Proceedings of the 23rd International Conference on Principles of Knowledge Representation and Reasoning
KR in the Wild

A Distributed Framework for Compiling and Reasoning with d-DNNF

Zhenghang Xu[1,2] , **Minghao Yin**[1,2] , **Jianan Wang**[1] , **Jean-Marie Lagniez**[3]
[1]School of Information Science and Technology, Northeast Normal University, Changchun, China
[2]Key Laboratory of Applied Statistics of MOE, Northeast Normal University, Changchun, China
[3]Univ. Artois, CNRS, CRIL, F-62300 Lens, France
{xuzh121, ymh, wangjn}@nenu.edu.cn, lagniez@cril.fr

Abstract

Knowledge Compilation (KC) is a powerful paradigm that enables efficient reasoning by transforming propositional formulas into tractable target languages, such as Deterministic, Decomposable Negation Normal Form (d-DNNF). However, as real-world problem instances grow in complexity, the offline compilation phase becomes a significant computational bottleneck, often exceeding the memory and temporal limits of single-node systems. While distributed computing has been successfully applied to model counting (#SAT), extending these techniques to knowledge compilation remains a challenge due to the difficulty of sharing partial circuit fragments across distributed nodes.

In this paper, we propose `dkc`, the first distributed knowledge compiler designed for large-scale Decision-DNNF generation. Leveraging a Cube-and-Conquer strategy, `dkc` effectively partitions the search space into independent subproblems, mitigating the communication overhead typically associated with work-stealing architectures in circuit-based tasks. Recognizing that the utility of compilation lies in subsequent querying, we further introduce `dreasoner`, a distributed reasoning engine. `dreasoner` is capable of performing core inference tasks (including model counting, direct access, and uniform sampling) across a distributed d-DNNF structure, even under variable conditioning. Our experimental evaluation on benchmarks demonstrates that our distributed architecture scales effectively, enabling the compilation and querying of complex formulas that remain beyond the reach of state-of-the-art sequential compilers.

1 Introduction

Propositional logic serves as a cornerstone of Artificial Intelligence, providing a rigorous framework for knowledge representation and reasoning (Brachman and Levesque 2004). It underpins critical subfields, including Explainable AI (XAI) (Darwiche 2023), automated reasoning (Biere et al. 2021), and decision support systems (Plaisted 2015). However, the expressiveness of propositional logic comes at a steep computational cost: core inference tasks, such as satisfiability (SAT), are NP-complete (Cook 1971), while counting tasks like #SAT reside in the complexity class #P. This inherent complexity creates a bottleneck for real-time applications, particularly in high-stakes environments where low-latency reasoning is non-negotiable.

Industrial applications exacerbate this challenge, demanding scalability for massive problem instances. For example, AWS's *Zelkova* engine (Rungta 2022) processes billions of verification queries daily, highlighting the urgent need for optimized reasoning pipelines. Beyond simple satisfiability, practical tasks such as *model counting*, *direct access*, *uniform sampling*, and *model enumeration* are essential for extracting meaningful insights from complex constraints.

To mitigate online computational costs, *Knowledge Compilation* (KC) has emerged as a standard paradigm (Cadoli and Donini 1997; Darwiche and Marquis 2002). KC shifts the computational burden to an offline phase, transforming propositional formulas into target languages that support polynomial-time queries. Prominent among these is the *Deterministic, Decomposable Negation Normal Form* (d-DNNF) language, which renders tasks like Weighted Model Counting (WMC) tractable. The practical utility of this approach is evident in systems like Renault's configuration tool (Astesana, Cosserat, and Fargier 2010), where precomputed compilations enable rapid, interactive product customization.

Despite its benefits, compiling large-scale formulas into d-DNNF remains a significant hurdle; the process is both time and memory intensive, often exceeding the resources of a single machine for real-world instances. To handle this growing complexity, parallel and distributed computing offers a promising path forward. Given the structural proximity between #SAT solving and d-DNNF compilation, recent advances in distributed model counting, such as `dCountAntom` (Burchard, Schubert, and Becker 2016) and the more recent `DisCount` (Xu, Yin, and Lagniez 2025), provide a theoretical foundation for distributed KC. These solvers typically rely on a *Cube-and-Conquer* strategy, decomposing the search space into independent subproblems (cubes) distributed across a cluster.

While work-stealing approaches, such as `DMC` (Lagniez, Marquis, and Szczepanski 2018), have proven effective for counting, extending them to compilation presents a unique challenge. In #SAT, workers only need to exchange scalar counts. In KC, however, workers must share and merge *partial circuits*. Managing these circuit fragments across a distributed network introduces prohibitive communication overhead. Consequently, the Cube-and-Conquer paradigm is better suited for compilation, as it allows for a more struc-

tured reconstruction of the global target circuit.

Furthermore, unlike model counting where the process terminates once a value is returned, the output of compilation is a persistent artifact intended for subsequent querying. In this paper, we bridge the gap between distributed generation and distributed utilization. We introduce `dkc`, the first distributed knowledge compiler based on a Cube-and-Conquer architecture, and `dreasoner`, a distributed reasoner designed to evaluate compiled d-DNNF fragments across a cluster. Our reasoner supports distributed model counting, direct access, and uniform sampling, including queries under partial assignments (conditioning).

The remainder of this paper is organized as follows: Section 2 provides the necessary formal background on the KC map and d-DNNF; Section 3 details the architectural design of the distributed compiler `dkc`; Section 4 describes the `dreasoner` query engine; Section 5 presents an empirical evaluation on industrial benchmarks; and Section 6 concludes with a summary and future perspectives.

2 Preliminaries

2.1 Propositional Logic and Notations

We assume a standard propositional language $\mathcal{L}_\mathcal{P}$ built from a finite set of atoms $\mathcal{P}$ and the usual logical connectives $(\wedge, \vee, \rightarrow, \leftrightarrow, \neg)$. This language $\mathcal{L}_\mathcal{P}$ follows classical semantics. For any formula $\Sigma \in \mathcal{L}_\mathcal{P}$, the set of propositional variables occurring in it is denoted by $Var(\Sigma)$. Given a finite variable set X, we denote by $\{0,1\}^X$ the set of all possible Boolean assignments over X. Each formula $\Sigma \in \mathcal{L}_\mathcal{P}$ induces a Boolean function over its variables, mapping each assignment in $\{0,1\}^{Var(\Sigma)}$ to a truth value in $\{0,1\}$.

Assignments that satisfy Σ (mapping it to 1) are called *models* or *satisfying assignments*, and the set of all such models is denoted by $\mathcal{M}(\Sigma)$. Formulas Σ_1 and Σ_2 are logically equivalent, denoted $\Sigma_1 \equiv \Sigma_2$, if and only if $\mathcal{M}(\Sigma_1) = \mathcal{M}(\Sigma_2)$. Logical entailment is defined as $\Sigma_1 \models \Sigma_2$ if $\mathcal{M}(\Sigma_1) \subseteq \mathcal{M}(\Sigma_2)$. The symbols $\bot$ and $\top$ represent the unsatisfiable and the tautological formulas, respectively.

A *literal* is a variable x or its negation $\neg x$. For any literal ℓ, its underlying variable is denoted $Var(\ell)$ (where $Var(x) = Var(\neg x) = x$), and its complement is denoted $\sim\ell$ (with $\sim x = \neg x$ and $\sim\neg x = x$). The *conditioning* of a formula Σ by a literal ℓ (where $\ell = x$ or $\ell = \neg x$), denoted $\Sigma[\ell]$, is the formula resulting from replacing x (resp. $\neg x$) with $\top$ and its complement with $\bot$. This is followed by a simplification process based on standard Boolean identities (e.g., $\top \vee \Gamma = \top$, $\top \wedge \Gamma = \Gamma$) until a fixed point is reached. This operation naturally extends to a set of literals $S = \{\ell_1, \ldots, \ell_m\}$ such that $\Sigma[S] = ((\Sigma[\ell_1]) \ldots)[\ell_m]$.

An assignment μ is a set of literals interpreted conjunctively. We distinguish between *total assignments*, which assign every variable in $\mathcal{P}$, and *partial assignments*. A formula in *Conjunctive Normal Form* (CNF) is a conjunction of clauses, where each clause is a disjunction of literals; for convenience, we view a CNF as a set of clauses and a clause as a set of literals.

Example 1. *Let* $\Psi = \{a \vee b, \neg a \vee \neg b, c \vee d \vee a, c \vee d \vee b\}$ *be a CNF formula with* $Var(\Psi) = \{a, b, c, d\}$.

2.2 d-DNNF Representation

The Deterministic Decomposable Negation Normal Form, or d-DNNF, is a specific type of Boolean circuit characterized by a single output node acting as its root. Structurally, a d-DNNF is represented as a rooted Directed Acyclic Graph (DAG), denoted $\langle V, E \rangle$. In this graph, leaf nodes consist of literals or Boolean constants ($\bot, \top$), while internal nodes are either decomposable $\wedge$ gates or deterministic $\vee$ gates. A gate $N = \wedge(N_1, \ldots, N_k)$ is *decomposable* if the variables appearing in the sub-circuits rooted at N_i and N_j are disjoint for all $i \neq j$. An OR gate $N = \vee(N_1, \ldots, N_k)$ is *deterministic* if its children N_i and N_j are pairwise inconsistent for any $i \neq j$. The size of a d-DNNF $\Sigma = \langle V, E \rangle$, denoted $|\Sigma|$, corresponds to the number of its edges $|E|$. As a language, d-DNNF is universal, meaning it is capable of representing any propositional theory (Darwiche 2002).

State-of-the-art d-DNNF compilers, including `C2D` (Darwiche 2004), `Dsharp` (Muise et al. 2012), `d4` (Lagniez and Marquis 2017a), and `SharpSAT-TD` (Kiesel and Eiter 2023), typically generate *decision-DNNF* representations. A decision-DNNF is a restricted form of d-DNNF where deterministic $\vee$ gates are replaced by decision gates of the form $N = ite(x, N_1, N_2)$. Here, ite represents the "`if-then-else`" connective, and the decision variable x must not appear in the sub-circuits N_1 or N_2. Also known as decomposable decision graphs (Fargier and Marquis 2006), decision-DNNFs can be transformed into d-DNNFs in linear time. By expanding a decision node $N = ite(x, N_1, N_2)$ into the fragment $(\neg x \wedge N_1) \vee (x \wedge N_2)$, the properties of decomposability (since $x \notin Var(N_i)$) and determinism (due to the mutual exclusivity of x and $\neg x$) are preserved.

Example 2 (Example 1 cont'ed)**.** *For the CNF formula* Ψ *from Example 1, an equivalent d-DNNF representation is* $\Sigma = ((\neg a \wedge b) \vee (a \wedge \neg b)) \wedge (c \vee (\neg c \wedge d))$.

The d-DNNF language is a highly effective representation for automated reasoning, as it supports a wide range of queries and transformations – such as satisfiability checking and conditioning – in polynomial time. Importantly, tasks like direct access (Carmeli et al. 2023; Bringmann, Carmeli, and Mengel 2025), uniform sampling (Sharma et al. 2018), and model counting (Darwiche 2002) can be performed efficiently when the input formula is compiled into a d-DNNF.

2.3 Queries

Beyond the theoretical elegance of d-DNNF, its primary value lies in its support for efficient reasoning. In real-world scenarios (ranging from interactive product configuration (Kübler, Zengler, and Küchlin 2010; Sundermann et al. 2024; Oh, Gazzillo, and Batory 2019; Plazar et al. 2019) to probabilistic reasoning (Chavira and Darwiche 2008; Bart et al. 2016) and system verification (Baluta et al. 2021; Heradio et al. 2016)) practitioners require more than just a compact representation; they need to extract specific insights efficiently. Core tasks such as checking consistency, estimating probabilities, or generating diverse scenarios rely on four fundamental queries: *Satisfiability*, *Model Counting*, *Direct Access*, and *Uniform Sampling*. While these tasks are generally intractable (NP-complete or #P-complete) on

raw CNF formulas, they become polynomial-time operations when the formula is compiled into a d-DNNF.

Satisfiability. The *Satisfiability* (SAT) check is the most fundamental query: it determines whether there exists at least one assignment μ such that $\mu \models \Psi$ (i.e., if $\|\Psi\| > 0$). While SAT is the canonical NP-complete problem, it is trivial on a d-DNNF. Due to the properties of decomposability and determinism, a d-DNNF circuit is satisfiable if and only if its root is not equivalent to the constant $\bot$. This check can be performed in constant time if the compiler prunes inconsistent branches during construction, or in linear time otherwise.

Model Counting. The *Model Counting* problem (a.k.a. #SAT) involves determining the total number of satisfying assignments for a given propositional formula Φ, denoted by $\|\Phi\|$. This task is the functional counterpart to the decision problem and is known to be #P-complete. While computationally intensive for general formulas, model counting becomes a linear-time operation on d-DNNF representations. This is achieved by traversing the circuit bottom-up: AND gates return the product of their children's counts (due to decomposability), and OR gates return the sum of their children's counts (due to determinism).

Example 3 (Ex. 1 cont.)**.** *For the formula Ψ from Example 1, $\|\Psi\| = 6$ and the models are: $\{a, \neg b, c, d\}$, $\{\neg a, b, c, d\}$, $\{a, \neg b, c, \neg d\}$, $\{\neg a, b, c, \neg d\}$, $\{a, \neg b, \neg c, d\}$, and $\{\neg a, b, \neg c, d\}$.*

Direct Access. The *direct access* task, originally introduced in the database literature (Bagan et al. 2008), requires returning the k-th model of a formula Ψ according to a lexicographical order $\prec_{lex}$. If k exceeds the total model count $\|\Psi\|$, the query fails. In propositional logic, $\prec_{lex}$ is defined by fixing an ordering τ over the variables and treating assignments as binary words in $\{0,1\}^{Var(\Psi)}$. We denote this ordered relation as $\prec^{\tau}_{lex}$, using $x <_{\tau} y$ to indicate that variable x precedes y in τ.

Example 4 (Ex. 1 cont.)**.** *Using $\tau = (a, b, c, d)$ for the formula Ψ, the first model (word 0101) is $\{\neg a, b, \neg c, d\}$, and the third model (word 0111) is $\{\neg a, b, c, d\}$.*

Direct access serves as a fundamental primitive for advanced tasks like uniform sampling without replacement (Sharma et al. 2018). While generally #P-hard, the query is tractable for d-DNNF because the language supports polynomial-time counting. The algorithm iteratively selects the next variable x in τ. Starting with an empty assignment σ, it checks if the count of the branch $\neg x$ (i.e., $\|\Psi[\sigma \cup \{\neg x\}]\|$) is at least k. If so, $\neg x$ is added to σ; otherwise, x is added, and k is offset by the count of the $\neg x$ branch.

Example 5 (Ex. 4 cont.)**.** *To find the third model of Ψ, we start with $\sigma = \emptyset$. Since $\|\Psi[\neg a]\| = 3 \geq 3$, we set $\sigma = \{\neg a\}$. Next, for variable b, we find $\|\Psi[\{\neg a, \neg b\}]\| = 0$. Since $0 < 3$, we set $\sigma = \{\neg a, b\}$. For c, $\|\Psi[\{\neg a, b, \neg c\}]\| = 1 < 3$, so we select c, resulting in $\sigma = \{\neg a, b, c\}$. Finally, variable d is determined, yielding $\{\neg a, b, c, d\}$.*

Uniform Sampling. Uniform sampling involves a generator $\mathcal{G}$ that produces a sample from $\mathcal{M}(\Psi)$ such that every model has an equal probability $1/\|\Psi\|$ of being selected. A standard technique for achieving seed-based reproducible uniform sampling involves determining the total model count c, generating a uniform random integer $k \in \{1, \ldots, c\}$, and retrieving the corresponding model via a direct access query (Lagniez and Lonca 2025). In contrast, the approach proposed in (Sharma et al. 2018) tags the circuit to compute uniform samples more efficiently. However, such methods do not impose a strict order on models, meaning the output is inherently dependent on the specific circuit structure. Since compilers generally lack deterministic control over the resulting circuit topology, these structure-dependent methods make seed-based reproducibility difficult to achieve, even though they are typically faster to compute.

Inference under Conditioning. A critical requirement for real-world reasoning is the ability to answer these queries under specific contexts. Given a set of evidence literals E, all aforementioned queries – satisfiability, model counting, direct access, and uniform sampling – can be performed on the conditioned formula $\Psi[E]$. Because d-DNNF is closed under conditioning, $\Psi[E]$ remains a d-DNNF, ensuring that inference remains polynomial even when restricted to a subspace of the models.

3 Distributed d-DNNF Compiler

In this section, we present the architecture of `dkc`, our distributed framework for knowledge compilation. Our approach builds upon the *Cube-and-Conquer* paradigm effectively employed in distributed model counter `DisCount`, but introduces fundamental structural changes to support the generation and persistent storage of circuit fragments, as well as an online query processing phase.

3.1 The Cube-and-Conquer Paradigm

The foundation of our approach is the decomposition of the initial formula Ψ into a set of independent subproblems using a Lookahead-based partitioning strategy. Formally, given a propositional formula Ψ, the Cube-and-Conquer method identifies a set of *cubes* (partial assignments) $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_m\}$ such that:

$$\Psi \equiv \bigvee_{\tau \in \mathcal{T}} (\Psi \wedge \tau) \tag{1}$$

Crucially, these cubes are generated to be pairwise disjoint ($\tau_i \wedge \tau_j \models \bot$ for $i \neq j$) and to cover the entire solution space of Ψ. This disjointness is particularly advantageous for d-DNNF compilation, as it naturally satisfies the *determinism* property at the root level of the global circuit.

In standard distributed counting, a Master dispatches cubes to a worker set $\mathcal{W}$ of size $N = |\mathcal{W}|$. Each $W_j \in$

$\mathcal{W}$ simplifies a cube τ to $\Psi_{|\tau}$, returns its count, and discards the task. The Master then aggregates the global sum $\sum_{\tau \in \mathcal{T}} \|\Psi \wedge \tau\|$.

3.2 From Transient Counting to Persistent Compilation

Although the "count-and-forget" approach is sufficient for #SAT, Knowledge Compilation requires the preservation of the structural information computed by the workers. To achieve this, we modify the worker lifecycle and the Master-Worker protocol as follows.

Distributed Compilation. Instead of a model counter, each worker is equipped with a sequential d-DNNF compiler (`d4` in our experiments). Upon receiving a cube τ, the worker compiles the conditioned formula $\Psi \wedge \tau$ into a d-DNNF fragment Σ_τ. Unlike the counting approach, the worker does *not* return the explicit circuit Σ_τ to the Master at the end. Transmitting large circuits over the network would create a significant bottleneck and likely overwhelm the Master's memory. Instead, the worker appends Σ_τ to a persistent, **ordered list** L_j stored in local memory.

$$L_j = \langle \Sigma_1, \Sigma_2, \ldots, \Sigma_k \rangle \tag{2}$$

where Σ_i denotes the i-th d-DNNF fragment compiled by worker W_j. The worker then acknowledges the completion of the task to the Master, but the circuit itself remains distributed. Maintaining a strict order is crucial for the subsequent reasoning phase, as it allows the worker to target specific circuit fragments by their index when performing sampling or direct access queries.

The Global Virtual Circuit. From a logical perspective, the global d-DNNF Δ representing Ψ effectively exists as a virtual structure distributed across the cluster. It can be viewed as a large deterministic OR gate located at the Master, with edges connecting to the roots of the local d-DNNF fragments stored in the lists of the workers:

$$\Delta = \bigvee_{j=1}^{N} \left(\bigvee_{\Sigma \in L_j} \Sigma \right) \tag{3}$$

The correctness of this virtual reconstruction follows directly from the Cube-and-Conquer partition: the cubes cover the whole solution space, and their pairwise inconsistency ensures determinism at the top OR gate. By keeping the fragments Σ local to the workers, `dkc` overcomes the memory wall that limits centralized compilers, allowing for the representation of formulas significantly larger than the RAM capacity of a single machine. This also motivates the distributed reasoning phase: collecting all fragments on one node would often be impractical, as the resulting circuit can exceed the memory available on any single machine.

3.3 Distributed Query Processing

The transition from offline compilation to online reasoning marks the second fundamental shift in our architecture. Unlike the standard distributed counting workflow, which terminates immediately after computing a global sum, `dkc` initializes a persistent query engine, `dreasoner`, once the compilation phase concludes.

In this phase, the Master node evolves from a *Task Dispatcher* into a *Query Coordinator*. The workflow for handling a query $\mathcal{Q}$ (whether for satisfiability, model counting, direct access, or uniform sampling) proceeds as follows:

1. **Query Broadcast:** The Master receives a query request $\mathcal{Q}$, potentially accompanied by a set of conditioning literals γ (evidence). The Master broadcasts $\mathcal{Q}$ and γ to all participating workers.
2. **Local Evaluation:** Each worker W_j executes the query against its local knowledge base. Specifically, the worker iterates through its persistent, ordered list L_j of d-DNNF fragments. For each fragment $\Sigma \in L_j$, the worker computes the query result locally (e.g., counting models consistent with γ). The specific algorithmic procedures for these local evaluations are detailed in the next section.
3. **Global Aggregation:** Workers transmit their local results back to the Master. The Master then aggregates these partial responses to derive the final global answer. For instance, in a model counting query, this involves summing the scalar counts returned by each worker ($C = \sum c_j$); for direct access or sampling, it involves identifying the specific worker holding the target model.

This architecture guarantees that traversing the complex d-DNNF structures remains fully distributed across the cluster, while the Master is responsible only for lightweight coordination and scalar aggregation.

3.4 The Compilation Algorithm

The control logic of `dkc`, outlined in Algorithm 1, is directly derived from the `DisCount` framework introduced in (Xu, Yin, and Lagniez 2025). We explicitly retain the core infrastructure of that approach, including the preprocessing pipeline, the formula broadcast mechanism, and the lookahead-based cube generation strategy, which effectively partitions the search space. Consequently, the Master's control loop remains structurally similar to the original counting architecture. The fundamental difference lies in the nature of the distributed task: whereas `DisCount` workers reduce sub-problems to scalar counts, `dkc` workers now act as compilers, generating and persisting d-DNNF fragments for future reasoning.

Preprocessing and Partitioning (Lines 1–3). The process begins with a preprocessing step on the Master node (line 1). Standard simplification techniques (e.g., unit propagation, vivification, ...) are applied to the input formula Σ to reduce its size before transmission. The simplified formula is then broadcast to all worker nodes $\mathcal{W}$ (line 2). Subsequently, the Master generates a set of disjoint cubes $\mathcal{C}$ using the heuristic inherited from `DisCount` (line 3). The number of cubes is determined by a scaling factor ($nbCubes$) relative to the cluster size $|\mathcal{W}|$, ensuring enough

Algorithm 1: `dkc` Compilation Control Loop

Data:

- Σ: a CNF formula;
- $nbCubes$: granularity factor (integer);
- $\mathcal{W}$: the set of workers.

Result: A d-DNNF ready for querying.

```
   // 1. Preprocessing and Broadcast
1  Σ ← preprocessing(Σ);
2  broadcast(Σ, W);
   // 2. Search Space Partitioning
3  C ← generateCubes(Σ, nbCubes × |W|, W);
   // 3. Distributed Compilation Loop
4  idle[w] ← 1 for each w ∈ W;
5  setWorkerCompilationMode(W);
6  while C ≠ ∅ do
7      if ∃ w ∈ W s.t. idle[w] = 1 then
8          τ ← pop(C);
9          sendCubeToCompile(τ, w);
10         idle[w] ← 0;
       // Poll for task completion
11     while ∃ w ∈ W s.t. free(w) and idle[w] = 0 do
12         idle[w] ← 1;
   // 4. Barrier Synchronization
13 while ∃ w ∈ W s.t. idle[w] = 0 do
14     if free(w) then idle[w] ← 1;
   // 5. Transition to Reasoning
15 dreasoner(W);
   // 6. Termination
16 for w ∈ W do stop(w);
17 return
```

granularity to smooth out runtime variations among workers.

Dynamic Dispatch Loop (Lines 3–12). The Master utilizes a dynamic scheduling strategy to manage the workload:

1. **Assignment:** As long as there are cubes remaining in $\mathcal{C}$ and idle workers are available, the Master pops a cube τ and dispatches the task (`sendCubeToCompile`) to the target worker (lines 7–10).
2. **Completion Check:** The Master continuously polls for completion signals (`free(w)`). When a worker w finishes compiling its assigned fragment (and appends it to its local list L_w), it notifies the Master, which then marks w as available to receive a new cube (lines 11–12).

Finalization and Transition (Lines 13–16). Once the dispatch queue $\mathcal{C}$ is empty, the Master enters a synchronization barrier, strictly waiting for all currently active workers to complete their final tasks and return to an idle state (lines 13–14). Crucially, unlike counting algorithms that terminate immediately after this point, `dkc` preserves the distributed state and transitions into the *Online Phase* (line 15). The function `dreasoner`($\mathcal{W}$) initializes the Query Coordinator loop (described in Section 4), effectively blocking the main execution flow to accept and process queries against the distributed circuit. Only when the reasoning session is explicitly ended by an external signal does the control flow return to terminate the workers (line 16).

4 Distributed d-DNNF Reasoner

This section describes how the `dreasoner` engine handles query processing. Unlike traditional counting solvers that terminate immediately after reporting a scalar, `dkc` transitions into a persistent serving state upon the completion of Algorithm 1. In this state, the global d-DNNF Δ is implicitly represented by the union of disjoint fragments distributed across the worker nodes. Consequently, the Master node shifts its role from a *Task Dispatcher* to a *Query Coordinator*, executing an event loop to process incoming requests. We categorize the supported operations into two classes: *Aggregation Queries* (satisfiability and model counting), which summarize properties of the entire solution space, and *Member Queries* (uniform sampling and direct access), which retrieve specific model instances.

4.1 Aggregation Queries

Aggregation queries compute global properties over the distributed d-DNNF Δ. Exploiting the decomposability and determinism of the fragments, these operations naturally map to a parallel reduction pattern.

Satisfiability. To check satisfiability under a partial assignment γ, the Master broadcasts γ to all workers. Each worker W_j scans its local fragments L_j; if any $\Sigma \in L_j$ is consistent with γ, the worker reports `true`. Formally, the global satisfiability is the disjunction of local consistency checks:

$$\texttt{SAT}(\Delta \wedge \gamma) = \bigvee_{j=1}^{N} \left(\bigvee_{\Sigma \in L_j} \texttt{SAT}(\Sigma \wedge \gamma) \right) \tag{4}$$

Model Counting. Analogously, for model counting, the Master broadcasts the request (potentially with weights or conditioning γ). Each worker sums the model counts of its resident fragments, returning a scalar partial sum to the Master, which computes the global total:

$$\#\texttt{SAT}(\Delta \wedge \gamma) = \sum_{j=1}^{N} \sum_{\Sigma \in L_j} \#\texttt{SAT}(\Sigma \wedge \gamma) \tag{5}$$

Crucially, since the fragments are persisted in worker memory, computing $\#\texttt{SAT}(\Sigma \wedge \gamma)$ reduces to a graph traversal linear in the size of the local fragments. This avoids the latency of disk I/O or recompilation, ensuring high throughput even under arbitrary conditioning.

4.2 Member Queries

Member queries involve generating full assignments (models) from the solution space. These operations are *count-based*, relying on the model counting primitive to navigate the decision tree structure.

Direct Access. The Direct Access query retrieves the k-th model of the formula according to a fixed global variable ordering $\prec_{lex}$. Since the decision-DNNF is distributed, the Master must orchestrate a global search variable by variable. Let x be the current variable in $\prec_{lex}$, γ the current partial assignment, and k the target index. At each step, the Master tentatively branches on $\neg x$ by querying the global count c of the subspace consistent with $\gamma \wedge \neg x$. If $c \geq k$, the target model lies in the $\neg x$ branch; the Master updates $\gamma \leftarrow \gamma \cup \{\neg x\}$. Otherwise, the model resides in the x branch; the Master updates $\gamma \leftarrow \gamma \cup \{x\}$ and adjusts the target index $k \leftarrow k - c$. This procedure repeats for all variables until a complete model is constructed.

Uniform Sampling. Uniform sampling generates a model such that every solution has an equal probability $1/N$ of being selected. Our architecture implements a two-stage routing protocol that leverages the disjointness of the cubes to minimize network overhead. First, the Master issues a global *Model Counting* query to determine the total number of solutions C, and generates a set of random indices $\mathcal{S} = \{s_1, \ldots, s_m\}$ where each $s \in [1, C]$. Instead of iterating through variables as in Direct Access, the Master directly identifies the worker W_t that "owns" the s-th solution using the prefix sums of the worker counts $\{c_j\}$. Specifically, W_t is the worker satisfying $\sum_{j<t} c_j < s \leq \sum_{j \leq t} c_j$. The query is then unicast exclusively to W_t with the local index $s' = s - \sum_{j<t} c_j$, instructing the worker to traverse L_t and retrieve the s'-th model.

Crucially, when generating a batch of samples under the same evidence γ, the model counts for every node in the circuit are cached during the initial global counting phase. Consequently, subsequent sampling queries reuse these values to navigate the circuit branches, ensuring the expensive counting operation is amortized across the entire batch.

4.3 Reasoning Algorithm

The control logic of the `dreasoner`, outlined in Algorithm 2, operates as a continuous, event-driven service. Unlike the linear compilation phase, the reasoning engine executes an infinite loop to process a stream of incoming queries until an explicit termination signal is received.

The Master blocks on the reception of client requests (Lines 1–5). Upon receiving a query $\mathcal{Q}$, it validates the active state and invokes the `ProcessQuery` dispatcher, which coordinates the distributed workers $\mathcal{W}$ based on the query type and evidence γ. Finally, upon completion of the reasoning task, the computed result is formatted and emitted via the `Display` function.

For *satisfiability* (Lines 9–10) and *model counting* (Lines 11–12), the algorithm follows a map-reduce pattern. The Master broadcasts the request to all workers and aggregates their partial results (via disjunction or summation) to return the global answer.

For *direct access* (Lines 13–23), the algorithm constructs the model by determining the assignment of each variable v according to the global order $\prec_{lex}$. At each step, it broadcasts a query to count the models consistent with the negative branch ($\gamma \wedge \neg x$) (Line 18). Let c be this count; if $k \leq c$, the target model resides in the negative branch, so the algorithm commits to $\neg x$ (Line 19). Otherwise, it deduces the model implies x, updates the target rank $k \leftarrow k - c$, and

Algorithm 2: `dreasoner` Master Control Logic

Data:
- $\mathcal{W}$: the set of workers;
- Δ: global decision-DNNF.

```
   // Main Event Loop
1  while true do
2      Q ← WaitForQuery();
3      if Q = terminate then break;
4      result ← ProcessQuery(Q, W);
5      Display(result);
   // Query Processing Function
6  Function ProcessQuery(Q, W):
7      γ ← Q.evidence;
8      switch Q.type do
9          case Satisfiability do
10             return SAT(Δ ∧ γ);
11         case Model Counting do
12             return #SAT(Δ ∧ γ);
13         case Direct Access do
14             μ ← γ;
15             k ← Q.k;
               // Binary search
16             foreach x ∈ Vars(Ψ) ordered by ≺_lex do
17                 if x ∈ Vars(μ) then continue;
18                 c ← #SAT(Δ ∧ μ ∧ ¬x);
19                 if c ≥ k then μ ← μ ∪ {¬x} ;
20                 else
21                     μ ← μ ∪ {x};
22                     k ← k − c;
23             return μ;
24         case Uniform Sampling do
25             C, {c_j} ← #SAT(Δ ∧ γ);
26             S ← GenRandomIndices(C, Q.size);
27             Γ ← ∅;
28             foreach s ∈ S do
29                 w_t ← SelectWorker(s, {c_j});
30                 s' ← s − Σ_{j<t} c_j;
31                 Γ ∪ Sample(w_t, s', γ);
32             return Γ;
```

proceeds to the next variable (Lines 21–22).

For *uniform sampling* (Lines 24–32), the algorithm employs a routing-based strategy. After retrieving the solution distribution $\{c_j\}$ and the global total C (Line 25), the Master maps a batch of random global indices $\mathcal{S}$ to specific workers (Line 26). It then unicasts retrieval requests to the identified worker (Line 29), explicitly querying for the model at the local index s' corresponding to the target global position within that worker's fragment (Lines 30–31).

5 Experiments

5.1 Implementation and Environment

Our distributed architecture, `dkc`, is implemented in `C++` as an extension of the open-source `DisCount` framework,[1] adhering to the configuration guidelines established in (Xu, Yin, and Lagniez 2025). The software used in the experiments, along with the corresponding logs, is available at https://zenodo.org/records/19936190. The system integrates several state-of-the-art components to optimize performance. For Boolean constraint propagation and solving, we employ CaDiCaL (Biere et al. 2024) as the underlying SAT solver, maintaining a single incremental solver instance to minimize overhead across repeated calls. Structural decomposition is handled by `FlowCutter` (Hamann and Strasser 2018), specifically the implementation from the PACE 2017 challenge,[2] with a strict time budget of 10 seconds allocated for computing the tree decomposition.

Prior to compilation, the formula undergoes simplification via the `B+E` preprocessor (Lagniez, Lonca, and Marquis 2020).[3] We utilize the `equiv` strategy (combining vivification, backbone extraction, and occurrence elimination (Lagniez and Marquis 2017c)) limited to a 5-second budget. Finally, for the worker nodes, we integrate the `d4` compiler (Lagniez and Marquis 2017b).[4] Crucially, `d4` is operated in *incremental mode*, allowing it to preserve its cache between different compilation tasks (cubes), thereby avoiding redundant computations during the local compilation phase.

All experiments were conducted on a cluster comprising 32 nodes, each equipped with an Intel® Xeon® E5-2643 v4 CPU (3.30 GHz) and running Rocky Linux 9.5 (kernel 5.14). The nodes are interconnected via a 1 GiB/s Ethernet network. All runs were fully distributed, with one MPI process allocated per physical core. The software environment was compiled using GCC 11.5 and Open MPI 5.1.0a1.

Our evaluation is divided into two parts. First, we assess the performance of the compilation process using benchmarks from the most recent Model Counting Competition (MC 2025), as detailed in Section 5.2. Second, we evaluate the efficiency of the reasoner in Section 5.3. For this latter task, we focus on a subset of benchmarks widely used for querying assessment, specifically selecting instances that are tractable for compilation to isolate the reasoning performance.

[1] https://zenodo.org/records/16536062

[2] https://github.com/kit-algo/flow-cutter-pace17.git

[3] http://www.cril.univ-artois.fr/kc/bpe2.html

[4] https://github.com/crillab/d4v2

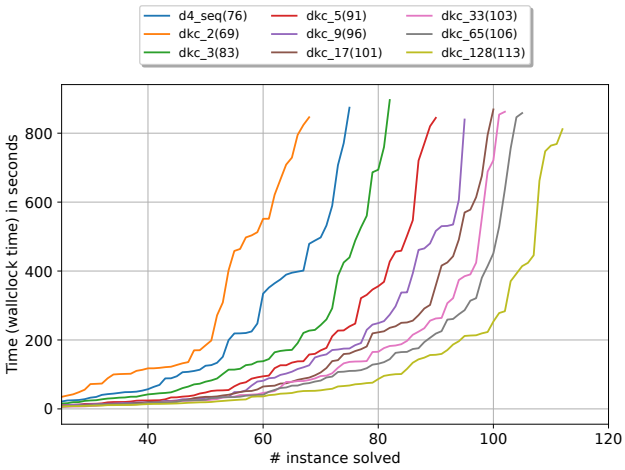

Figure 1: Performance comparison of the model counters `d4` (`d4_seq`) and various versions of `dkc` for different numbers of cores. The plot reports the number of instances solved as a function of time. For each solver, the total number of instances solved is indicated in parentheses in the legend.

5.2 Evaluating the Compilation Process

To evaluate the efficiency of `dkc`, we utilized benchmarks from the most recent Model Counting Competition (MC 2025).[5] A wall-clock time limit of 900 seconds and a memory limit of 32 GiB were enforced for each run. Figure 1 presents a cactus plot illustrating the number of instances solved (x-axis) within a given wall-clock time (y-axis) by `d4` and various `dkc` configurations. To assess scalability, we varied the total core count from 2 to 128. Since `dkc` designates a dedicated Master node, these configurations correspond to $1, 2, 4, \ldots, 127$ active workers.

We first observe that the sequential solver `d4`, with 76 solved instances, outperforms `dkc` when restricted to a single worker, which solves only 69. This performance gap indicates that the overhead of the Cube-and-Conquer strategy, specifically the management of cubes and network communication, is detrimental when no parallelism is available to offset the cost. However, this overhead is quickly amortized; with just two workers, `dkc` solves 83 instances, surpassing the sequential baseline. Increasing the worker count significantly boosts throughput, enabling the solution of progressively harder instances within the timeout. Specifically, the number of solved instances scales robustly: 91 with 4 workers, 96 with 8 workers, and continuing to climb steadily to 113 solved instances with 127 workers.

To assess the scalability of `dkc`, Figure 2 reports the distribution of speedups as the cluster size expands from 3 to 128 cores. The y-axis (log scale) represents the speedup factor relative to the baseline configuration of 2 cores (1 Master + 1 Worker). For a given instance solved in t_n seconds using n cores, the speedup is calculated as $\frac{\min(t_2, 900)}{t_n}$, where t_2 is the wall-clock time of the baseline run. Crucially, since t_2 is capped at the 900-second timeout, these values represent

[5] https://nextcloud.liu.se/s/MfLJWiDfYzGjXs8

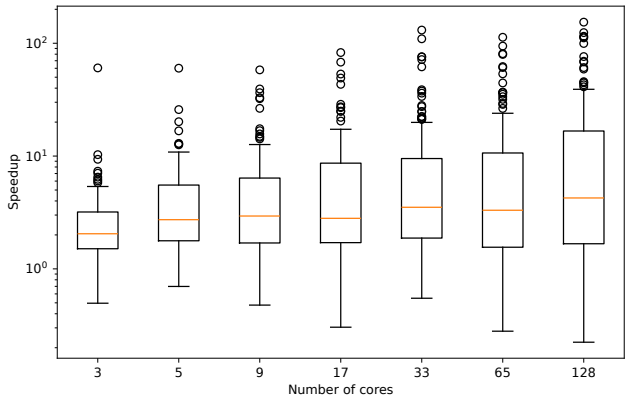

Figure 2: Distribution of speedups achieved by `dkc` across varying core counts, normalized against the minimal distributed configuration (1 Master + 1 Worker).

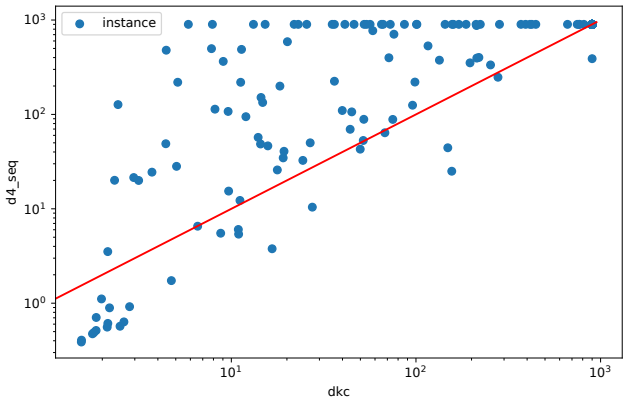

Figure 3: Scatter plot comparing the solving times of `dkc` with 128 cores (x-axis) versus the sequential baseline `d4` (y-axis). Points above the diagonal indicate instances where the distributed approach outperforms the sequential one.

a *conservative lower bound* on the actual speedup for any instance that times out in the baseline configuration.

The results demonstrate robust scalability. We observe a consistent upward trend in the median speedup (orange line) as the core count increases, confirming that `dkc` effectively translates additional worker nodes into reduced runtime. Notably, the distribution exhibits significant positive skew; the high-performing outliers indicate that for computationally intensive instances, the framework achieves near-linear speedups. This suggests that the Cube-and-Conquer strategy successfully amortizes the communication and synchronization overhead, particularly when the workload is sufficient to saturate the distributed workers.

To conclude the evaluation of the compilation process, Figure 3 presents a scatter plot comparing the runtime of the sequential baseline `d4` against the fully distributed `dkc` configuration (128 cores). Each data point represents a single benchmark instance, with its coordinates (x, y) corresponding to the solving time of `dkc` and `d4`, respectively. The results clearly demonstrate a threshold effect: for harder instances, specifically those requiring more than 100 seconds sequentially, the distributed approach consistently pays off, yielding significant speedups. However, the presence of outliers below the diagonal (on the right) reveals that for certain instances, the Cube-and-Conquer decomposition may be detrimental. In these cases, the overhead of partitioning the search space likely disrupts the variable ordering heuristics that allow the sequential solver to find a compact decision-DNNF, resulting in a net performance loss despite the parallelism.

5.3 Evaluating the Query Process

Our evaluation utilizes benchmarks from (Lagniez and Lonca 2025).[6] These instances are particularly suitable for evaluating `dreasoner` as they represent concrete applications where compilation into decision-DNNF is mandatory. Due to computational constraints, we did not consider the full suite of 1,425 benchmarks; instead, we randomly selected a representative subset of 200 instances. We impose a global wall-clock time limit of 1800 seconds and a memory limit of 32 GiB per run. To strictly evaluate each phase of our architecture, we separately measure the time required for distributed compilation (t_c) and the time required to answer the query batch (t_q). To prevent a single phase from monopolizing the resource budget, we enforce a local timeout of 900 seconds for each; a run is considered a timeout if either $t_c > 900$ s or $t_q > 900$ s.

To evaluate the `dreasoner` engine, we generated a synthetic workload designed to mimic real-world usage patterns. The workload construction begins by computing a single satisfying assignment (seed model) ω for each instance using the `kissat` solver. We generate a batch of $n = 1,000$ queries per instance, distributed according to the following profile: 50% Satisfiability, 30% Model Counting, 10% Uniform Sampling, and 10% Direct Access.

The complexity of the queries is regulated by varying the size of the evidence set γ. For each query in the sequence, the number of conditioning literals $|\gamma|$ increments linearly from 1 up to a cap of 2% of the total variables, at which point it resets to 1. This cyclic approach ensures that the system is tested across a spectrum of conditioning densities, ranging from lightly to moderately constrained subspaces.

The content of the evidence γ is derived from the seed model ω but treated differently depending on the query type:

- **Model Counting, Sampling, and Direct Access:** For these tasks, we strictly select $|\gamma|$ literals from ω without modification. This guarantees that the defined subspace contains at least one solution (the seed itself), preventing trivial zero-count results and ensuring the reasoner must traverse a valid sub-circuit. For *uniform sampling* and *direct access*, the query index i denotes the position of the query in the batch $(q_1, \ldots, q_n)$: the i-th sampling query requests i samples, while the i-th direct access query requests the i-th model in the lexicographical order.
- **Satisfiability:** To generate a mix of satisfiable and unsat-

[6] https://zenodo.org/records/15837216

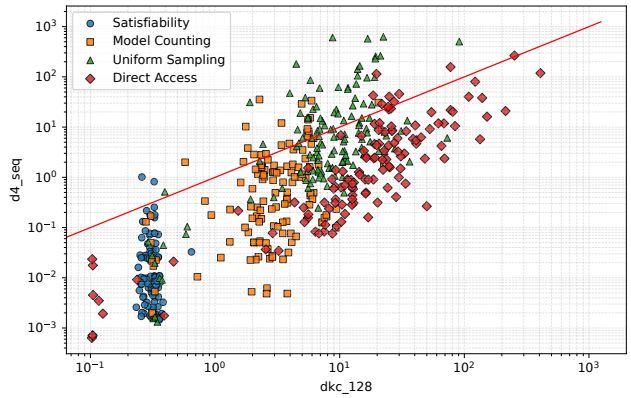

Figure 4: Scatter plot comparing the 4 different queries times of `dkc` with 128 cores (x-axis) versus the sequential baseline `d4` (y-axis). Points above the diagonal indicate instances where the distributed approach outperforms the sequential one.

isfiable queries, we select $|\gamma|$ literals from ω but randomly flip the polarity of each literal with a probability of 0.5. This "perturbed" evidence tests the solver's ability to distinguish between consistent subspaces and those that have been rendered inconsistent by conflicting constraints.

First, we observe that these benchmarks are relatively tractable, as detailed in the supplementary material. Specifically, the sequential version of `d4` successfully compiled 172 instances, whereas `dkc` solved 181. In 116 cases, both approaches require less than one second to complete. For such lightweight instances, the distributed architecture is less advantageous; the inherent communication overhead dominates the runtime and cannot be effectively amortized. However, on harder instances, `dkc` outperforms the sequential baseline. Since these results align with the trends observed in the previous section, we do not discuss them further. This also reflects the total offline-plus-online cost: `d4` is preferable on very small instances, whereas the distributed approach becomes beneficial, and sometimes necessary, when sequential compilation reaches the timeout or memory limit. Instead, the remainder of this evaluation focuses on the instances solved by both approaches, allowing for a direct comparison of the reasoning engine's efficiency.

Figure 4 presents a detailed scatter plot comparing the runtime of the sequential baseline `d4` against the distributed `dkc` configuration (128 cores). In this analysis, we decompose the workload to isolate the performance characteristics of each query; each data point represents the *cumulative* wall-clock time required to answer all queries of a specific type for a single instance. The results reveal a nuanced landscape where the benefits of parallelism are governed by the computation-to-communication ratio. In contrast to the compilation phase, where `dkc` demonstrated clear superiority, the query processing phase exhibits a different trend: a significant portion of data points lie below the diagonal. This indicates that for these tractable benchmarks, the inherent overhead of the distributed architecture often masks its computational advantages. However, this result must be contextualized. Since the reported times aggregate 1,000 queries, the average processing time per query is often subsecond. In this low-latency regime, network communication becomes the dominant bottleneck.

More specifically, regarding *satisfiability* (blue circles), we observe that the entire batch of queries is typically handled in under 0.1 seconds. In this ultra-low latency regime, the distributed reasoner cannot compete; the mandatory overhead of broadcasting requests and reducing results dominates the execution time, rendering parallelism ineffective.

For *model counting* (orange squares), the landscape is more nuanced. While a majority of points remain below the diagonal, we observe a shift for instances where the sequential solving time exceeds 1 second. In these cases, the local aggregation workload on each worker is substantial enough to mask the communication latency. Because the time is spent on local computation rather than network synchronization, the distributed approach begins to yield a net benefit.

The most significant performance penalty is observed for *direct access* (red diamonds). As detailed in Algorithm 2, retrieving a single model requires a sequence of model counting queries (one per variable). For a formula with N variables, this necessitates N distinct round-trip communications between the Master and the cluster. Since the individual counting operations are extremely fast, the total runtime is dominated by the accumulation of these synchronization barriers. Consequently, scaling to 128 cores yields diminishing returns, as the latency of synchronizing a larger cluster exacerbates the cumulative delay.

In contrast, the architecture demonstrates robustness for *uniform sampling* (green triangles), where the performance distribution is significantly more favorable than direct access. This validates our routing-based strategy: once the initial counts are cached, the Master directs sampling requests to specific workers rather than broadcasting every step. This unicast mechanism minimizes global synchronization overhead, allowing `dkc` to effectively leverage the cluster's aggregate memory bandwidth.

In summary, while `dkc` incurs overhead for lightweight queries on tractable instances, it offers a scalable solution for memory-intensive reasoning tasks where the size of the compiled circuit precludes single-node execution.

6 Conclusion and Perspectives

In this work, we presented `dreasoner`, a novel distributed architecture that unifies high-performance knowledge compilation with real-time query processing. By leveraging the Cube-and-Conquer paradigm, our approach decomposes the search space into disjoint fragments, constructing an implicit global d-DNNF across a cluster of worker nodes. Unlike traditional distributed model counters that terminate after a single counting operation, `dkc` transitions into a persistent state, transforming the cluster into a query engine.

Our experimental evaluation clearly demonstrates that `dreasoner` achieves robust scalability up to 128 cores. While communication overhead renders the distributed approach less effective for trivial instances, it significantly outperforms the state-of-the-art sequential solver `d4` on computationally intensive benchmarks. Notably, the distributed

architecture surpasses the sequential baseline with as few as two active workers. Furthermore, despite the inherent network latency, we showed that the system efficiently handles complex member queries, such as uniform sampling and direct access, by caching intermediate counts and amortizing the compilation cost across batched requests.

Future work will focus on extending the query language to support weighted model counting, thereby broadening the applicability of `dreasoner` to probabilistic reasoning. Additionally, we aim to optimize the latency of iterative queries like direct access, where network round-trips currently dominate the execution time. We plan to investigate protocol optimizations inspired by *branch prediction* mechanisms. By employing speculative execution or lookahead strategies, the Master could predict the likely traversal path and speculatively resolve subsequent variable assignments, effectively reducing message volume and masking communication latency. Finally, we will explore the integration of GPU-accelerated arithmetic circuit representations (Maene, Derkinderen, and Martires 2025) to enhance throughput.

Acknowledgments

This work has been partly supported by the CERADOC project of the French National Agency for Research (ANR-25-CE23-3078), Jilin province philosophy and social sciences project under Grant 2023ZD15.

AI Declaration

The authors declare that no generative AI tools or AI-assisted technologies were used in writing, editing, revising, or preparing this manuscript.

References

Astesana, J.; Cosserat, L.; and Fargier, H. 2010. Constraint-based vehicle configuration: A case study. In *22nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2010, Arras, France, 27-29 October 2010 - Volume 1*, 68–75. IEEE Computer Society.

Bagan, G.; Durand, A.; Grandjean, E.; and Olive, F. 2008. Computing the jth solution of a first-order query. *RAIRO Theor. Informatics Appl.* 42(1):147–164.

Baluta, T.; Chua, Z. L.; Meel, K. S.; and Saxena, P. 2021. Scalable quantitative verification for deep neural networks. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, 312–323. IEEE.

Bart, A.; Koriche, F.; Lagniez, J.; and Marquis, P. 2016. An improved CNF encoding scheme for probabilistic inference. In Kaminka, G. A.; Fox, M.; Bouquet, P.; Hüllermeier, E.; Dignum, V.; Dignum, F.; and van Harmelen, F., eds., *ECAI 2016 - 22nd European Conference on Artificial Intelligence, 29 August-2 September 2016, The Hague, The Netherlands - Including Prestigious Applications of Artificial Intelligence (PAIS 2016)*, volume 285 of *Frontiers in Artificial Intelligence and Applications*, 613–621. IOS Press.

Biere, A.; Heule, M.; van Maaren, H.; and Walsh, T., eds. 2021. *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press.

Biere, A.; Faller, T.; Fazekas, K.; Fleury, M.; Froleyks, N.; and Pollitt, F. 2024. Cadical 2.0. In Gurfinkel, A., and Ganesh, V., eds., *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*, volume 14681 of *Lecture Notes in Computer Science*, 133–152. Springer.

Brachman, R. J., and Levesque, H. J. 2004. *Knowledge Representation and Reasoning*. Elsevier.

Bringmann, K.; Carmeli, N.; and Mengel, S. 2025. Tight fine-grained bounds for direct access on join queries. *ACM Trans. Database Syst.* 50(1):1:1–1:44.

Burchard, J.; Schubert, T.; and Becker, B. 2016. Distributed parallel #sat solving. In *2016 IEEE International Conference on Cluster Computing, CLUSTER 2016, Taipei, Taiwan, September 12-16, 2016*, 326–335. IEEE Computer Society.

Cadoli, M., and Donini, F. M. 1997. A survey on knowledge compilation. *AI Commun.* 10(3-4):137–150.

Carmeli, N.; Tziavelis, N.; Gatterbauer, W.; Kimelfeld, B.; and Riedewald, M. 2023. Tractable orders for direct access to ranked answers of conjunctive queries. *ACM Trans. Database Syst.* 48(1):1:1–1:45.

Chavira, M., and Darwiche, A. 2008. On probabilistic inference by weighted model counting. *Artif. Intell.* 172(6-7):772–799.

Cook, S. A. 1971. The complexity of theorem-proving procedures. In Harrison, M. A.; Banerji, R. B.; and Ullman, J. D., eds., *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, 151–158. ACM.

Darwiche, A., and Marquis, P. 2002. A knowledge compilation map. *J. Artif. Intell. Res.* 17:229–264.

Darwiche, A. 2002. A compiler for deterministic, decomposable negation normal form. In Dechter, R.; Kearns, M. J.; and Sutton, R. S., eds., *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada*, 627–634. AAAI Press / The MIT Press.

Darwiche, A. 2004. New advances in compiling CNF into decomposable negation normal form. In de Mántaras, R. L., and Saitta, L., eds., *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, 328–332. IOS Press.

Darwiche, A. 2023. Logic for explainable AI. In *38th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2023, Boston, MA, USA, June 26-29, 2023*, 1–11. IEEE.

Fargier, H., and Marquis, P. 2006. On the use of partially ordered decision graphs in knowledge compilation and quantified boolean formulae. In *Proceedings, The Twenty-First*

National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA, 42–47. AAAI Press.

Hamann, M., and Strasser, B. 2018. Graph bisection with pareto optimization. *ACM J. Exp. Algorithmics* 23.

Heradio, R.; Perez-Morago, H.; Fernández-Amorós, D.; Bean, R.; Cabrerizo, F. J.; Cerrada, C.; and Herrera-Viedma, E. 2016. Binary decision diagram algorithms to perform hard analysis operations on variability models. In Fujita, H., and Papadopoulos, G. A., eds., *New Trends in Software Methodologies, Tools and Techniques - Proceedings of the Fifteenth SoMeT_16, Larnaca, Cyprus, 12-14 September 2016*, volume 286 of *Frontiers in Artificial Intelligence and Applications*, 139–154. IOS Press.

Kiesel, R., and Eiter, T. 2023. Knowledge compilation and more with SharpSAT-TD. In Marquis, P.; Son, T. C.; and Kern-Isberner, G., eds., *Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning, KR 2023, Rhodes, Greece, September 2-8, 2023*, 406–416.

Kübler, A.; Zengler, C.; and Küchlin, W. 2010. Model counting in product configuration. In Lynce, I., and Treinen, R., eds., *Proceedings First International Workshop on Logics for Component Configuration, LoCoCo 2010, Edinburgh, UK, 10th July 2010*, volume 29 of *EPTCS*, 44–53.

Lagniez, J., and Lonca, E. 2025. Enhancing query efficiency for D-DNNF representations through preprocessing. In Casini, G.; Dundua, B.; and Kutsia, T., eds., *Logics in Artificial Intelligence - 19th European Conference, JELIA 2025, Kutaisi, Georgia, September 1-4, 2025, Proceedings, Part II*, volume 16094 of *Lecture Notes in Computer Science*, 125–140. Springer.

Lagniez, J., and Marquis, P. 2017a. An improved decision-DNNF compiler. In Sierra, C., ed., *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, 667–673. ijcai.org.

Lagniez, J., and Marquis, P. 2017b. An improved decision-DNNF compiler. In Sierra, C., ed., *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, 667–673. ijcai.org.

Lagniez, J., and Marquis, P. 2017c. On preprocessing techniques and their impact on propositional model counting. *J. Autom. Reason.* 58(4):413–481.

Lagniez, J.; Lonca, E.; and Marquis, P. 2020. Definability for model counting. *Artif. Intell.* 281:103229.

Lagniez, J.; Marquis, P.; and Szczepanski, N. 2018. DMC: A distributed model counter. In Lang, J., ed., *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, 1331–1338. ijcai.org.

Maene, J.; Derkinderen, V.; and Martires, P. Z. D. 2025. Klay: Accelerating arithmetic circuits for neurosymbolic AI. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Muise, C. J.; McIlraith, S. A.; Beck, J. C.; and Hsu, E. I. 2012. Dsharp: Fast d-DNNF compilation with sharpSAT. In Kosseim, L., and Inkpen, D., eds., *Advances in Artificial Intelligence - 25th Canadian Conference on Artificial Intelligence, Canadian AI 2012, Toronto, ON, Canada, May 28-30, 2012. Proceedings*, volume 7310 of *Lecture Notes in Computer Science*, 356–361. Springer.

Oh, J.; Gazzillo, P.; and Batory, D. S. 2019. *t*-wise coverage by uniform sampling. In Berger, T.; Collet, P.; Duchien, L.; Fogdal, T.; Heymans, P.; Kehrer, T.; Martinez, J.; Mazo, R.; Montalvillo, L.; Salinesi, C.; Tërnava, X.; Thüm, T.; and Ziadi, T., eds., *Proceedings of the 23rd International Systems and Software Product Line Conference, SPLC 2019, Volume A, Paris, France, September 9-13, 2019*, 15:1–15:4. ACM.

Plaisted, D. A. 2015. History and prospects for first-order automated deduction. In Felty, A. P., and Middeldorp, A., eds., *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*, 3–28. Springer.

Plazar, Q.; Acher, M.; Perrouin, G.; Devroey, X.; and Cordy, M. 2019. Uniform sampling of SAT solutions for configurable systems: Are we there yet? In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*, 240–251. IEEE.

Rungta, N. 2022. A billion SMT queries a day (invited paper). In Shoham, S., and Vizel, Y., eds., *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I*, volume 13371 of *Lecture Notes in Computer Science*, 3–18. Springer.

Sharma, S.; Gupta, R.; Roy, S.; and Meel, K. S. 2018. Knowledge compilation meets uniform sampling. In Barthe, G.; Sutcliffe, G.; and Veanes, M., eds., *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018*, volume 57 of *EPiC Series in Computing*, 620–636. EasyChair.

Sundermann, C.; Heß, T.; Nieke, M.; Bittner, P. M.; Young, J. M.; Thüm, T.; and Schaefer, I. 2024. Evaluating state-of-the-art #sat solvers on industrial configuration spaces. In Rabiser, R.; Wimmer, M.; Groher, I.; Wortmann, A.; and Wiesmayr, B., eds., *Software Engineering 2024, Fachtagung des GI-Fachbereichs Softwaretechnik, Linz, Austria, February 26 - March 1, 2024*, volume P-343 of *LNI*, 67–68. Gesellschaft für Informatik e.V.

Xu, Z.; Yin, M.; and Lagniez, J. 2025. An embarrassingly parallel model counter. In *Proceedings of the 22nd International Conference on Principles of Knowledge Representation and Reasoning, KR 2025, Melbourne, Australia. November 11-17, 2025*.