A Comparative Study of BDD Packages for Probabilistic Symbolic Model Checking

Tom van Dijk[1], Ernst Moritz Hahn[2], David N. Jansen[3], Yong Li[2], Thomas Neele[1], Mariëlle Stoelinga[1], Andrea Turrini[2], and Lijun Zhang[2]

[1] University of Twente, Formal Methods & Tools, Enschede, The Netherlands
[2] State Key Laboratory of Computer Science, Institute of Software, CAS, Beijing, China
[3] Radboud Universiteit, Model-Based System Development, Nijmegen, The Netherlands

Abstract Symbolic data structures using Binary Decision Diagrams (BDDs) have been successfully used in the last decades to analyse large systems. While various BDD and MTBDD packages have been developed in the community, the CUDD package remains the default choice of most of the symbolic (probabilistic) model checkers. In this paper, we provide the first comparative study of the performance of various BDD/MTBDD packages for this purpose. We provide experimental results for several well-known probabilistic benchmarks and study the effect of several optimisations. Our experiments show that no BDD package dominates on a single core, but that parallelisation yields significant speedups.

1 Introduction

Probabilities play a central role in many areas such as distributed systems, sensor networks, and robotics. They are used to break symmetries, e.g., to elect a leader [21], to resolve conflicts in a network, like the exponential backoff in CSMA/CD [20], or to model unreliable components, such as sensors.

Model checking [7] is an important approach to assess the correctness of such systems, by exploring the state space of a model and checking whether a probabilistic property is satisfied. Model checking faces the so-called *state space explosion* problem: a combinatorial blowup of the number of states in the number of system components and variables. For real-world models, it is therefore infeasible to traverse all states explicitly, since they often contain billions of states [10]. To obtain a more compact representation of the state space, Burch *et al.* proposed Binary Decision Diagrams (BDDs) [10, 27]. These have now become a standard technique to tackle large systems, with successful applications in the analysis of many systems.

BDDs are a heuristic method to represent a large set of states or a transition matrix. They are typically small if the state space contains symmetries, for example in a system containing multiple similar modules. Standard BDDs store state spaces (denoted by $S \subseteq \mathbb{B}^N$) and transition relations according to their characteristic function $\mathbb{B}^N \to \mathbb{B}$. To store functions with any codomain, multi-terminal BDDs (MTBDDs) have been proposed [12, 16]. In this way, functions $\mathbb{B}^N \to \mathbb{N}$ and $\mathbb{B}^N \to \mathbb{R}$ can be represented.

In [5], MTBDDs were first applied in probabilistic symbolic model checking to represent the transition probabilities. They play a central role in the leading probabilistic model checker PRISM [23], which exploits the well-known MTBDD package CUDD [33].

Many parts of probabilistic model checking can be carried out using BDD operations only, and these are often the computationally most expensive steps of the process. In particular, the computation of the set of reachable states and the qualitative precomputation step, which finds the set of states satisfying a formula with probability 0 or 1, are computationally heavy, since they operate on the initial, large model.

In fact, MTBDDs may not be most efficient in probabilistic model checking. The reduced state space obtained after the above operations already contains the information required to compute the probability of the formula by means of numerical approaches, such as linear equation systems or linear programming. MTBDDs may be disadvantageous in such cases, as their regularity is destroyed [4, 19]. A major advantage of avoiding MTBDDs is that the probabilistic model checking process could be accelerated by choosing a suitable BDD package.

In this paper, we provide a comparative study of the model checking process using several BDD packages, together with different settings that influence the performance of these packages. In our model checker ISCASMC [17], we have integrated the packages CUDD [33], BuDDy [13], CacBDD [26], JDD [35], Sylvan [36], and BeeDeeDee [25]. As case studies, we use various well-known probabilistic examples from the PRISM website [31]. We observe that there is no clear winner for the single-core BDD packages while computing BDD operations in parallel may improve the runtime considerably, in particular for large models. We observe moreover that native BDD features offered by the package to atomically perform a sequence of BDD operations on average improve both time and memory consumption, but there are cases where such a feature slightly degrades the performance. Our results suggest it is indeed useful to be able to choose among several BDD packages and optimisations when performing probabilistic model checking, since different BDD packages perform very differently on different models.

2 Probabilistic Model Checking

In this section, we first recall the formal definitions of Markov chains and Markov decision processes. To specify probabilistic properties, we then employ the temporal logic PCTL. Moreover, we sketch some techniques of model checking based on (MT)BDDs.

2.1 Markov Decision Processes

A Markov Chain (MC) can be used to describe a fully probabilistic model; a Markov Decision Process (MDP) serves to describe a system containing both probabilistic and nondeterministic choices. Such a system typically arises from the parallel composition of multiple probabilistic models.

Definition 1. *A* Markov decision process *is a tuple* $\mathcal{M} = (S, s_0, AP, L, Act, \mathrm{P})$ *where* S *is a countable, nonempty set of states,* $s_0 \in S$ *is the initial state,* AP *is a set of atomic propositions,* $L\colon S \to 2^{AP}$ *a labelling function,* Act *is a countable, nonempty set of actions, and* $\mathrm{P}\colon S \times Act \times S \to [0,1]$ *is the transition probability function such that for all states* $s \in S$ *and actions* $\alpha \in Act$ *it holds that* $\sum_{s' \in S} \mathrm{P}(s, \alpha, s') \in \{0, 1\}$.

A (discrete-time) Markov chain *can be seen as an MDP where* $|Act| = 1$.

2.2 PCTL Model Checking

Now we introduce the probabilistic logic PCTL, which we use in our case studies.

Definition 2. *A PCTL formula is a state formula ϕ defined by the following grammar, where $a \in AP$ is an atomic proposition, $\bowtie \in \{<, >, \geq, \leq\}$, $p \in [0,1] \cap \mathbb{Q}$, $n \in \mathbb{N}$, and ψ is a path formula.*

$$\phi ::= a \mid \phi \wedge \phi \mid \neg\phi \mid \mathcal{P}_{\bowtie p}[\psi]$$
$$\psi ::= \mathbf{X}\phi \mid \phi \ \mathbf{U} \ \phi \mid \phi \ \mathbf{U}_{\leq n} \ \phi$$

A qualitative *formula is a formula where each p in $\mathcal{P}_{\bowtie p}[\cdot]$ is either 0 or 1.*

We use standard derived operators such as $\phi_1 \vee \phi_2 = \neg(\neg\phi_1 \wedge \neg\phi_2)$, $\mathit{false} = a \wedge \neg a$, $\mathit{true} = \neg\mathit{false}$, $\mathbf{F}\phi = \mathit{true}\ \mathbf{U}\ \phi$, $\mathbf{G}\phi = \neg\mathbf{F}\neg\phi$ and their bounded counterparts.

PCTL model checking is performed by a recursive descent into the formula under consideration. For model checking the PCTL formula $\mathcal{P}_{\bowtie p}[\psi]$ over a MC $\mathcal{M}$, we first need to compute the probability that a path starting from each state s satisfies the path formula ψ, which is denoted $p_s(\psi)$. We divide all states into three disjoint sets: S^{no}, S^{yes}, and $S^{?}$; we call this operation the *precomputation*. The sets S^{no} and S^{yes} contain the states s such that $p_s(\psi)$ is trivially 0 and 1, respectively. The remaining states belong to $S^{?}$. In order to compute $p_s(\psi)$ for $s \in S^{?}$, we reduce it to solving a linear equation system iteratively. The remaining work is straightforward: we just compare the results with $\bowtie p$. Model checking PCTL over MDPs is similar, but it should deal with adversaries (schedulers, policies) and fairness because MDPs are non-deterministic models, see [7].

2.3 BDD-Based Probabilistic Symbolic Model Checking

In this section, we recall BDDs and MTBDDs. Then, we briefly discuss symbolic model checking using BDDs.

The concept of (reduced ordered) binary decision diagrams (BDDs) was proposed by Bryant [9], based on early works in [1, 24]. A BDD represents a Boolean function $f(x_1, x_2, \ldots, x_n)$. It is a rooted directed acyclic graph [2] where terminal nodes (leaves) are labelled with either 0 or 1. Non-terminal nodes contain a variable label x_i and two edges labelled 0 and 1, typically called the low and the high edge. Each non-terminal node represents the Boolean expression $(\neg x_i \wedge f_{x_i=0}) \vee (x_i \wedge f_{x_i=1})$, where the cofactors $f_{x_i=0}$ and $f_{x_i=1}$ are represented by the target nodes of the low and the high edge, respectively. Furthermore, variable labels x_i appear along paths in the BDD according to an ordering. Finding a good variable ordering is critical for the performance of symbolic model checking; good heuristics exist, which are beyond the scope of this paper. Reduction rules that remove redundant and duplicate nodes ensure that BDDs represent Boolean functions canonically.

Some BDD packages use *complement edges* as further edge labels to denote negation of a Boolean function, which results in efficient negation and allows optimisations that are beyond the scope of this paper. To construct and manipulate BDDs efficiently, BDD packages usually have a unique node table to store all the BDD nodes and provide a way to access nodes in constant time. They also have a cache that stores previously

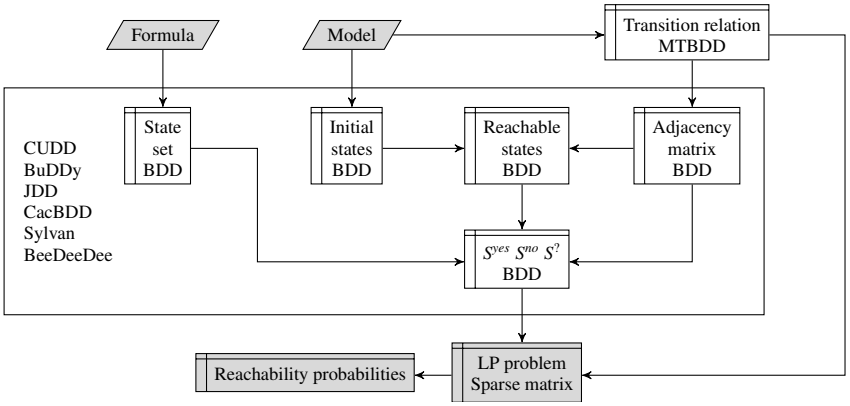

Figure 1. Overview of the data structures used by the model checking algorithm

computed results to avoid duplicate computations. The *initial node size* and *initial cache size* are the initial numbers of entries in the unique node table and the cache, respectively.

MTBDDs. The papers [12, 16] use a variant of BDDs to represent general matrices. It is claimed that MTBDDs are the space-optimal representation of both dense and sparse matrices, and of permutation matrices [16]. Unlike BDDs, the terminal nodes in MTBDDs are not restricted to be 0 or 1. (MT)BDDs enable storing and manipulating very large matrices in a symbolic manner due to their shared structures. Symbolic encoding of MDPs with (MT)BDDs can be then applied directly, see [5].

BDD-based Probabilistic Symbolic Model Checking. Figure 1 gives an overview of the data flow during model checking for PCTL formulas. The data structures with white background are symbolically stored as (MT)BDDs while the data structures with grey background are stored explicitly. Given a PCTL formula $\mathcal{P}_{\bowtie p}[\psi]$ and a description of the model, the common procedure is to first encode the transition relation of the model as MTBDDs and the formula and initial states as BDDs. Moreover, a BDD copy of the transition relation is constructed by abstracting the probabilities between the states. The resulting transition relation is the adjacency matrix of the underlying graph. Then the reachability analysis identifies the reachable states of the model. By employing the *reachE* and *prob1E* iterations proposed in [14], the reachable states are divided into the three sets we mentioned before, namely S^{no}, S^{yes}, and $S^{?}$.

In the next step, one computes the probability of ψ by solving a linear equation system for MCs, or a Linear Programming (LP) problem for MDPs. First, the transition relation MTBDD is transformed to a probability sparse matrix, which is used to encode the corresponding LP problem to compute the probabilities. In practice, approximate value iteration is often used. The resulting reachability probability is then compared with $\bowtie p$ to decide whether $\mathcal{P}_{\bowtie p}[\psi]$ is satisfied. For nested PCTL formulas, the smallest state subformula is handled first, and then the above procedure is recursively applied while traversing through the abstract syntax tree of the formula in a bottom up order.

And-Exist Optimisation. One of the first steps in the PCTL model checking algorithm is the computation of the states that are reachable from the initial states. These are obtained by a fix-point computation of the transition probability matrix for MCs or of the transition probability function for MDPs starting from the set of initial states. For simplicity, consider the MC case and define the function $Post\colon 2^S \to 2^S$ as $Post(X) = \{\, s' \in S \mid \exists s \in X.\mathrm{P}(s,s') > 0 \,\}$. Symbolically, we can represent this operation as $\exists V.R \wedge T$ where R is the BDD encoding the set of states X, T the BDD encoding the adjacency matrix, and V are the variables representing the current states. The resulting BDD encodes the set of next states (using variables $\widehat{V}$); by means of a variable renaming from $\widehat{V}$ to V, we convert it to represent the states in $Post(X)$.

A similar construction underlies the computation of the predecessor states; this operation is the base of the precomputation step that, even for general PCTL properties, may be performed very frequently and it is quite time consuming.

In the above two computations, we first have to build the conjunction of two BDDs encoding the current states and the adjacency matrix and afterwards to remove a number of variables from this conjunction by existential quantification such as $\exists v.bdd_1 \wedge bdd_2$. It is often the case that the BDD representing the conjunction is quite large. Therefore, building $bdd_1 \wedge bdd_2$ and afterwards applying the existential quantification is often very slow. On the other hand, the BDD obtained after the existential quantification is often quite small. To improve this operation, many BDD packages support a so-called And-Exist (relational product) operator, in which these two steps are performed at once. This means that the construction of the intermediate BDD $bdd_1 \wedge bdd_2$ can be avoided, so as to reduce computation time and memory consumption.

2.4 BDD Packages

In this paper, we study the performance of six different BDD packages: CUDD [33], BuDDy [13], CacBDD [26], JDD [35], Sylvan [36], and BeeDeeDee [25]. CUDD is a well-known BDD implementation used in several model checkers. BuDDy has been integrated in several theorem provers and provides many efficient BDD operations. As for CacBDD, experiments in [26] show that it outperforms CUDD in many benchmarks. JDD is a BDD package implemented in Java. Sylvan is a novel parallel decision diagram implementation that parallelises the BDD operations [36]. BeeDeeDee is a recent Java thread-safe implementation of a BDD package. We remark that we are aware of other BDD packages including ABCD [8], PBF [37], Janssen's BDD [22], Carnegie Mellon's BDD [30], BDDNOW [29], and CAL BDD [32]: they are not included in our tool as these packages are outdated and no longer maintained since 2000. Two recently updated packages are BiDDy [28] and MEDDLY [3], however they lack certain basic operations and can therefore not be compared to the other packages.

CUDD is a C implementation of BDDs and MTBDDs developed by Fabio Somenzi, University of Colorado at Boulder. It provides support for operating with ordinary BDDs, Algebraic Decision Diagrams (ADDs), and Zero-suppressed Binary Decision Diagrams (ZDDs). ADDs are a special implementation of MTBDDs that are used, for instance, by PRISM as its MTBDD implementation. The three types of decision diagrams provide essentially the same set of operations; this means, for instance, that an

Table 1. Overview of the features of the BDD packages used.

BDD engine	implementation language	MTBDDs/ ADDs	ZDDs	And-Exist	dynamic var reordering	remarks
CUDD	C	✔	✔	✔(✗)	✔	
BuDDy	C	✗	✗	✔	✔	
CacBDD	C++	✗	✗	✔	✗	dyn. cache mgmt.
JDD	Java	✗	✔	✔	✗	
Sylvan	C	✗	✗	✔	✗	supports multi-core
BeeDeeDee	Java	✗	✗	✗	✗	thread-safe

operation available when operating with BDDs is also available for ADDs. A notable exception is the And-Exist operation that is not yet available for ADDs. Besides its several operations on BDDs, ADDs, and ZDDs, CUDD also supports the conversion of BDDs into ADDs or ZDDs and vice versa. In addition, it also provides a large assortment of variable reordering methods. Though written in C, it provides a C++ interface that provides overloaded operators and that offers to free the decision diagrams that are no longer used by the application.

BuDDy is a BDD package implemented in C by Jørn Lind-Nielsen as a Ph. D. project on model checking finite state machines. It supplies most useful operations for the manipulation of BDDs as well as functions for integer arithmetic like addition and relational operations like And-Exist. It provides also several highly efficient vectorised BDD operations and it supports dynamic variable reordering and garbage collection.

CacBDD is a BDD package written in C++ by Guanfeng Lv. It supports common BDD operations as well as other useful operations like the multiple-operand And-Exist. An interesting aspect of CacBDD is its dynamic cache management algorithm and lazy garbage collection that offer remarkable improvements in the performance of the BDD operations at the expense of free physical memory.

JDD is, unlike the previous BDD packages, a pure Java BDD package developed by Arash Vahidi. JDD supports BDDs as well as ZDDs and it has been originally inspired by BuDDy. Though it is implemented purely by Java, it is still an efficient BDD package and, thanks to its new cache scheme, JDD's memory usage per node is less than BuDDy, which is the major advantage of JDD.

Sylvan is a parallel decision diagram package implemented in C by Tom van Dijk as a Ph. D. project on multi-core decision diagrams. It uses work-stealing and scalable parallel data structures to provide parallelisation of algorithms on decision diagrams. Sylvan currently supports BDDs and list decision diagrams, which are a variation of multi-valued decision diagrams. Among the implemented parallel BDD operations, Sylvan provides other useful operations for model checking such as And-Exist and the Rel-next operation that combines And-Exist with variable renaming. It has been designed as an extensible framework with custom BDD operations in mind and features parallel garbage collection.

BeeDeeDee is a thread-safe BDD package implemented in Java developed by Juliasoft, a spin-off company from the University of Verona. It supports the most common BDD operations. Thread-safety allows for sharing BDD nodes between threads, thus reducing the memory footprint when used in multi-threaded model checking. To improve its performance, BeeDeeDee uses the most modern techniques for multi-threading in Java such as the split locks that are used to control the concurrent garbage collection and the concurrent accesses to the node table.

An overview of the features of the considered BDD packages is given in Table 1.

3 Experimental Results

In order to compare the different BDD packages, we have implemented the BDD-based probabilistic model checking methods in our tool ISCASMC [17].

In ISCASMC, we provide a common high level Java interface to interact with the specific BDD packages, each of them wrapped into a dedicated Java class. This separates the BDD implementations from the model checking algorithms and enables extending the tool with new or updated BDD packages, without having to change other parts of the tool. ISCASMC is mainly implemented in Java and it uses Java Native Access (JNA)[1] to call BDD libraries written in C or C++. JNA is a library decreasing the programmer's effort to call native methods from Java. JNA introduces a small overhead compared to using Java Native Interface (JNI). The overhead turned out to be negligible compared to the total runtimes.

Since we use some features that are not provided natively by all BDD packages, like the And-Exist operation, we have implemented ISCASMC such that it falls back to use ordinary BDD operations when these features are not available. The BDD package to use in the model checking process can be chosen by setting the corresponding ISCASMC command line option. We do not use dynamic variable reordering, since it is not supported by all BDD packages we compare. Also, the reorder algorithm typically has a high performance cost and good static orders were available for the models under consideration.

Experimental Setting and Models. We have performed several experiments on the BDD packages by tuning their settings. All experiments have been performed on a Linux machine with an Intel Core i7-4790 processor at 3.6GHz with 16GB of RAM of which only 8GB are usable by ISCASMC. The time-out for the experiments is 30 minutes.

Table 2 shows the models and the path properties taken from the PRISM website [31] we used for the experiments: the IEEE 1394 FireWire protocol (implementation) [34] ("firewire-impl"), the Google File System model [6] ("google"), the Asynchronous Leader Election protocol [21] ("leader"), the Dining Cryptographers protocol [11] ("dining-crypt"), the Dining Philosophers protocol with no fairness assumption [15] ("phil-nofair"), and the Workstation cluster [18] ("cluster"). Except for the "cluster" and "google" models that are Continuous Time Markov Chains (CTMCs)[2],

[1] https://github.com/twall/jna#readme

[2] As we consider qualitative unbounded properties, they can be checked by transforming the CTMCs to the corresponding embedded Markov chains.

Table 2. Models and properties.

model	path property ψ
firewire-impl	**F**((s1=8 & s2=7) | (s1=7 & s2=8))
leader	**F** ("elected")
dining-crypt	**F** ("done" & parity=func(mod, N, 2))
phil-nofair	**F** ("eat")
cluster	**F** ("premium")
google	**F** ("light_hardware_disaster")

other models are MDPs. The actual formula we checked is $\mathcal{P}_{\geq 1}[\psi]$ for the firewire-impl, leader, cluster, and google models while it is of the form $\mathit{filter}(\mathit{forall}, \mathit{cond} \Rightarrow \mathcal{P}_{\geq 1}[\psi])$ for the remaining dining-crypt (where *cond* is "*hungry*") and phil-nofair (where *cond* is $\mathit{pay} = 0$) models. The keyword *filter* allows us to analyse the property in a given set of states we are interested in: $\mathit{filter}(\mathit{forall}, (\mathit{cond}) \Rightarrow \mathcal{P}_{\geq 1}[\psi])$ is satisfied whenever for all states satisfying *cond* the property $\mathcal{P}_{\geq 1}[\psi]$ holds. Note that the initial state may or may not be considered in the analysis, depending on whether it satisfies *cond*.

Remark 1. All the properties we consider here are qualitative properties, so they can be decided by using the information from the sets S^{yes}, S^{no}, and $S^{?}$ obtained from the precomputation step on BDDs. This allows us to measure the time spent by IscasMC working with only (MT)BDDs, so we get a better understanding of the effects of the different packages and options on the time spent for checking the formulas. We do use MTBDDs to construct the transition relation, as doing so is easier than just using BDDs. The construction never took a significant amount of time.

We further emphasise that our tool can handle quantitative properties. For quantitative properties, various BDD packages will produce the same problem instance but –mostly– in different orders. This will further influence performance of the linear programming problem solvers in a way that is loosely connected to the BDD packages.

Running Time. Table 3 shows the running time in seconds for the six different engines CUDD, BuDDy, CacBDD, JDD, BeeDeeDee, and Sylvan. We repeated each experiment 10 times and report the rounded average time of the 10 runs. We used CUDD as a pure BDD package (cudd-bdd) or as a pure MTBDD package (cudd-mtbdd) while for Sylvan we considered the sequential computation with 1 worker (sylvan-1) or the parallel computation with 7 workers (sylvan-7); we use 7 workers instead of 8 to reserve a processor core for other threads in Java and the operating system, as explained at the end of the section. We kept the default values for the BDD packages except for the initial cache size that we set to 2 612 440 entries and the initial node size to 1 250 000 entries, for packages supporting such options. Moreover, we enabled the And-Exist optimisation for all packages whenever supported natively (see further below). For each model, we considered several instances corresponding to different parameter choices. For example, for the model "firewire-impl", we considered the values 36, 45, 54, and 63 for the parameter "delay". We marked by '–TO–' the cases where the computation took more than 30 minutes and by '–MO–' the computations using more than 8GB of

Table 3. ISCASMC performances with different BDD packages where the values are the rounded average running time of 10 executions.

BDD engine	time (secs)											
	cluster / N				firewire-impl / delay				leader / N			
	1536	1792	2048	2304	36	45	54	63	6	7	8	9
cudd-mtbdd	63	92	125	170	109	65	69	86	11	58	218	709
cudd-bdd	40	54	75	95	**65**	**43**	**42**	**45**	**7**	**25**	**86**	**301**
buddy	**4**	**4**	**6**	**7**	78	46	51	82	**7**	48	165	662
cacbdd	21	28	40	53	87	71	72	85	8	28	95	410
jdd	**4**	**4**	7	**7**	88	49	54	77	8	47	170	637
beedeedee	7	8	12	16	91	55	61	61	11	42	183	853
sylvan-1	5	5	8	8	74	48	50	54	**7**	27	111	615
sylvan-7	5	5	8	8	**29**	**26**	**27**	**28**	**4**	**9**	**28**	**139**
	google / M				phil-nofair / N				dining-crypt / N			
	500	1000	1500	2000	7	8	9	10	25	30	35	40
cudd-mtbdd	8	32	85	144	25	98	339	–MO–	13	44	68	206
cudd-bdd	6	22	56	91	19	74	268	–MO–	9	17	31	51
buddy	**4**	**9**	**23**	**30**	24	122	465	–TO–	18	41	91	169
cacbdd	5	15	42	68	**13**	**51**	**194**	–MO–	**5**	**8**	**13**	**21**
jdd	**4**	10	25	34	37	203	706	–TO–	19	39	92	172
beedeedee	6	22	56	97	32	136	514	–TO–	20	48	101	166
sylvan-1	5	12	32	46	20	120	500	–TO–	6	11	18	56
sylvan-7	6	12	30	35	**5**	**25**	**102**	–MO–	6	**8**	16	47

RAM. We highlighted the best runtimes among all packages in bold font; we marked also the best runtimes excluding sylvan-7 so to consider only sequential computations.

By looking at the results in Table 3, we can immediately see that no BDD package outperforms the others in all case studies. The first thing we note is that Sylvan-7 takes a large advantage from its parallel operations for the high time-consuming models, but the overhead induced by the synchronisation on the parallel operations penalise it on small cases. If we focus on Sylvan-1 and the other sequential BDD packages, we note that for the CTMC models "cluster" and "google" BuDDy and JDD perform better than the other packages, while for the remaining MDP models the best are CUDD as BDD ("firewire-impl" and "leader") and CacBDD ("phil-nofair" and "dining-crypt"). We remark that Sylvan-1 is usually very close to the best-performing BDD packages for "cluster" and "dining-crypt". The CUDD package remains the default choice of most of the symbolic (probabilistic) model checkers, but an order of magnitude in the runtime could be saved sometimes—see the "cluster" and "dining-crypt" examples.

The overall runtime of the packages on all experiments is summarised by the next table, where we sum all entries in Table 3 excluding the failures.

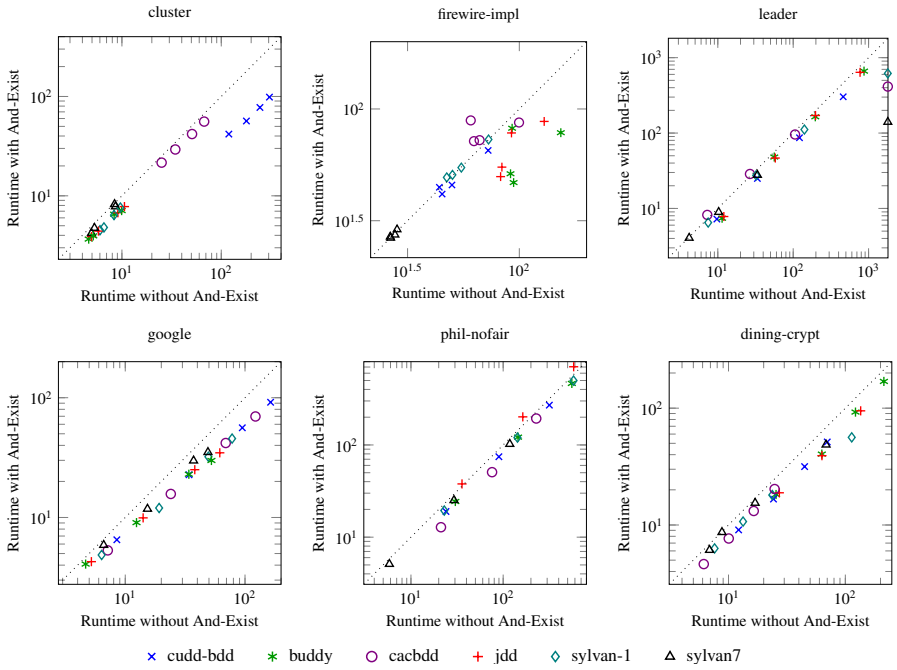

Figure 2. And-Exist Setting comparison (time).

cudd-mtbdd	cudd-bdd	buddy	cacbdd	jdd	beedeedee	sylvan-1	sylvan-7
2837	1522	2156	1433	2493	2598	1838	608

It is worthwhile to note that CUDD (as MTBDD) is always one of the slowest packages and the slowest on the overall set of experiments. This suggests that using only BDD operations sometimes improves the runtime of the model checker for probabilistic systems quite considerably.

And-Exist Optimisation. Figure 2 shows the effect of the use of the And-Exist optimisation in the model checking algorithm. We have performed the experiments by using the BDD packages providing native support to the And-Exist optimisation in the same setting as for Table 3; thus, we omitted CUDD (as MTBDD) and BeeDeeDee since they do not provide such an optimisation.

Each mark in the plot corresponds to the execution of the BDD package with and without the And-Exist optimisation for one instance of the models. The points below (above) the dotted line represent the cases where the usage of And-Exist has reduced (increased) the runtime. As we can see, it is in general convenient to use such an optimisation, but there are cases, like for CacBDD on "firewire-impl" and JDD on "phil-nofair" where it is preferable to not use And-Exist.

Figure 3 is similar to Figure 2, except for the fact that we consider the used memory instead of the runtime. We can note that usually the use of And-Exist helps in reducing

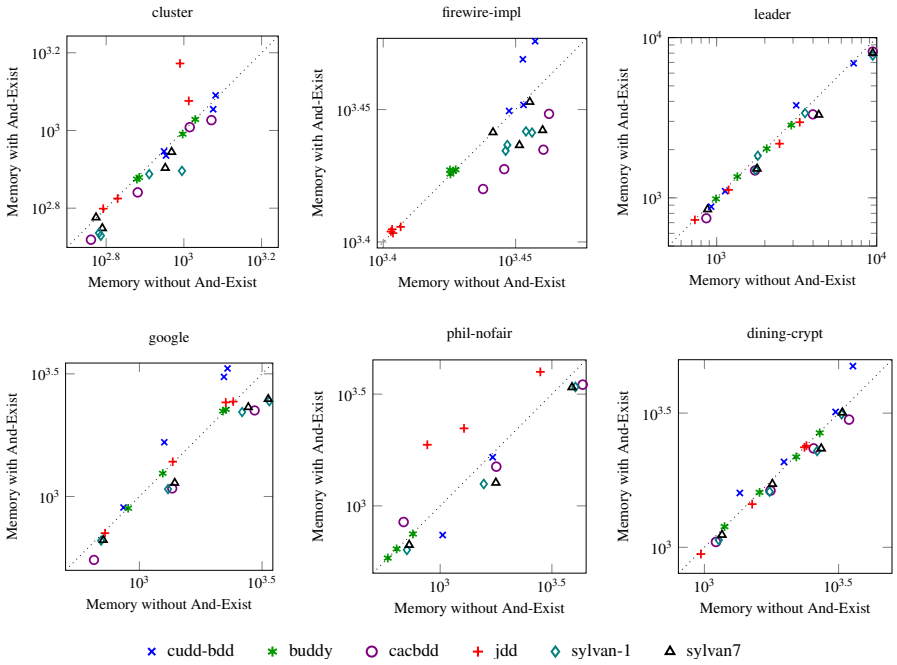

Figure 3. And-Exist Setting comparison (memory, in MB).

the memory footprint but there are cases where CUDD (as BDD) and JDD require more memory when And-Exist is used.

Remark 2. Due to space limitations we omit a detailed report of the memory usage of the BDD packages. We have observed, unless a memory-out is reached, irregular behaviour of memory usage. We think that it is due to the fact that different BDD packages have their own way of memory managing strategies, for instance by preferring to allocate new memory instead of performing a garbage collection.

Impact of the Initial Cache Size. In Figure 4 we plot the outcomes of several experiments on the "firewire-impl" model with BuDDy by varying only the initial cache size. Note that here the reference value $D = 262\,144 = 2^{18}$ is one tenth of the value we used for Table 3. As one can expect, increasing this value usually improves the running time. However, we can first note a big decrease in the running time going from 0.25D to 1D; then, by enlarging the cache size to 4D, the runtime increases for then decreasing again as expected. Note that by making the cache much larger (from 18D) slightly increases the runtime again. The hashing of the elements in the cache and the locality of the cached data may be the causes of the observed behaviour.

In Figure 5 we plot the result of varying the initial cache size and node size. For the cache size, we range from $D = 262\,144 = 2^{18}$ to 20D; for the node size, we range from $0.5 \cdot 10^6$ to $2 \cdot 10^6$. By looking at the plots, we can note that increasing the cache size is

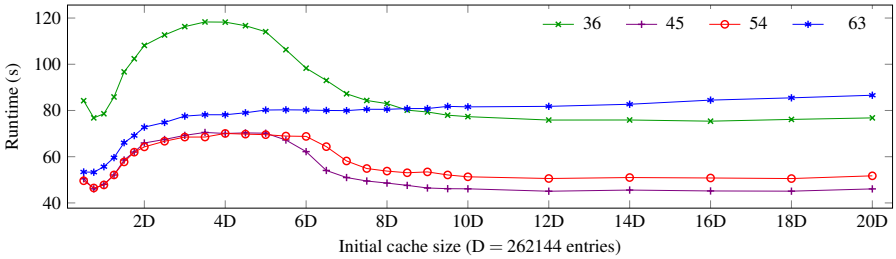

Figure 4. BuDDy performance on "firewire-impl" with different initial cache sizes.

counter-productive if the node size is too small, while it is always worth to increase the node size. Note however that increasing both parameters too much may cause a failure by memory-out.

Effect of the Number of Workers on Sylvan. Figure 6 shows the speedup gained by ISCASMC by using Sylvan with a different number of workers. For this case, we used a different machine equipped with 48 cores (4 AMD Opteron 6168 processors) and 128GB of RAM, but with the same time and memory limits as before.

In general, we observe that the gained speedup is correlated to the runtime size of the models. The "leader" and "phil-nofair" models result in a higher speedup. The highest speedup we obtain is with the "phil-nofair" model: here we obtain a speedup of 26.5 with 47 workers. For the "cluster", "google", "dining-crypt" examples, the speedup is at most 3. For the smallest models, the speedup increases first with the number of workers, but with too many workers the gained speedups are lost again. This behavior could be explained as follows. When reachability consists of only small BDD operations, then there is little opportunity for parallelisation. Meanwhile, all workers are competing to execute the same suboperations, which aggravates the overhead from parallelisation. For the "dining-crypt" example with $N = 25$, with more than 46 workers we even observe a slowdown compared to running with 1 worker.

In all experiments we observe reductions of the speedup when nearly all cores are used. This is likely to be due to the fact that all the 48 workers have to share the cores with the system processes and the Java virtual machine, thus there is an increased scheduling activity affecting the workers. Instead, with at most 47 workers, at least one core remains always available for the system processes and the Java virtual machine, so the scheduling activity is not expected to affect the workers. We can derive that in general it is better to use a number of workers at most equal to the number of cores minus 1; this is why in Table 3 and Figure 2 we use 7 workers instead of 8, as confirmed by similar experiments we performed but omitted for lack of space.

Finally, the result of a speedup of 26.5 with 47 workers is similar to results obtained with LTSmin (a C-only model checker) in [36] and suggests that Java and JNA do not have a significant impact on scalability, at least until 44 workers. The performance drop with 45–48 workers, however, is not seen in [36].

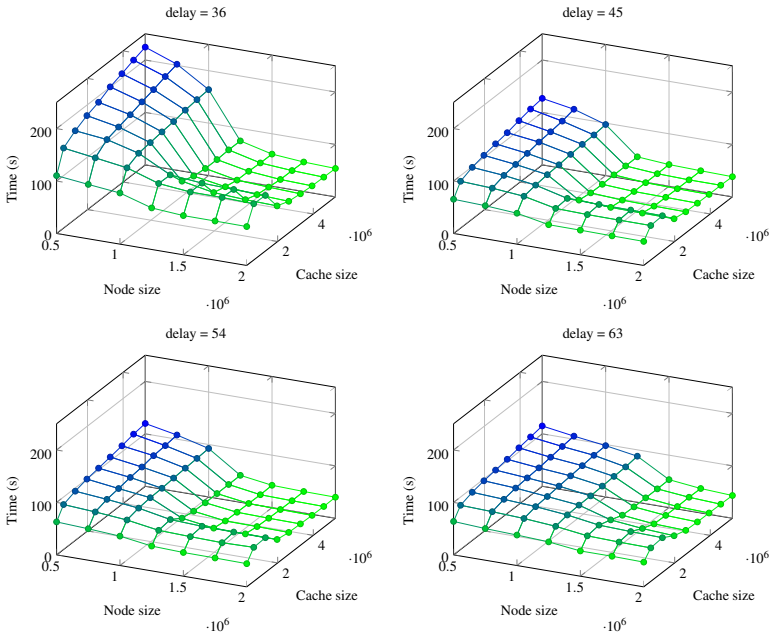

Figure 5. BuDDy performance on "firewire-impl" with different initial cache and node sizes.

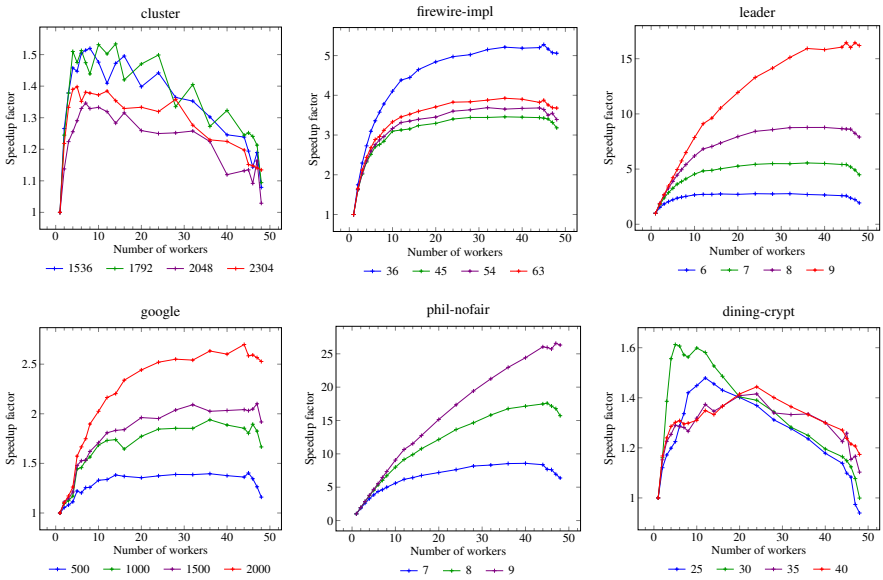

Figure 6. ISCASMC speedup with Sylvan BDD package with different workers.

4 Conclusion

This paper demonstrates the performances of different BDD packages in the context of probabilistic model checking. From the experiments, we have seen that no BDD package is remarkably faster than the others; CUDD (as BDD) and CacBDD performed rather well on MDP models while BuDDy and JDD were more suitable for continuous time MCs. The parallel BDD package Sylvan can outperform the other packages in cases where the overall running time is sufficiently high and multiple cores are used, and is competitive with the other packages when used sequentially, despite the overhead added by parallelisation. This shows that parallelisation of BDD operations is very good for performance and that other BDD packages might also profit from this approach. The experiments confirmed that BDDs are sufficient for probabilistic model checking and much faster than MTBDDs. We have also shown that the And-Exist optimisation speeds up the whole verification process in some case studies, while for others it does not lead to a considerable speedup or even leads to a decreased performance.

Acknowledgments. This work has been supported by the EU FP7 project SENSATION (318490), the STW-ProRail ExploRail project ArRangeer (12238), the NWO projects MaDriD (612.001.101) and BEAT (612.001.303), the National Natural Science Foundation of China (Grants 61472406, 61472473, 61450110461), the Chinese Academy of Sciences Fellowship for International Young Scientists (Grants 2013Y1GB0006, 2015VTC029), the CAS/SAFEA International Partnership Program for Creative Research Teams, and the Sino-German CDZ project CAP (GZ 1023).

References

1. S. B. Akers. Binary decision diagrams. *IEEE Trans. on Computers*, 27:509–516, 1978.
2. H. R. Andersen. An introduction to binary decision diagrams. Course Notes on the WWW, 1997.
3. J. Babar and A. Miner. Meddly: Multi-terminal and edge-valued decision diagram library. In *QEST*, pages 195–196, 2010.
4. R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *ICCAD*, pages 188–191, 1993.
5. C. Baier, E. M. Clarke, V. Hartonas-Garmhausen, M. Kwiatkowska, and M. Ryan. Symbolic model checking for probabilistic processes. In *ICALP*, volume 1256 of *LNCS*, pages 430–440, 1997.
6. C. Baier, E. M. Hahn, B. R. Haverkort, H. Hermanns, and J.-P. Katoen. Model checking for performability. *MSCS*, 23(4):751–795, 2013.
7. C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, Cambridge, 2008.
8. A. Biere. ABCD. `http://fmv.jku.at/abcd/`.
9. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, 100(8):677–691, 1986.
10. J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. *I&C*, 98(2):142–170, June 1992.
11. D. Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *J. of Cryptology*, 1(1):65–75, 1988.

12. E. M. Clarke, M. Fujita, P. C. McGeer, K. McMillan, J. C.-Y. Yang, and X. Zhao. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. In *IWLS*, 1993. http://repository.cmu.edu/compsci/453.
13. H. Cohen, J. Whaley, J. Wildt, and N. Gorogiannis. BuDDy. http://sourceforge.net/p/buddy/.
14. L. de Alfaro, M. Kwiatkowska, G. Norman, D. Parker, and R. Segala. Symbolic model checking of probabilistic processes using MTBDDs and the Kronecker representation. In *TACAS*, volume 1785 of *LNCS*, pages 395–410, 2000.
15. M. Duflot, L. Fribourg, and C. Picaronny. Randomized dining philosophers without fairness assumption. *Distributed Computing*, 17(1):65–76, 2004.
16. M. Fujita, P. C. McGeer, and J. C.-Y. Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *FMSD*, 10(2–3):149–169, 1997.
17. E. M. Hahn, Y. Li, S. Schewe, A. Turrini, and L. Zhang. IscasMC: A web-based probabilistic model checker. In *FM*, volume 8442 of *LNCS*, pages 312–317, 2014.
18. B. R. Haverkort, H. Hermanns, and J.-P. Katoen. On the use of model checking techniques for dependability evaluation. In *SRDS*, pages 228–237, 2000.
19. H. Hermanns, M. Kwiatkowska, G. Norman, D. Parker, and M. Siegle. On the use of MTBDDs for performability analysis and verification of stochastic systems. *J. of Logic and Algebraic Programming*, 56(1–2):23–67, 2003.
20. IEEE 802.3–2002. Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Standard, 2002.
21. A. Itai and M. Rodeh. Symmetry breaking in distributed networks. *I&C*, 88(1):60–87, 1990.
22. G. Janssen. The Eindhoven BDD package. ftp://ftp.ics.ele.tue.nl/pub/users/geert/bdd.tar.gz.
23. M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *CAV*, volume 6806 of *LNCS*, pages 585–591, 2011.
24. C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38(4):985–999, 1959.
25. A. Lovato, D. Macedonio, and F. Spoto. A thread-safe library for binary decision diagrams. In *SEFM*, volume 8702 of *LNCS*, pages 35–49, 2014.
26. G. Lv, K. Su, and Y. Xu. CacBDD: A BDD package with dynamic cache management. In *CAV*, volume 8044 of *LNCS*, pages 229–234, 2013.
27. K. L. McMillan. *Symbolic model checking*. Springer, 1993.
28. R. Meolic. Biddy a multi-platform academic bdd package. *J. of Software*, 7(6), 2012.
29. K. Milvang-Jensen and A. J. Hu. BDDNOW: A parallel BDD package. In *FMCAD*, volume 1522 of *LNCS*, pages 501–507, 1998.
30. Model Checking Group at Carnegie Mellon University. BDD. http://www.cs.cmu.edu/~modelcheck/bdd.html.
31. PRISM web site. http://www.prismmodelchecker.org.
32. R. K. Ranjan and J. Sanghavi. CAL BDD. http://embedded.eecs.berkeley.edu/Research/cal_bdd/.
33. F. Somenzi. CUDD: CU decision diagram package release 2.5.0. http://vlsi.colorado.edu/~fabio/CUDD/.
34. M. Stoelinga and F. W. Vaandrager. Root contention in IEEE 1394. In *ARTS*, volume 1601 of *LNCS*, pages 53–75, 1999.
35. A. Vahidi. JDD, a pure Java BDD and Z-BDD library. http://javaddlib.sourceforge.net/jdd/.
36. T. van Dijk and J. van de Pol. Sylvan: Multi-core decision diagrams. In *TACAS*, volume 9035 of *LNCS*, pages 677–691, 2015.
37. B. Yang, Y. Chen, R. E. Bryant, and D. R. O'Hallaron. Space- and time-efficient BDD construction via working set control. In *ASP-DAC*, pages 423–432, 1998.