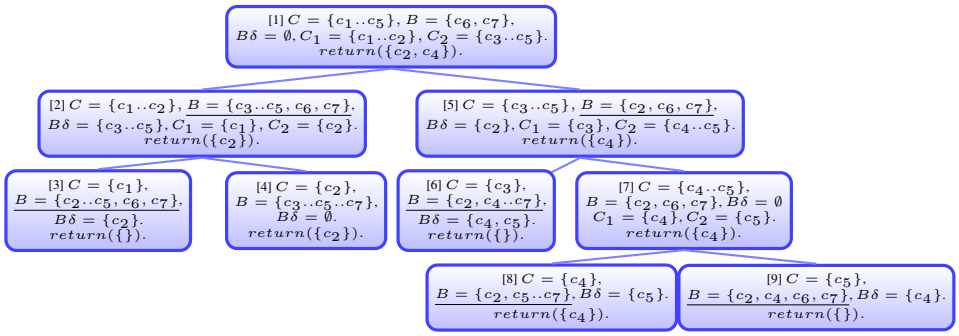

Fig. 1: QX execution trace for $C = \{c_1..c_5\}$ and $B = \{c_6, c_7\}$ assuming a minimal conflict set $CS = \{c_2, c_4\}$. Underlined Bs denote QX consistency checks. For example, in the incarnation [2] of the QX function, the consistency check activated is $\{c_3..c_5, c_6, c_7\}$.

QUICKXPLAIN [7] (a variant is shown in Algorithms 1 and 2) supports the determination of minimal (irreducible) conflicts in a given set of constraints (C). QUICKXPLAIN is activated if the background knowledge B (often assumed to be empty or a consistent set of constraints) is *inconsistent* with the set of constraints C (we assume this consistency check to be performed by a direct solver call). The core algorithm is implemented in the function QX (Algorithm 2) that determines a minimal conflict which is a minimal subset of the constraints in C with the conflict set property.

The function QX (Algorithm 2) focuses on isolating those constraints that are part of a minimal conflict. If C includes only one element ($C = \{c_\alpha\}$), this element can be considered as element of the conflict - this is due to the invariant property *inconsistent*($C \cup B$). If the B is consistent and C has more than one element, C is divided into two separate sets, where (in our QX variant) the second part (C_b) is added to B in order to analyse further elements of the conflict. The function QX activates a consistency

check (INCONSISTENT) to figure out whether the considered background knowledge is inconsistent, i.e., no solution exists. $B\delta$ indicates constraints added to B.

Algorithm 1 QUICKXPLAIN$(C, B) : CS$

```
1: if CONSISTENT(B ∪ C) then
2:     return('no conflict')
3: else if C = ∅ then
4:     return(∅)
5: else
6:     return(QX(C, B, ∅))
7: end if
```

Algorithm 2 QX$(C = \{c_1..c_m\}, B, B\delta) : CS$

```
1: if Bδ ≠ ∅ and INCONSISTENT(B) then
2:     return(∅)
3: end if
4: if C = {c_α} then
5:     return({c_α})
6: end if
7: k = ⌊m/2⌋
8: C_a ← c_1...c_k; C_b ← c_{k+1}...c_m;
9: Δ_2 ← QX(C_a, B ∪ C_b, C_b); Δ_1 ← QX(C_b, B ∪ Δ_2, Δ_2);
10: return(Δ_1 ∪ Δ_2)
```

In many of the mentioned application scenarios, there exists an exponential number of conflicts and ways to resolve a conflict [7]. Especially in interactive scenarios, it is extremely important to identify *preferred conflicts*, i.e., conflicts with a high probability of being the basis of a relaxation acceptable for the user. For example, if a user is strongly interested in low-priced digital cameras, a conflict set that includes a price limit might be of low relevance for the user (since the user is not willing to change the price limit). QUICKXPLAIN [7] supports the determination of *preferred conflicts*. Although our discussions focus on constraint-based representations, the approach can be applied to any kind of satisfiability problem such as propositional satisfiability (SAT) and description logics (DL). We assume *monotonic satisfiability* (see Proposition 1).

Proposition 1. *If a solution for a CSP satisfies all constraints $c_i \in C$ then it also satisfies every proper subset $C' \subset C$.*

QUICKXPLAIN determines one *minimal preferred conflict* at a time which includes constraints one might be willing to relax. In the line with [7], an explanation X is preferred over an explanation Y under the following condition (see Definition 3).

Definition 3. *We define a total order $<$ on the constraints in $C = \{c_1..c_m\}$ which is represented as $[c_1 < c_2 < .. < c_m]$. If $c_i < c_j$, i.e., the importance of c_i is higher than c_j, then $i < j$. If X and Y are two lexicographical constraint orderings of $c_1..c_n$, Y is preferred over X ($Y >_{lex} X$)* iff $\exists k : c_k \in X - Y$ *and* $X \cap \{c_{k-1}..c_1\} = Y \cap \{c_{k-1}..c_1\}$.

Example 3. Given the constraint ordering $[c_3 < c_4 < c_5 < c_6]$ and two binary conflict sets $X = \{c_3, c_4\}$ and $Y = \{c_4, c_5\}$, Y is preferred over X since $c_3 \in X - Y$ with $X \cap \{c_2, c_1\} = Y \cap \{c_2, c_1\}$.

3 Parallelizing QUICKXPLAIN

Our approach to parallelize the consistency checks in QX substitutes the *direct* solver call INCONSISTENT(B) in QX with the activation of a lookahead function (QXGEN) in which consistency checks are not only triggered to directly provide feedback to QX requests, but also to be able to provide fast answers for consistency checks potentially relevant in upcoming states of a QX instance. In the parallelized variant of QUICKXPLAIN, consistency checking is activated by QX with INCONSISTENT($C, B, B\delta$) (see Algorithm 3). This also activates the QXGEN function (see Algorithm 4) that starts to generate and trigger (in a parallelized fashion) further consistency checks that might be of relevance in upcoming QX phases. For the description of QXGEN, we employ a two-level *ordered set* notation which requires, for example, to embed the QX B into $\{B\}$, etc. In QXGEN, C, $B\delta$, and B are interpreted as *ordered sets*.

Algorithm 3 INCONSISTENT($C, B, B\delta$):$Boolean$

1: **if** ¬EXISTSCONSISTENCYCHECK(B) **then**
2: QXGEN($\{C\}, \{B\delta\}, \{B - B\delta\}, \{B\delta\}, 0$)
3: **end if**
4: $return$(¬LOOKUP(B))

QXGEN-generated consistency checking tasks are stored in a LOOKUP table (see, e.g., Table 1). Thus, in the parallelized variant, QX has to activate the consistency check with INCONSISTENT($C, B, B\delta$). In contrast to the original QUICKXPLAIN approach, C and $B\delta$ are needed as additional parameters to conduct inferences about needed future consistency checks. While in the standard QX version $B\delta \subseteq B$, we assume $B\delta$ and B to be separate units in QXGEN.

node-id	constraint set	inconsistent
1	$\{c_3, c_4, c_5, c_6, c_7\}$	$true$
1.1	$\{c_2, c_3, c_4, c_5, c_6, c_7\}$	$false$
1.1.1	$\{c_1, c_6, c_7\}$	$true$
1.2.1	$\{c_4, c_5, c_6, c_7\}$	-

Table 1: LOOKUP table indicating the consistency of individual constraint sets. The consistency checking tasks have been generated by ADDCC in the QXGEN function (see Figure 2) and are executed in parallel. The '-' entry for the constraint set $\{c_4, c_5, c_6, c_7\}$ indicates that the corresponding consistency check is still ongoing or has not been started up to now. Algorithm 3 uses the LOOKUP function to test the consistency of a constraint set.

The QXGEN function (see Algorithm 4) predicts future potentially relevant consistency checks needed by QX and activates individual consistency checking tasks in an asynchronous fashion using the ADDCC (*add consistency check*) function. The ADDCC function triggers an asynchronous service that is in charge of adding consistency checks (parameter of ADDCC) to a LOOKUP table and issuing the corresponding solver calls. The global parameter $lmax$ is used define the maximum search depth of one activation of QXGEN.

In QXGEN, $|f(X)|$ denotes the number of constraints c_i in X (it is introduced due to the subset structure in C). Furthermore, SPLIT(C, C_a, C_b) splits C at position $\lfloor \frac{|C|}{2} \rfloor$ if $|C| > 1$ or C_1 (the first element of C) at position $\lfloor \frac{|C_1|}{2} \rfloor$ if $|C| = 1$ and $|C_1| > 1$ into C_a and C_b. Otherwise, no split is needed (C_1 is a singleton).

The first inner condition of QXGEN ($|f(\delta)| > 0$) generates a consistency check if this is needed. A consistency check is needed, if $B\delta$ gets extended from C or a singleton C has been identified which then extends B. The function ADDCC is used to add consistency check tasks which can then be executed asynchronously in a parallelized fashion. Thus, a consistency check in the LOOKUP table can be easily identified by an ordered constraint set that has also been used as parameter of ADDCC, for example, ADDCC($\{\{c_4, c_5\}, \{c_6, c_7\}\}$) results in the LOOKUP table enty $\{c_4, c_5, c_6, c_7\}$ which is internally represented with 4567.

Algorithm 4 QXGEN($C, B\delta, B, \delta, l$)
$C = \{C_1..C_r\}$ ... *consideration set* (*subsets* C_α)
$\mathrm{B}\delta = \{B\delta_1..B\delta_n\}$ ... *added knowledge* (*subsets* $\mathrm{B}\delta_\beta$)
$\mathrm{B} = \{\mathrm{B}_1..B_o\}$ ... *background* (*subsets* B_γ)
$\delta = \{D_1..D_p\}$ ... *to be checked* (*subsets* D_π)
l ... *current lookahead depth*

```
 1: if l < lmax then
 2:    if |f(δ)| > 0 then
 3:       AddCC(Bδ ∪ B)
 4:    end if
 5:    {Bδ ∪ B assumed consistent}
 6:    if |f(C)| = 1 ∧ |f(Bδ)| > 0 then
 7:       QXGen(Bδ, ∅, B ∪ {C1}, {C1}, l + 1)
 8:    else if |f(C)| > 1 then
 9:       Split(C, Ca, Cb)
10:       QXGen(Ca, Cb ∪ Bδ, B, Cb, l + 1)
11:    end if
12:    {Bδ ∪ B assumed inconsistent}
13:    if |f(Bδ)| > 0 ∧ |f(δ)| > 0 then
14:       QXGen({Bδ1}, Bδ − {Bδ1}, B, ∅, l + 1)
15:    end if
16: end if
```

If $B\delta \cup B$ is assumed to be *consistent*, additional elements from C have to be included such that an inconsistent state can be generated (which is needed for identifying a minimal conflict). This extension of $B\delta$ can be achieved by dividing C (if $|f(C)| > 1$, i.e., more than one constraint is contained in C) into two separate sets C_a and C_b and to add C_b to $B\delta$. If $|f(C)| = 1$, this singleton can be added to B which is responsible of collecting constraints that have been identified as being part of the minimal conflict. This is the case due to the already mentioned invariant property, i.e., $C \cup B\delta \cup B$ is inconsistent. If C contains only one constraint, i.e., C_1 is a singleton, it is part of the conflict set. If $B\delta \cup B$ is assumed to be *inconsistent*, it can be reduced and at least one conflict element will be identified in the previously added $B\delta_1$. If δ does not contain an element, no further recursive calls are needed since $B\delta - B\delta_1$ has already been checked.

The QXGEN function (Algorithm 4) is based on the idea of issuing recursive calls and adapting the parameters of the calls depending on the two possible situations 1) *consistent*($B\delta \cup B$) and 2) *inconsistent*($B\delta \cup B$). In Table 2, the different parameter settings are shown in terms of FOLLOW sets representing the settings of C, $B\delta$, B, and δ in the next activation of QXGEN.

Optimizations. To further improve the performance of QX, we have included mechanisms that help to identify irrelevant executions of consistency checks (see Table 3).

Cond.	Follow Sets			
	C	$B\delta$	B	δ
$\|f(C)\| = 1$	$B\delta$	$\emptyset$	$B \cup \{C_1\}$	$\{C_1\}$
$\|f(C)\| > 1$	C_a	$C_b \cup B\delta$	B	C_b
$\|f(B\delta)\| > 0$	$\{B\delta_1\}$	$B\delta - \{B\delta_1\}$	B	$\emptyset$

Table 2: FOLLOW sets of the QXGEN function. Depending on the assumption about the consistency of $B\delta \cup B$, the follow-up activations of QXGEN have to be parameterized differently.

If a consistency check has been completed, we are able to immediately decide whether some of the still ongoing or even not started ones can be canceled since they are not relevant anymore. For example, if the result of a consistency check is $\{c_3..c_7\}$ is $true$ (see Figure 2), the check $\{c_4..c_7\}$ does not have to be executed anymore (see also Proposition 1) or has to be canceled since QUICKXPLAIN will not need this check (see Figure 2). The deletion criteria for ongoing or even not started consistency checks can be pregenerated for $lmax$. An example for $lmax = 3$ is provided in Table 3.

Node-Id	Result of Consistency Check	
	true	*false*
1	*del*(1.2.x)	*del*(1.1.x)
1.1	*del*(1.1.2.x)	*del*(1.1.1.x)
1.2	*del*(1.2.2.x)	*del*(1.2.1.x)

Table 3: Optimizing the execution of consistency checks by detecting irrelevant ones. If the result of a specific check is known, LOOKUP (Algorithm 3) triggers the corresponding delete (del) operation. Nodes without associated consistency check are ignored.

In Table 3, the used *node-ids* are related to QXGEN instances, for example, in Figure 2 the consistency check $\{c_3..c_7 = true\}$ has the *node-id* 1.

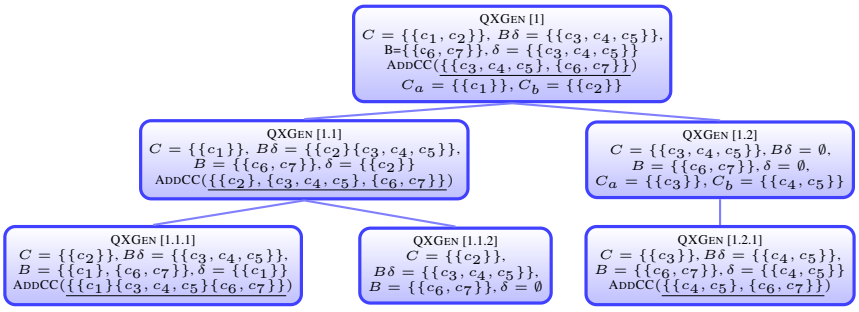

Fig. 2: QXGEN execution trace for $C = \{\{c_1..c_5\}\}$, $B\delta = \emptyset$, $B = \{\{c_6, c_7\}\}$, $\delta = \emptyset$, and $lmax = 3$. The consistency checks $\{c_3, c_4, c_5, c_6, c_7\}$ and $\{c_2, c_3, c_4, c_5 c_6, c_7\}$ (flattened list generated by ADDCC) can be used by the QUICKXPLAIN instance of Figure 1. The QXGEN nodes {[1],[1.1],[1.1.2]} represent the first part of the QUICKXPLAIN search path in Figure 1.

4 Analysis

QX *complexity*. Assuming a splitting $k = \lfloor \frac{m}{2} \rfloor$ of $C = \{c_1..c_m\}$, the worst case time complexity of QUICKXPLAIN in terms of the number of consistency checks needed for calculating one minimal conflict is $2k \times log_2(\frac{m}{k}) + 2k$ where k is the minimal conflict

set size and m represents the underlying number of constraints [7]. Since consistency checks are the most time-consuming part of conflict detection, the runtime performance of the underlying algorithms must be optimized as much as possible.

QXGEN *complexity*. The number (nc) of different possible minimal conflict sets that could be identified with QXGEN for an inconsistent constraint set C consisting of m constraints is represented by Formula 1.

$$nc(C) = \binom{m}{1} + \binom{m}{2} + ... + \binom{m}{m} \tag{1}$$

The upper bound of the space complexity in terms of recursive QXGEN calls for $lmax = n$ is in the worst case $2^{n-1} - 1$. Due to the combinatorial explosion, only those solutions make sense that scale $lmax$ depending on the available computing cores (see the reported evaluation results).

If the non-parallelized version of QUICKXPLAIN is applied, only sequential consistency checks can be performed. The approach presented in this paper is more flexible since $\#processors$ consistency checks can be performed in parallel. Assuming a maximum QXGEN search depth of $lmax = 4$, the maximum number of generated consistency checks is $\frac{2^{lmax-1}}{2}$ due to the binary structure of the search tree, i.e., 4 in our example. Out of these 4 checks, a maximum of 3 will be relevant to QX, the remaining ones are irrelevant for identifying the conflict in the current QX session. Thus, the upper bound of relevant consistency checks generated by QXGEN is $lmax$, i.e., one per QXGEN search level (see the outer left search path in Figure 2), the minimum number of relevant consistency checks is $\lceil \frac{lmax}{2} \rceil$, i.e., 2 in our example. Assuming $\#processors = 16$, our approach can theoretically achieve a performance boost of factor 3 since max. 3 relevant consistency checks can be performed in parallel. It is important to mention, that within the scope of these upper and lower bounds, a performance improvement due to the integration of QXGEN can be guaranteed independent of the underlying knowledge base.

Termination of QXGEN. If the parameter $lmax = n$, recursive calls of $QXGen$ stop at level $n - 1$. In every recursive step, based on the inconsistency invariant between $C \cup B\delta \cup B$, *either* 1) C is reduced to C_a and $B\delta$ gets extended with C_b (if $B\delta \cup B$ is consistent) or to $B \cup C$ if C is a singleton, *or* 2) $B\delta$ is further reduced (if $B\delta \cup B$ is inconsistent). Obviously, in the second case, the constraints in C are not relevant anymore, since a conflict can already be found in $B\delta$ (which is inconsistent with B).

QX-*conformance of* QXGEN. QXGEN correctly predicts QX consistency checks. It follows exactly the criteria of QX. If $|f(C)| = 1$, i.e., C includes only one constraint c_α, $B\delta \cup B$ is consistent and - as a consequence of the inconsistency invariant - c_α is a conflict element and therefore has to be added to B. The issued consistency check is $B \cup C$ - if inconsistent, a conflict has already been identified. If $|f(C)| > 1$, $C_b \cup B\delta \cup B$ has to be checked, since $B\delta \cup B$ is consistent and by adding C_b we follow the goal of restoring the inconsistency of $B\delta \cup B$. Finally, if $B\delta \cup B$ is inconsistent, no check has to be issued since $B\delta - \{B\delta_1\} \cup B$ has already been checked in the previous $QXGen$ call and obviously was considered consistent. Thus, QXGEN takes into account all QX states that can occur in the next step and exactly one of the generated consistency checks (if needed) will be relevant for QX. Finally, the generated consistency checks are irredundant since each check is only generated if new constraints from C are added to $B\delta$ or a new constraint (singleton C) is added to B.

Runtime Analysis. The following evaluations have been conducted on the basis of a *Java 8* based implementation of the parallelized QUICKXPLAIN (QX) version presented in this paper. For the implementation, we applied the *ForkJoin* framework for running parallel tasks in *Java*. This, while being less automatic than newer java implementations allowed us to fully control when threads are created and destroyed. For representing our test knowledge bases and conducting the corresponding consistency checks, we have used the SAT solving enviroment *SAT4j* (see www.sat4j.org). All experiments reported in the following have been conducted using an *Intel Xeon multi-core (16 cores) E5-2650 of 2.60GHz computer and 64 GB of RAM* that supports hyperthreading simulation of 64 cores which would allow us to run up to lmax=6. We have selected configuration knowledge bases (feature models) from the publicly available BETTY toolsuite [12], which allows for a systematic testing of different consistency checking and conflict detection approaches for knowledge bases. The knowledge base instances (represented as background knowledge B in QUICKXPLAIN) that have been selected for the purpose of our evaluation, range from 500 to 5.000 binary variables and also vary in terms of the number of included constraints (B 25-1.500 binary constraints). On the basis of these knowledge bases, we randomly generated requirements ($c_i \in C$) that covered $30\% - 50\%$ of the variables included in the knowledge base. These requirements have been generated in such a way that conflict sets of different cardinalities could be analyzed (see Table 4). In order to avoid measuring biases due to side effects in thread execution, we repeated each evaluation setting $3\times$.

The results of our QXGEN performance analysis are summarized in Table 4. On an average, the runtime needed by standard QUICKXPLAIN ($lmax = 1$) to identify a preferred minimal conflict of cardinality 1 is $3.2\times$ higher compared to a parallelized solution based on $QXGen$ ($lmax = 5$). In Table 4, each entry represents the average runtime in $msec$ for all knowledge bases with a preferred conflict set of cardinality n, where the same set of knowledge bases has been evaluated for $lmax$ sizes 1–6 ($lmax = 1$ corresponds to the usage of standard QX without QXGEN integration). In this context, we have introduced an additional baseline version 6(rd), where only a randomly selected set of QXGEN consistency checks has been evaluated. It can be observed that with an increasing $lmax$ the performance of QX increases. Starting with $lmax = 6$, a performance deterioration can be observed which can be explained by the fact that the number of pre-generated consistency checks starts to exceed the number of physically available processors. In the line of our algorithm analysis, the number of relevant consistency checks that can be performed with $lmax = 5$ is between 5 and 3. Taking into account the overheads for managing the parallelized consistency checks, the shown results support our theoretical analysis of QXGEN.

5 Conclusions

We have introduced a parallelized variant of the QUICKXPLAIN algorithm that is used for the determination of minimal conflict sets. Example applications are the model-based diagnosis of hardware designs and the diagnosis of inconsistent user requirements in configuration and recommender applications. Current approaches to the detection of minimal conflicts do not take into account the capabilities of multi-core architectures. Our parallelized variant of QUICKXPLAIN provides efficient conflict detection especially when dealing with large and complex knowledge bases.

	Conflict Cardinality				
$lmax$	1	2	4	8	16
1	**103993.0**	**286135.4**	**11006.4**	**30995.3**	**177354.7**
2	69887.3	193354.2	9528.0	26258.7	154093.7
3	49823.5	130657.5	11094.6	28639.6	154155.0
4	50042.8	136150.8	7577.0	19753.1	120992.0
5	**32481.4**	**88963.7**	**7750.0**	**18975.6**	**98242.7**
6	34946.2	94783.0	6367.6	17743.7	95816.3
6(rd)	105678.3	195987.5	112546.8	28676.1	179876.2

Table 4: Avg. runtime (in $msec$) of parallelized QX when determining minimal conflicts.

References

1. Bakker, R., Dikker, F.: Diagnosing and Solving Over-determined Constraint Satisfaction Problems. In: 13th International Joint Conference on Artificial Intelligence (IJCAI'93). pp. 276–281. Chambéry, France (1993)
2. Bordeaux, L., Hamadi, Y., Samulowitz, H.: Experiments with Massively Parallel Constraint Solving. In: 21st International Joint Conference on Artifical Intelligence. pp. 443–448. Morgan Kaufmann Publishers, San Francisco, CA, USA (2009)
3. Felfernig, A., Burke, R.: Constraint-based Recommender Systems: Technologies and Research Issues. In: ACM International Conference on Electronic Commerce (ICEC'08). pp. 17–26. Innsbruck, Austria (2008)
4. Gent, I., Miguel, I., Nightingale, P., McCreesh, C., Prosser, P., Nooore, N., Unsworth, C.: A Review of Literature on Parallel Constraint Solving. Theory and Practice of Logic Programming **18**(5–6), 725–758 (2018)
5. Hamadi, Y., Sais, L.: Handbook of Parallel Constraint Reasoning. Springer (2018)
6. Jannach, D., Schmitz, T., Shchekotykhin, K.: Parallelized Hitting Set Computation for Model-Based Diagnosis. In: 29^{th} AAAI Conference on Artificial Intelligence. pp. 1503–1510. AAAI Press, Austin, Texas (2015)
7. Junker, U.: QuickXPlain: Preferred Explanations and Relaxations for Over-constrained Problems. In: 19th national conference on Artifical intelligence. pp. 167–172. AAAI Press, San Jose, CA (2004)
8. Marques-Silva, J., Heras, F., Janota, M., Previti, A., Belov, A.: On Computing Minimal Correction Subsets. In: 23rd international Joint Conference on Artificial Intelligence. pp. 615–622. Beijing, China (2013)
9. Reiter, R.: A Theory of Diagnosis from First Principles. Artificial Intelligence **23**(1), 57–95 (1987)
10. Ricci, F., Rokach, L., Shapira, B., Kantor, P.: Recommender Systems Handbook. Springer (2011)
11. Rossi, F., Venable, K., Walsh, T.: A Short Introduction to Preferences: Between Artificial Intelligence and Social Choice. Morgan & Claypool Publishers (2011)
12. Segura, S., Galindo, J., Benavides, D., Parejo, J., Ruiz-Cortés, A.: BeTTy: Benchmarking and Testing on the Automated Analysis of Feature Models. In: 6th International Workshop on Variability Modeling of Software-Intensive Systems. pp. 63–71. VaMoS'12, ACM, New York, NY, USA (2012)
13. Stumptner, M.: An Overview of Knowledge-based Configuration. Ai Communications **10**(2), 111–125 (1997)
14. Tsang, E.: Foundations of Constraint Satisfaction. Academic Press (1993)