HermesBDD: A Multi-Core and Multi-Platform Binary Decision Diagram Package

Luigi Capogrosso, Luca Geretti, Marco Cristani, Franco Fummi, Tiziano Villa
Department of Computer Science, University of Verona, Italy
name.surname@univr.it

***Abstract*—BDDs are representations of a Boolean expression in the form of a directed acyclic graph. BDDs are widely used in several fields, particularly in model checking and hardware verification. There are several implementations for BDD manipulation, where each package differs depending on the application. This paper presents *HermesBDD*: a novel multi-core and multi-platform binary decision diagram package focused on high performance and usability. *HermesBDD* supports a static and dynamic memory management mechanism, the possibility to exploit lock-free hash tables, and a simple parallel implementation of the IF-THEN-ELSE procedure based on a higher-level wrapper for threads and futures. *HermesBDD* is completely written in C++ with no need to rely on external libraries and is developed according to software engineering principles for reliability and easy maintenance over time. We provide experimental results on the n-Queens problem, the de-facto SAT solver benchmark for BDDs, demonstrating a significant speedup of $18.73\times$ over our non-parallel baselines, and a remarkable performance boost w.r.t. other state-of-the-art BDDs packages.**

***Index Terms*—Binary Decision Diagrams, Boolean Functions, Parallel Algorithms, Multi-Platform Package**

I. INTRODUCTION

Binary decision diagrams (BDDs) were introduced by Akers [1] and developed by Bryant [2] and provide a data structure for representing and manipulating Boolean functions. There are several implementations of BDD packages in the literature (details in Sec. II), but they focus mainly on package performance regarding speedup and memory efficiency. However, we believe that performance is only one aspect to judge a BDD package. Other aspects, such as (not necessarily in order of importance), *functionality*, *robustness*, *reliability*, *portability*, and *documentation*, matter as well. According to these principles, we developed *HermesBDD*: a novel multi-core and multi-platform binary decision diagram package focused on high performance and usability. It supports a static and dynamic memory management mechanism, the possibility to exploit lock-free hash tables, and a parallel implementation of the IF-THEN-ELSE procedure based on a higher-level wrapper for threads and futures. Additionally, *HermesBDD* presents a well-documented source code, it is completely written in C++ with no need to rely on external libraries, and it is developed according to engineering principles such as testability, code coverage, and continuous integration.

The work has been partially supported by the Italian Ministry of Education, University and Research (MIUR) with the grant "Dipartimenti di Eccellenza" 2018-2022, and by the project INdAM, GNCS 2020 (Strategic Reasoning and Automated Synthesis of Multi-Agent Systems) funded by MIUR (Italian Ministry of Education, University and Research).

We provide experimental results on the n-Queens problem showing how our multi-core implementation improves the performance over our non-parallel baselines and how the different memory management techniques affect the overall speedup. Finally, we compare *HermesBDD* with three of the best state-of-the-art BDD libraries, *i.e.*, CUDD [3], Sylvan [4], and BuDDy [5], demonstrating a remarkable speedup boost. The experiments demonstrate the goodness of the proposed package, but given space constraints, extensive benchmarking will be the subject of future work.

In summary, the contributions of *HermesBDD*[1] are:

- A computationally faster BDD package, by exploiting multi-threading for parallel processing and concurrent access to a BDD;
- Multi-platform compatibility (Windows, Linux, and macOS) to accommodate integration within tools from different environments;
- Support for a static and dynamic memory management mechanism, the possibility to exploit lock-free hash tables, and a novel parallel implementation of the IF-THEN-ELSE procedure based on a higher-level wrapper for threads and futures;
- High usability and robustness by design, thanks to a development based on engineering principles such as code coverage and continuous integration, along with independence from external software to offer high usability, reliability, and easy maintenance over time.

II. RELATED WORK

In this section, we provide an overview of the most widely used BDD libraries. For a survey on early packages see [6].

CUDD [3] stands for Colorado University Decision Diagram. It is a single-core package for the manipulation of BDDs, algebraic decision diagrams (ADDs), and Zero-suppressed binary decision diagrams (ZDDs) written in C, with a C++ wrapper.

BuDDy [5] is a BDD single-core library written in C, with many highly efficient vectorized BDD operations, dynamic variable reordering, automated garbage collection, a C++ interface with automatic reference counting, and much more.

Biddy [7] is a BDD package under GPL license, whose most distinguishing features are its specially designed C interface and a novel implementation of automatic garbage collection.

[1]https://luigicapogrosso.github.io/HermesBDD

Algorithm 1: P_ITE(). The multi-core IF-THEN-ELSE.

```
1  x ← TOP(f, g, h);
2  future_t ← ASYNC(ITE, f_x, g_x, h_x);
3  t ← future_t.AWAIT.RESULT();
4  future_e ← ASYNC(ITE, f_x', g_x', h_x');
5  e ← future_e.AWAIT.RESULT();
```

CacBDD [8] is a single-core C++ BDD package that implements dynamic cache management, which takes into account the hit rate of the computed table and the available memory.

BeeDeeDee [9] is a thread-safe Java library for BDD manipulation. BeeDeeDee allows clients to share a single factory of BDDs, in real parallelism and to reduce the memory footprint of their overall execution at a very low synchronization cost.

Sylvan [4] is a parallel BDD library written in C that provides scalable parallel execution of the standard BDD operations. It supports custom decision diagram terminal types, and it also implements operations on a specialized list of decision diagrams for model-checking.

DecisionDiagrams is a single-core implementation for numerous variants of BDDs that is used at Microsoft Research. Written in C#, currently, it maintains 100% code coverage. The library is based on a cache-optimized implementation of decision diagrams [10].

Based on the work of Lars Arge [11], Adiar [12] is a single-core BDD package that makes use of time-forward processing to improve the I/O complexity of BDD manipulation. This achieves efficient manipulation of BDDs, even when they outgrow the memory limit of a given machine.

In [13], Miyasaka *et al*. describe a simple BDD package without dynamic variable reordering, which is much faster than a conventional BDD package with reordering. The proposed BDD package is used in logic optimization with permissible functions. Moreover, in [14], Miyasaka presents a framework to compare BDD packages through auto-tuning.

III. METHODOLOGY

In this section, we present the algorithms and the techniques developed for our efficient parallelization. Due to space constraints, for a technical glance and interesting properties of BDDs, refer to [1], [2].

A. The Multi-Core IF-THEN-ELSE Algorithm

In *HermesBDD* we parallelize the IF-THEN-ELSE function treating the two recursive calls as independent tasks. Starting from the task-based parallel flow of [15], we rewrote it with C++ primitives without using external libraries, which allowed us to introduce a more efficient hash table management mechanism (as we will see in Sec. IV-B). In particular, this flow gives us a simple way to use multiple threads through primitives: instead of making a recursive call to execute the IF-THEN-ELSE function, start a thread at each recursive step, then wait for the thread to finish. With this implementation, the only synchronization between workers is when the results of

Algorithm 2: UPDATE_UTABLE(). Creates a BDD node using the unique hash table to ensure that there are no duplicates.

```
   Data: (x, t, e)
1  if t = e then
2  |  return e;
3  end
4  if IS_COMPLEMENTED(e) then
5  |  v ← NODE(x, t, e);
6  |  n ← LOOKUP_OR_CREATE(v);
7  |  return COMPLEMENT(n);
8  else
9  |  v ← NODE(x, t, e);
10 |  return LOOKUP_OR_CREATE(v);
11 end
```

suboperations are stored in the unique hash table. This table is shared globally, in order to prevent workers from computing a suboperation that was finished already by some other worker. This technique seems to be best suited to parallelize BDDs. We have tried and compared other implementations, *i.e*., parallelism on different levels of the tree. But, given the nature of BDDs, these techniques present an important task overhead, vs. a negligible, or often negative, speedup. Alg. 1 shows the pseudocode of our implementation. Furthermore, *HermesBDD* provides an option to the user, at compile-time, to decide whether to use the multi-core or sequential implementation of the IF-THEN-ELSE algorithm.

To this end, we use the C++ `std::async()` function, which is a high-level wrapper for threads and futures, followed by the matching function to retrieve the results of the computation. Standard C++ provides `std::thread()` which is a fairly low-level construct, and so its usage is often more cumbersome and error-prone than desired. Instead, `std::async()` automatically creates a thread to call the thread function and it conveniently returns an object `std::future` without the hurdle of manual thread management and decoupling the task from the result.

To ensure that the results are canonical reduced BDDs, we use the UPDATE_UTABLE() method, as shown in Alg. 2. Specifically, the LOOKUP_OR_CREATE() function checks atomically if data is already in the unique hash table, and if not, it adds it. Finally, in order to enforce canonicity (*i.e*., complement only on 1 edge), we use the functions IS_COMPLEMENTED() and COMPLEMENT().

B. A Lock-Free Unique Hash Table for Multi-Core BDDs

Traditionally, concurrency issues, such as data race, are solved by locks, providing mutual exclusion. Since blocked processes must wait, locks have a negative impact on the speedup of parallel programs. A lot of literature has been dedicated to developing non-blocking algorithms, specifically, Herlihy *et al*., in [16], distinguish between *lock-free*, *wait-free*, and *lock-less* algorithms.

Algorithm 3: LOOKUP_OR_CREATE(). Insert an entry into the unique hash table.

Data: d
1 $hash \leftarrow$ CALCULATE_HASH(d);
2 $index \leftarrow hash \,\%\, elems$;
3 $table\_slot \leftarrow table[index]$;
4 SPINLOCK_PROTECTOR($table\_slot$);
5 INSERT_IN_UTABLE(d)

In particular, we use a lock-free unique hash table, which is implemented using the `std::atomic_flag`. Based on this class, we build a spinlock in order to protect the critical section. Specifically, a spinlock is a lock that causes a thread trying to acquire it to wait in a loop while repeatedly checking whether the lock is available. The use of spinlocks is particularly recommended when the critical section is supposed to perform a minimal amount of work, *i.e.*, the spinlock is held for a very short period of time, as in our case. Also, it operates faster compared to mutex since context switching is reduced. Spinlock does not cause the thread to be preempted but instead, it keeps on spinning until the lock on the resource is released.

The pseudocode for inserting a new value in the unique hash table is given in Alg. 3. This computes the hash, calculates the index, acquires the lock, and finally writes the data into the table. The GET_FROM_UTABLE() algorithm works in exactly the same way. This is not reported, since it differs from Alg. 3 only by line 5, where it calls the function that compares the parameters and returns the result value.

C. The Memory Management Mechanism

BDD algorithms are considered memory intensive since they have little computation for each unit of memory access. Hence, memory allocation techniques for the tables play an important role, and have a great effect, on the performance of the implementations of a BDD package. In *HermesBDD*, we implemented both dynamic and static memory allocation techniques in order to exploit fine-grain parallelism. Also in this case, at compile-time, *HermesBDD* provides an option for the users in order to select the dynamic or the static memory allocation mechanism.

As dynamic memory management, we implemented a simple but effective technique based on doubling the memory space required by the tables. At the beginning of the process, M bytes of memory are allocated to store N nodes. If this space is not enough during program execution, more space of size $M * 2$ will be allocated. Since the table is shared by all nodes created by the library, this allows reusing memory. For simplicity and efficiency, there is no strategy for cleaning up the slot table after a given node is removed.

In the static allocation technique, instead, a contiguous slice of M bytes of memory is reserved at the start of the process. In this case, the variables get allocated permanently, until the program executes or the function call finishes, and once the memory is allocated, the memory size cannot change, so it is more efficient than dynamic allocation.

TABLE I
HermesBDD NON-PARALLEL EXECUTION TIME BASED ON THE n-QUEENS PROBLEM COMPLEXITY. VALUES ON THE AVERAGE OF 50 SAMPLES USING THE STATIC MEMORY ALLOCATION ON A 32-CORE MACHINE.

	$\mathbf{6 \times 6}$	$\mathbf{7 \times 7}$	$\mathbf{8 \times 8}$
***HermesBDD* baseline**	15.85	79.99	423.33

TABLE II
HermesBDD PARALLEL SPEEDUP BASED ON THE n-QUEENS PROBLEM COMPLEXITY AND THE N. OF CORES. SPEEDUP WAS OBTAINED FROM THE AVERAGE OF 50 SAMPLES USING THE STATIC MEMORY ALLOCATION.

Chessboard	2 Core	8 Core	16 Core	32 Core
$\mathbf{6 \times 6}$	$1.59\times$	$2.00\times$	$2.38\times$	$2.95\times$
$\mathbf{7 \times 7}$	$2.08\times$	$2.75\times$	$3.96\times$	$4.69\times$
$\mathbf{8 \times 8}$	$4.56\times$	$9.25\times$	$10.69\times$	$\mathbf{\underline{18.73\times}}$

IV. EXPERIMENTS

In this section, we perform quantitative and qualitative analyses to demonstrate the potentiality of our *HermesBDD* package. Experiments were carried out on a 64-bit 32-core AMD Ryzen Threadripper 1950X CPU 3.4GHz machine. The library is tested for compilation using GCC (minimum required: 10.2), Clang (minimum required: 11.0), and MSVC (minimum required: 19.20).

We ran benchmarks from the n-Queens problem, using a simple SAT solver based on [17]. In particular, we would like to emphasize that further experiments, as well as systematic comparisons with different parallel tools, will be the subject of future work. Here, also due to space constraints, we focus only on a single well-known problem; nonetheless, we aim to show the potentialities of *HermesBDD* with no interest in presenting a benchmarking paper on BDDs.

A. The Speedup Latency w.r.t. our Baselines

Tab. I shows the result on an average of 50 samples of our baselines on the n-Queens problems with the 6×6, 7×7, and 8×8 chessboards, using the static memory allocation mechanism.

Tab. II shows the result of our multi-core implementation in terms of speedup, where the speedup latency is computed as $S = T_{ms}(no\_parallel)/T_{ms}(parallel)$, on the same chessboards, using the same number of cores.

Specifically, comparing Tab. I with Tab. II, several facts emerge: *i)* smaller models (*e.g.*, 6×6 and 7×7 chessboard) have lower speedups w.r.t. larger models, which exhibit the best speedups. *ii)* This implies that the speedup is increasing with the size of the Boolean formula. *iii)* Finally, our implementation scales well with the number of cores as long as the problem is sufficiently complex, yielding a speedup of up to $18.73\times$ in the 8×8 case.

B. Comparison w.r.t. other BDD Packages

As a comparative approach, we consider CUDD, Sylvan, and BuDDy, which are three of the most important state-of-the-art BDD packages. Tab. III reports the results on an

TABLE III
HermesBDD W.R.T. CUDD [3], SYLVAN [4] AND BUDDY [5] EXECUTION TIME AND MEMORY SPACE REQUIRED. RESULTS WERE OBTAINED ON AVERAGE FROM 50 SAMPLES OF THE n-QUEENS PROBLEM, USING A 32-CORE MACHINE, AND THE STATIC MEMORY ALLOCATION.

	HermesBDD	**CUDD**	**Sylvan**	**BuDDy**
Time (ms)	**23.33**	26.90	33.75	46.50
Memory (GB)	**0.5**	1.3	1.9	0.7

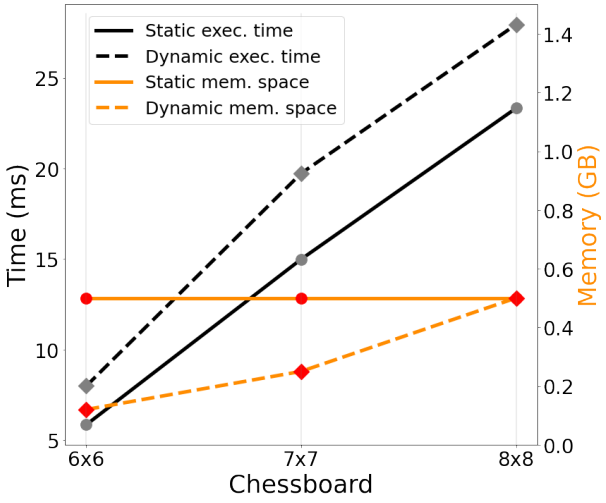

Fig. 1. *HermesBDD* results on the impact of memory allocation in terms of time and memory space, on an average of 50 samples of the n-Queens problem on a 32-core machine.

average of 50 samples of the n-Queens problem with an 8×8 chessboard, using a 32-core machine, and the static memory allocation mechanism. Specifically, we decided to use static memory allocation for this experiment because it is the memory management technique that allows us to push performance to the maximum in terms of computation time.

In summary, as we can see from Tab. III, the combination of using a lightweight approach for parallelism, a lock-free hash table technique, and a static memory allocation yields better performances both for execution time and memory space.

C. The Impact of Memory Allocation

In this experiment, we evaluate the impact of both static and dynamic memory allocation techniques in terms of execution time and memory space. Specifically, Fig. 1 shows the results of a single test in which we run, sequentially, the n-Queens problem on a 6×6, then on a 7×7, and finally on an 8×8 chessboard on an average of 50 samples of the n-Queens problem on a 32-core machine. As mentioned in Sec. III-A, thanks to the unique hash table the 7×7 case benefits from the 6×6 storage, just as the 8×8 case benefits from the 7×7 storage.

The behaviors explained in Sec. III-C are confirmed: a dynamic memory allocation guarantees a more efficient memory occupancy, but the overall execution time is slower than static memory allocation due to the presence of overhead caused by the management of the memory allocation at run time (*e.g.*, the increase of execution time in percentage is 36% for the 6×6 chessboard, down to 19% for the 8×8 chessboard). On the other hand, static memory allocation provides better performance in terms of execution time but at the expense of inefficient memory management (*e.g.*, in the case of the 6×6 chessboard, more MBs are allocated than required).

V. CONCLUSIONS

In this paper, we presented *HermesBDD*, a multi-core and multi-platform package for BDD manipulation. We designed and implemented three different algorithms to support a parallel implementation of BDD operations: a multi-core IF-THEN-ELSE procedure based on a higher-level wrapper for threads and futures, a lock-free hash table, and both a static and dynamic memory allocation mechanism.

The performance demonstrated a significant speedup. Thus, we can say that the experiments validate the proposed package, whereas more extensive benchmarking will be the subject of future work.

REFERENCES

[1] S. B. Akers, "Binary decision diagrams," *IEEE Computer Architecture Letters*, vol. 27, no. 06, pp. 509–516, 1978.

[2] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *Computers, IEEE Transactions on*, vol. 100, no. 8, pp. 677–691, 1986.

[3] F. Somenzi, "CUDD: CU decision diagram package-release 2.4. 0," *University of Colorado at Boulder*, 2012.

[4] T. v. Dijk and J. v. d. Pol, "Sylvan: Multi-core decision diagrams," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2015, pp. 677–691.

[5] J. Lind-Nielsen, "BuDDy: A binary decision diagram package." 1999.

[6] G. Janssen, "A consumer report on BDD packages," in *16th Symposium on Integrated Circuits and Systems Design, 2003. SBCCI 2003. Proceedings.* IEEE, 2003, pp. 217–222.

[7] R. Meolic, "Biddy-a Multi-platform Academic BDD Package." *J. Softw.*, vol. 7, no. 6, pp. 1358–1366, 2012.

[8] G. Lv, K. Su, and Y. Xu, "CacBDD: A BDD package with dynamic cache management," in *International Conference on Computer Aided Verification*. Springer, 2013, pp. 229–234.

[9] A. Lovato, D. Macedonio, and F. Spoto, "A thread-safe library for binary decision diagrams," in *International Conference on Software Engineering and Formal Methods*. Springer, 2014, pp. 35–49.

[10] G. Janssen, "Design of a pointerless BDD package," in *International Workshop on Logic Synthesis (IWLS)*, 2001.

[11] L. Arge, "The I/O-complexity of ordered binary-decision diagram manipulation," in *International Symposium on Algorithms and Computation*. Springer, 1995, pp. 82–91.

[12] S. C. Sølvsten, J. v. de Pol, A. B. Jakobsen, and M. W. B. Thomasen, "Adiar Binary Decision Diagrams in External Memory," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2022, pp. 295–313.

[13] Y. Miyasaka, A. Mishchenko, and M. Fujita, "A simple BDD package without variable reordering and its application to logic optimization with permissible functions," in *Proc. Int. Workshop Log. Synth.*, 2019, pp. 1–8.

[14] Y. Miyasaka and M. Fujita, "Auto-tuning framework for BDD packages," in *Proceedings of IWLS 2020*, 2020, pp. 66–73.

[15] T. Van Dijk, A. Laarman, and J. Van De Pol, "Multi-core BDD operations for symbolic reachability," *Electronic Notes in Theoretical Computer Science*, vol. 296, pp. 127–143, 2013.

[16] M. Herlihy, N. Shavit, V. Luchangco, and M. Spear, *The art of multiprocessor programming*. Newnes, 2020.

[17] C. Bright, J. Gerhard, I. Kotsireas, and V. Ganesh, "Effective problem solving using SAT solvers," in *Maple Conference*. Springer, 2020, pp. 205–219.

[18] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang, "Symbolic model checking: 10^{20} states and beyond," *Information and Computation*, vol. 98, no. 2, p. 142 – 170, 1992.

[19] Bresolin, Davide and Collins, Pieter and Geretti, Luca and Segala, Roberto and Villa, Tiziano and Gonzalez, Sanja Živanović, "A Computable and Compositional Semantics for Hybrid Automata," in *Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control*, ser. HSCC '20. New York, NY, USA: Association for Computing Machinery, 2020.

[20] Immler, Fabian and Althoff, Matthias and Benet, Luis and Chapoutot, Alexandre and Chen, Xin and Forets, Marcelo and Geretti, Luca and Kochdumper, Niklas and Sanders, David P. and Schilling, Christian, "Arch-comp19 category report: Continuous and hybrid systems with nonlinear dynamics," vol. 61, 2019, p. 41 – 61.

[21] Brayton, Robert and Carloni, Luca P. and Sangiovanni-Vincentelli, Alberto L. and Villa, Tiziano, "Design automation of electronic systems: Past accomplishments and challenges ahead [scanning the issue]," *Proceedings of the IEEE*, vol. 103, no. 11, pp. 1952–1957, 2015.

[22] G. Paul, A. Pal, and B. B. Bhattacharya, "On finding the minimum test set of a bdd-based circuit," in *Proceedings of the 16th ACM Great Lakes symposium on VLSI*, 2006, pp. 169–172.

[23] F. Ahishakiye, J. I. Requeno Jarabo, L. M. Kristensen, and V. Stolz, "Mc/dc test cases generation based on bdds," in *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*. Springer, 2021, pp. 178–197.

[24] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 8, no. 2, pp. 244–263, 1986.

[25] S. Chaki and A. Gurfinkel, "BDD-based symbolic model checking," in *Handbook of Model Checking*. Springer, 2018, pp. 219–245.

[26] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L.-J. Hwang, "Symbolic model checking: 1020 states and beyond," *Information and computation*, vol. 98, no. 2, pp. 142–170, 1992.

[27] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill, "Symbolic model checking for sequential circuit verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 4, pp. 401–424, 1994.

[28] T. Ball and S. K. Rajamani, "Bebop: a path-sensitive interprocedural dataflow engine," in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2001, pp. 97–103.

[29] S.-i. Minato, "Zero-suppressed BDDs and their applications," *International Journal on Software Tools for Technology Transfer*, vol. 3, no. 2, pp. 156–170, 2001.

[30] A. Aziz, S. Taşiran, and R. K. Brayton, "BDD variable ordering for interacting finite state machines," in *Proceedings of the 31st annual Design Automation Conference*, 1994, pp. 283–288.

[31] D. M. Miller and M. A. Thornton, "QMDD: A decision diagram structure for reversible and quantum circuits," in *36th International Symposium on Multiple-Valued Logic (ISMVL'06)*. IEEE, 2006, pp. 30–30.

[32] R. E. Bryant, "Binary decision diagrams and beyond: Enabling technologies for formal verification," in *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*. IEEE, 1995, pp. 236–243.

[33] W. Chan, R. J. Anderson, P. Beame, D. H. Jones, D. Notkin, and W. E. Warner, "Optimizing symbolic model checking for statecharts," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 170–190, 2001.

[34] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.

[35] T. Rauber and G. Rünger, *Parallel programming*. Springer, 2013.

[36] R. E. Bryant, "Symbolic boolean manipulation with ordered binary-decision diagrams," *ACM Computing Surveys (CSUR)*, vol. 24, no. 3, pp. 293–318, 1992.

[37] R. K. Brayton, G. D. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic minimization algorithms for VLSI synthesis*. Springer Science & Business Media, 1984, vol. 2.

[38] F. M. Brown, *Boolean reasoning: the logic of Boolean equations*. Courier Corporation, 2003.

[39] M. Elbayoumi, M. S. Hsiao, and M. ElNainay, "A novel concurrent cache-friendly binary decision diagram construction for multi-core platforms," in *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2013, pp. 1427–1430.

[40] D. Miller and R. Drechsler, "Negation and duality in reduced ordered binary decision diagrams," in *1997 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, PACRIM. 10 Years Networking the Pacific Rim, 1987-1997*, vol. 2. IEEE, 1997, pp. 692–696.

[41] ——, "Dual edge operations in reduced ordered binary decision diagrams," in *ISCAS'98. Proceedings of the 1998 IEEE International Symposium on Circuits and Systems (Cat. No. 98CH36187)*, vol. 6. IEEE, 1998, pp. 159–162.

[42] S.-i. Minato, N. Ishiura, and S. Yajima, "Shared binary decision diagram with attributed edges for efficient boolean function manipulation," in *27th ACM/IEEE Design Automation Conference*. IEEE, 1990, pp. 52–57.

[43] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a BDD package," in *27th ACM/IEEE design automation conference*. IEEE, 1990, pp. 40–45.

[44] F. Somenzi, "Binary decision diagrams," *NATO ASI SERIES F COMPUTER AND SYSTEMS SCIENCES*, vol. 173, pp. 303–368, 1999.

[45] T. Stornetta and F. Brewer, "Implementation of an efficient parallel bdd package," in *33rd Design Automation Conference Proceedings, 1996*. IEEE, 1996, pp. 641–644.

[46] B. Yang, R. E. Bryant, D. R. O'Hallaron, A. Biere, O. Coudert, G. Janssen, R. K. Ranjan, and F. Somenzi, "A performance study of BDD-based model checking," in *International conference on formal methods in computer-aided design*. Springer, 1998, pp. 255–289.

[47] C. Meinel and T. Theobald, *Algorithms and Data Structures in VLSI Design: OBDD-foundations and applications*. Springer Science & Business Media, 1998.

[48] I. Rivin, I. Vardi, and P. Zimmermann, "The n-queens problem," *The American Mathematical Monthly*, vol. 101, no. 7, pp. 629–639, 1994.

[49] D. Kunkle, V. Slavici, and G. Cooperman, "Parallel disk-based computation for large, monolithic binary decision diagrams," in *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, 2010, pp. 63–72.

APPENDIX

A. Introduction

BDDs enabled breakthroughs in the area of formal verification, by advancing the capability of state-of-the-art *model-checking* by several orders of magnitude in terms of the size of state spaces that can be successfully explored. While BDDs have been applied successfully to discrete systems [18], exploiting them to compute the evolution of cyber-physical systems [19] allows introducing an efficient discretized representation of states even for non-linear dynamical systems [20]. So BDDs and SAT are required engines in the CAD toolbox [21]. Finally, another area in which these diagrams are particularly useful is that of *test generation*, *i.e.*, finding a set of inputs that can be used to confirm that a given implementation performs correctly [22], [23].

Model-checking (or property-checking) is the method that creates abstractions of complex systems to verify if their function meets a given specification. Algorithms for model-checking were initially implemented in an explicit-state manner [24]. This meant that all automata involved in verification were represented listing their states explicitly. From a theoretical perspective, the data structure used to represent the automata makes no difference, but from a practical perspective, it means that model checkers could only handle automata with few reachable states [25]. So, practical hardware verification via model-checking called for a computational breakthrough, which appeared in the form of BDD-based symbolic model-checking [26], *i.e.*, representing all states using functions, instead of storing them individually. This fact enabled practical verification of industrial systems, beginning with hardware [27], and extending to software [28]. Moreover, it led to important developments for decision diagrams, such as new types of BDDs (*e.g.*, [29]) and new variable-ordering heuristics (*e.g.*, [30]). For example, QMDDs [31] are a kind of BDDs specifically designed to represent and manipulate matrices that allow compact simulation and specification of reversible quantum gates and circuits.

Although direct exploration of parameters is possible in some cases, it is also always computationally intensive [32]. Therefore, techniques to increase the performance of model-checking tools are under constant development. Until the last decade, the usual policy to improve the memory and computation performance was to increase the CPU frequencies and implement various hardware and software optimizations [33]. However, with recent developments in parallel hardware architectures, the challenge to push the frontiers of BDD applicability calls for the deployment of BDD packages on multi-processor architectures. This can be accomplished using various techniques, such as better algorithms [34], and parallel computing using multiple processors [35]. In this sense, having a BDD package that can perform optimized operations through efficient algorithms, and perform them on multiple processors, will enable us to push the frontiers of formal verification with BDDs.

B. Background

In this section, we briefly review the background, definition, and interesting properties of BDDs.

1) Terminology: Let $\mathbf{x}$ denote a vector of Boolean variables $x_1, x_2, \ldots x_n$, and let $\mathbf{a}$ denote a vector of values $a_1, a_2, \ldots a_n$, $a_i \in \{0, 1\}$. Let $\mathbf{1}$ denote the function that always yields 1, and $\mathbf{0}$ the function that always yields 0.

We can define Boolean operations $\wedge, \vee, \oplus$, and $\neg$ over functions according to the Boolean operations on the underlying elements. So, for example, $f \wedge g$ is a function h such that $h(\mathbf{a}) = f(\mathbf{a}) \wedge g(\mathbf{a})$ for all $\mathbf{a}$ [36].

For function f, variable x_i and a binary value $b \in \{0, 1\}$, we can define a *restriction* of f as the function resulting when x_i is set to value b:

$$f|_{x_i \leftarrow b}(\mathbf{a}) = f(a_1, \ldots, a_{i-1}, b, a_{i+1}, \ldots, a_n) \tag{1}$$

In particular, the two restrictions of a function f w.r.t. a variable x_i referred to as *cofactors* of f w.r.t. x_i [37].

Given the two cofactors of f w.r.t. to variable x_i, the function can be reconstructed as:

$$f = (x_i \wedge f|_{x_i \leftarrow 1}) \vee (\neg x_i \wedge f|_{x_i \leftarrow 0}) \tag{2}$$

This identity is commonly referred to as *Shannon decomposition* of f w.r.t. x_i [38].

2) Definition: Any Boolean function $f : B^n \rightarrow B$, where n is the number of variables, and $B \in \{0, 1\}$, is a mapping from a n-bit binary vector, to a single bit output [39]. This function can be represented as a binary decision diagram, which is a directed acyclic graph (DAG) expressing the Shannon decomposition (Eq. 2) of a Boolean function.

In particular, BDDs are defined as tuples $[V, h, l, v]$ where V represents the set of internal vertices, $h, l : V \rightarrow V \cup \{0, 1\}$ are functions representing the high and low edges of a node, v indicates the variable associated to a vertex, and leaves can be 0 or 1.

A BDD is called *ordered* (OBDD) if each variable is encountered at most once along each path and variables appear in the same order on all paths. So, on every path from the root to a leaf, every variable is encountered at most once.

A BDD is called *reduced ordered* (ROBDD) if it contains no nodes with two identical child nodes and no duplicated sub-graphs. The advantage of a ROBDD is that it is canonical (*i.e.*, unique) for a particular function and variable order [2]. Any BDD can be reduced by applying the following two rules:

1) Eliminate redundant nodes.
2) Eliminate duplicated sub-graphs by sharing sub-graphs.

This property makes it useful in functional equivalence checking and other operations like functional technology mapping. A path from the root node to the 1-terminal represents a (possibly partial) variable assignment for which the represented Boolean function is true. As the path descends to a low (or high) child from a node, then that node's variable is assigned to 0 (or 1).

TABLE IV
IMPLEMENTATION OF BOOLEAN FUNCTIONS IN TERMS OF IF-THEN-ELSE OPERATORS.

No.	Boolean	ITE Op.	No.	Boolean	ITE Op.
0	0	0	8	$f \mp g$	$ITE(f, 0, g')$
1	$f \wedge g$	$ITE(f, g, 0)$	9	$f \Leftrightarrow g$	$ITE(f, g, g')$
2	$f \not\Rightarrow g$	$ITE(f, g', 0)$	10	$\overline{g}$	$ITE(g, 0, 1)$
3	f	f	11	$f \Leftarrow g$	$ITE(f, 1, g')$
4	$f \not\Leftarrow g$	$ITE(f, 0, g)$	12	$\overline{f}$	$ITE(f, 0, 1)$
5	g	g	13	$f \Rightarrow g$	$ITE(f, g, 1)$
6	$f \oplus g$	$ITE(f, g', g)$	14	$f \barwedge g$	$ITE(f, g', 0)$
7	$f \vee g$	$ITE(f, 1, g)$	15	1	1

3) Complemented Edges: To decrease memory usage and calculation time, sub-graphs can be reused to represent related Boolean functions by adding attributes to edges in a BDD. Several methods are mentioned in the literature, such as [40], [41].

Introduced by Minato [42] and Brace [43], complemented edges (also called negated edges), indicate that leaves 1 and 0 will be switched, negating the Boolean function represented by the node. If an edge is complemented, then it refers to the negation of the Boolean function that corresponds to the node to which the edge points (the Boolean function represented by the BDD rooted in the node). It allows performing complementation in constant time. For example, a ROBDD can be represented more compactly using complemented edges [44]. In order to guarantee the canonicity of BDDs with complemented edges, Minato [42] proposed the following constraints:

- Only 0 is used as the value of a leaf.
- No complemented edges are allowed on 0-edges.

4) The IF-THEN-ELSE *Operator:* We exploit the observation that all Boolean operators with two arguments can be expressed using the IF-THEN-ELSE operator, with arguments the two Boolean functions or their negations, or the constants 0 and 1. The IF-THEN-ELSE (ITE) is defined as follows:

$$\texttt{ITE}(f, g, h) = (f \wedge g) \vee (f' \wedge h) \tag{3}$$

where f, g and h are all Boolean functions. The ITE can be recursively computed as follows:

$$\texttt{ITE}(f, g, h) = \texttt{ITE}(v, \texttt{ITE}(f_v, g_v, h_v), \texttt{ITE}(f_{v'}, g_{v'}, h_{v'})) \tag{4}$$

where v is a Boolean variable on which the functions depend, f_v and $f_{v'}$ are the positive and negative cofactors for f, respectively (same holds for g and h). Such a recursive formulation also allows for efficient storing and indexing of BDDs. In fact, it forms the basic structure of the unique table, in which each entry stores a triplet (variable, right-pointer, left-pointer) [45].

We define the IF-THEN-ELSE normal form as a Boolean expression using only the IF-THEN-ELSE operator and the constants 0 and 1. In Tab. IV we report the equivalent IF-THEN-ELSE operator for each of the 16 possible Boolean operators on two variables.

Algorithm 4: ITE(). The IF-THEN-ELSE.

```
Data: (f, g, h)
Result: r
if terminal_case then
    return computed_result;
else
    if {(f, g, h), r} in cache then
        r ← GET_FROM_CACHE({(f, g, h), r});
        return r;
    else
        x ← TOP(f, g, h);
        t ← ITE(f_x, g_x, h_x);
        e ← ITE(f_x', g_x', h_x');
        if t = e then
            return t;
        end
        r ← UPDATE_UTABLE(x, t, e);
        UPDATE_CACHE({(f, g, h), r});
        return r;
    end
end
```

Alg. 4 presents the pseudocode of our sequential IF-THEN-ELSE implementation, based on [43]. The function takes three arguments, which are the pointers to BDDs representing the IF, THEN, and ELSE Boolean functions, and returns a pointer to the resulting BDD.

The algorithm starts by checking if a terminal case applies, where the result can be easily computed. If so, the result is returned. Otherwise, the algorithm checks in the computed table if the result has already been computed, and if so, it returns it. If not, the variable of the root node is determined through the TOP() function, and the cofactors of the original IF-THEN-ELSE operator are recursively computed. If they are equal, one of them is returned as the result. Otherwise, the function UPDATE_UTABLE() is used to check if the new BDD node has already been created. If so, a pointer to it is returned. If not, the BDD node is first created and then returned, and the information for it is inserted in the computed table to avoid future recomputations.

5) Efficiency Improvements: BDDs are typically irregular and require an unpredictable number of memory accesses with high demands on memory latency [46]. Researchers have spent much time and effort examining how they can represent these structures more efficiently.

Bryant *et al*. in [43] described various efficiency improvements that are now standard in a BDD implementation. They include complemented edges, hashing, caching, and garbage collection.

Moreover, the size of BDDs depends heavily on the chosen variable ordering. We suggest referring to [47] by Meinel *et al*. for a thorough discussion on variable ordering.

An approach to BDD implementation that discards the use of pointers in favor of simple arrays is discussed in G. Jansen's paper [10]. This approach is shown to outperform a pointer-based approach in memory and CPU efficiency, at the expense of more complicated garbage collection.

C. The n-Queens Problem

The eight-queens puzzle is the problem of placing eight chess queens mutually non-attacking on an 8×8 chessboard [48]. Thus, a solution requires that no two queens share the same row, column, or diagonal. In particular, this problem can be posed in more general terms through the following question: *Given n, in how many ways can n queens be placed on an $n \times n$ chessboard without threatening each other?*

Inspired by [49], we implement a solution algorithm by constructing a BDD row-by-row that represents whether the row is in a legal state. Then we count the number of satisfying assignments of the accumulation BDD.

D. Comparison w.r.t. other BDD Packages

There are several differences between the implementation in CUDD, Sylvan, and BuDDy, and the one in *HermesBDD* that makes it difficult to compare the performances. For example, CUDD, Sylvan, and *HermesBDD* use *reference counting* for garbage collection, while BuDDy uses the *mark-and-sweep* technique, which requires less accounting work. However, the preallocated tables were large enough that garbage collection did not occur in any of the packages. BuDDy also uses several other optimizations, such as increased memory locality by storing related BDD nodes near each other in the hash table, while Sylvan and *HermesBDD* store BDD nodes at the same position as the hash in the hash table. Finally, BuDDy and CUDD are not thread-safe, as opposed to Sylvan and *HermesBDD*. Finally, we can say that our implementation has an advantage compared to Sylvan's because it features a native implementation of parallelism. Indeed, C++ primitives guarantee complete control of all task-based aspects, whereas Sylvan relying on an external library to handle parallelism does not have these guarantees. This allows us to implement a lock-free mechanism for the management of the hash table, which is more efficient than the lock-less type mechanism used by Sylvan. Moreover, interested readers can refer to [6] for a comprehensive discussion on the differences between the implementation of CUDD and BuDDy.

E. Conclusions

The performance demonstrated a significant speedup, such as $18.73\times$ over our non-parallel baselines, and a remarkable performance boost w.r.t. other state-of-the-art BDD packages, such as $1.41\times$ w.r.t. Sylvan. Thus, we can say that the experiments validate the proposed package, whereas more extensive benchmarking will be the subject of future work. Additionally, to fill a gap in the literature, *HermesBDD* comes with a well-documented source code, does not rely on external libraries, and has been designed according to software engineering principles such as CI/CD for reliability and easy maintenance over time.

In future work, we will fine-tune the current implementation by adding functions for composition, Boolean quantification, and BDD minimization, exploiting further optimizations.