Reinforcement Learning Guided Boundary Activity Passing for Incremental Bounded Model Checking

Sutirtha Bhattacharyya
University of Bremen
Bremen, Germany
sutirtha@uni-bremen.de

Chandan Kumar Jha
University of Bremen
Bremen, Germany
chajha@uni-bremen.de

Rolf Drechsler
University of Bremen, DFKI GmbH
Bremen, Germany
drechsler@uni-bremen.de

Abstract

The performance of Incremental *Bounded Model Checking (BMC)* is strongly dependent on the efficiency of the underlying *Boolean Satisfiability (SAT)* solvers. Recent SAT solvers employ different heuristics for non-trivial tasks like branching, clause management, and phase selection. These heuristics have been sophisticated through decades of research, but when used in the context of BMC, they still leave room for improvement due to its incremental nature. One of the most important heuristics which dictates the order of variables in the SAT problem that needs to be assigned, is called Branching. Recently, in the context of incremental BMC, *Boundary Activity Passing (BAP)* has been introduced to improve branching decisions across successive frames. Although BAP can significantly improve BMC performance, it is not effective in every frame. This motivates a selective approach. In this paper, we propose a *Multi-Armed Bandit (MAB)* based *Reinforcement Learning (RL)* approach to selectively apply BAP in each BMC frame. Our mechanism shows up to 1.25× speedup on SAT benchmarks and explores 2.47× additional frames on average for UNDET benchmarks from *Hardware Model Checking Competition (HWMCC)*'25 compared to the *state-of-the-art (SOTA)*.

1 Introduction

Formal Verification (FV) has become the leading approach for hardware design verification. Among the algorithms popularly used for FV, *Bounded Model Checking (BMC)* [7] is a symbolic verification algorithm that verifies temporal properties of systems up to a given bound. In the context of hardware verification, BMC is primarily used for the verification of sequential circuits. This is because sequential circuits require reasoning about the circuit's behavior across multiple clock cycles. This makes verification more challenging due to larger state spaces compared to purely combinational verification. On the other hand, for purely combinational circuits, such temporal unrolling is unnecessary and correctness is mainly established through *Combinational Equivalence Checking (CEC)*. Hence, in this work, we focus on improving BMC for sequential hardware verification.

BMC works by encoding the system's initial states and transition relation, and iteratively unfolding the design over increasing time frames. At each BMC depth (or frame), the verification problem is translated into a *Boolean Satisfiability (SAT)* problem that encodes all possible executions of the system up to that depth and the negated form of the property that needs to be checked. The property is negated so that a satisfying assignment to the generated formula corresponds to a counterexample that violates the property. A SAT solver is then used to solve the generated SAT problem. If the formula is satisfiable, it indicates that the property is violated. Otherwise, one can guarantee that the system is safe up to the depth the engine has explored within the provided time-bound.

Modern *state-of-the-art (SOTA)* SAT solvers [5, 8, 9] are mainly based on *Conflict Driven Clause Learning (CDCL)* [18] which extends the traditional *Davis-Putnam-Logemann-Loveland (DPLL)* [13, 14] procedure using clause learning and non-chronological backtracking. The performance of these solvers depends heavily on the branching heuristics that determine the sequence of variable assignments during search. One of the widely used branching heuristics is *Variable State Independent Decaying Sum (VSIDS)* [19], where each variable is associated with an activity score that is increased for variables participating in a conflict and decayed periodically to prioritize recent conflicts.

The SAT formulas generated at each frame in BMC are incremental. Therefore, as BMC unrolls the design, newer SAT variables appear in the formula that correspond to replicated instances of the same circuit component across successive frames. Hence, these newer variables are related to older variables from previous frames. However, SOTA hardware verification tools such as ABC [11] adopt the default activity initialization scheme for the newer variables. Addressing this limitation, a recent work [20] proposed *Boundary Activity Passing (BAP)*. BAP initializes the activity of the newer variables to the activity of the corresponding older variables related through the design component at each BMC frame. Although BAP can improve BMC performance in many cases, applying it uniformly across all frames can also degrade performance. Thus, a selective approach to applying BAP is required. While the prior work [20] also addresses this issue, the selection mechanism is biased towards not applying BAP and fails to generalize to the latest benchmarks from *Hardware Model Checking Competition (HWMCC)*'25.

In this work, we propose a novel *Multi-Armed Bandit (MAB)*-based *Reinforcement Learning (RL)* approach for selectively applying BAP during incremental BMC. Our approach is integrated into the ABC verification tool and shows significant improvement on benchmarks from the latest HWMCC 2025. To summarize, our contributions in this work are

- A new MAB-based RL approach for selective BAP integrated in the SOTA ABC [11] tool, improving BMC at scale.
- A systematic analysis and proper intuition for the hyperparameters and biases used in the learning method, eliminating the need for hit-and-trial for future researchers.
- Extensive results on systematically selected benchmarks from the latest HWMCC'25 [10].

MLCAD '26, Jeju Island, Republic of Korea

2 Background

This section presents the required background for understanding the proposed methodology. It discusses BMC, VSIDS, BAP, and the MAB framework of RL.

2.1 Bounded Model Checking and Hardware Verification

BMC [6, 7, 12] is a popular FV technique that checks violation of properties within a finite execution bound. It transforms the problem into a SAT problem and is mainly used for efficient bug finding in hardware and software systems. BMC checks if a transition system M violates a property ϕ within k steps. This procedure is modelled with the initial state predicate $I(s_0)$, the transition relation $T(s_i, s_{i+1})$, and the property $\phi(s)$. The system is unrolled up to depth k, and the following formula is constructed:

$$I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg\phi(s_i) \tag{1}$$

This formula is satisfiable if and only if there is a counterexample in the design (as the property is negated).

In hardware verification, the design is usually represented as an *And-Inverter Graph (AIG)*. In this form, the circuit's functionality is represented structurally only using AND, inverters, and sequential elements. Verification tools like ABC [11] represent the AIG as a network of nodes internally before converting it to a SAT problem in the *Conjunctive Normal Form (CNF)* using Tseitin Transformation [22]. The primary objectives when improving a BMC engine are:

- Derivation of faster counterexamples for SAT instances.
- Exploration of greater depth for UNDET instances.

Here, SAT instances correspond to benchmarks for which a counterexample can be found within the timeout limit and UNDET instances are benchmarks for which neither a proof nor a counterexample is found within the time-bound.

2.2 Variable State Independent Decaying Sum

The performance of a SAT solver heavily depends on its branching heuristic. A branching heuristic decides on the sequence of decision variables during search. One of the popular heuristics for branching is VSIDS [19]. VSIDS decides on the branching variable based on an activity value associated with the variable. In a SAT solver like Glucose [3–5], the variables are stored in a max heap data structure (`order_heap`) with a comparator based on the activity values. The activity values of the variables are initialized to 0, and as the solver progresses and conflicts occur, the values are increased accordingly. The activity of a variable increases when it appears in a conflict clause and decreased periodically by a factor between 0 and 1. The decay of the activity scores occurs to prioritize recent conflicts. The activity score is meant to capture the relative importance of a particular variable in the formula, so when the solver needs to branch on a variable, it picks out the unassigned variable with the highest activity value from the order heap.

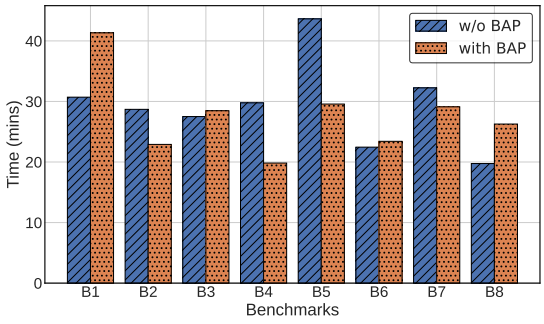

Figure 1: Time to SAT (BAP vs no BAP)

2.3 Boundary Activity Passing

When BMC unrolls the system for the next time frame, copies of design components are created, which are reflected as SAT variables. Since these are *copies* of the same design components, these variables are expected to demonstrate similarity during SAT solving. Despite this structural similarity, tools like ABC initialize these newly introduced variables with the default SAT solver initialization strategy which is typically setting them to 0. This problem was first identified in [20], and the authors proposed a concept called Boundary Activity Passing (BAP).

The correspondence between variables across successive BMC frames can be identified using the underlying circuit object identifiers maintained by ABC. Since variables in the newer frames are created from copies of the same circuit components, ABC is able to map them to their corresponding variables from earlier frames. Using this mapping, BAP initializes the activity of newly introduced variables using the activity values of the corresponding variables from previous frames.

2.4 Multi-Armed Bandit

The MAB [16, 21] is a framework that enables sequential decision making under uncertainty. It is modeled as a scenario where an agent repeatedly selects actions (or *arms*) from a set, receiving a reward for the selection. The objective of the agent is to maximize the cumulative reward over time, while learning the reward distribution of the arms.

The main challenge in MAB is the exploration-exploitation problem. The agent must balance between selecting less frequently sampled arms to learn their distribution better (exploration) and selecting the current best known arm (exploitation). There are several techniques to balance exploration-exploitation, like ϵ-greedy and *Upper Confidence Bound (UCB)* [2]. In this work, we adopt the UCB strategy to balance exploration and exploitation during selective BAP application.

3 Motivation

This section motivates the proposed method by demonstrating the need for selective BAP and analyzing the limitations of SOTA approaches.

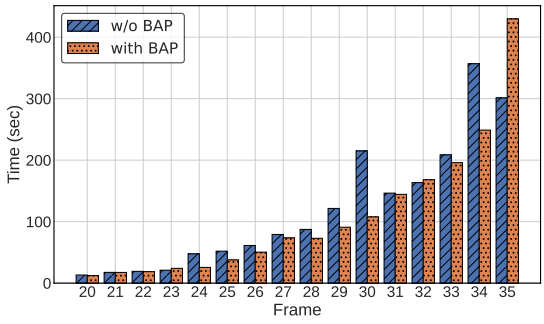

Figure 2: Per-Frame Runtime Behavior

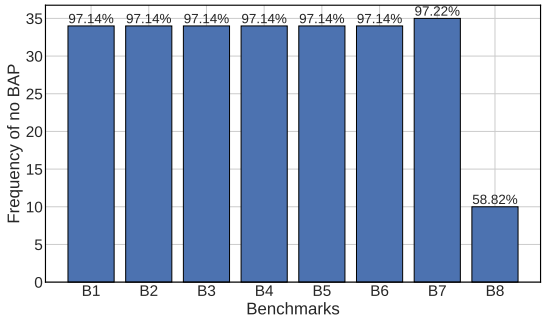

Figure 3: Biased BAP in SOTA

3.1 Need for Selective BAP

As discussed in Section 2.3, BAP initializes the activities of the newer variables based on the corresponding older variables. While this improves performance on some benchmarks, it can also deteriorate performance compared to the baseline. Fig. 1 shows the time taken to find a counter example (time to SAT) for systematically selected benchmarks (described in Section 5) from HWMCC'25 that took more than 1000 seconds to solve in the baseline (w/o BAP) system. The blue hatched bars represent the time taken to find a counterexample without using BAP, and the orange dotted bars represent the time taken when using BAP uniformly at all frames. Among 8 benchmarks, 4 (B2, B4, B5, B7) show improved performance when BAP is used, and for the rest, the performance is worse. These results indicate that the effectiveness of uniformly applying BAP at all frames varies significantly across benchmarks. While BAP improves branching decisions for certain designs, it can also negatively affect the solver's performance. This shows that a static approach that applies BAP at all frames is not optimal.

To further analyze this behavior, we look at the time taken per frame when using BAP and found that the performance varies at each frame depending on the application of BAP. Fig. 2 illustrates the per-frame runtime for benchmark B7 during incremental BMC, comparing executions with and without BAP. The *with BAP* execution times are represented with orange dotted bars, and the *w/o BAP* execution times are marked with blue hatched bars. We see that at frame 34, using BAP is faster, while at frame 35, not using BAP

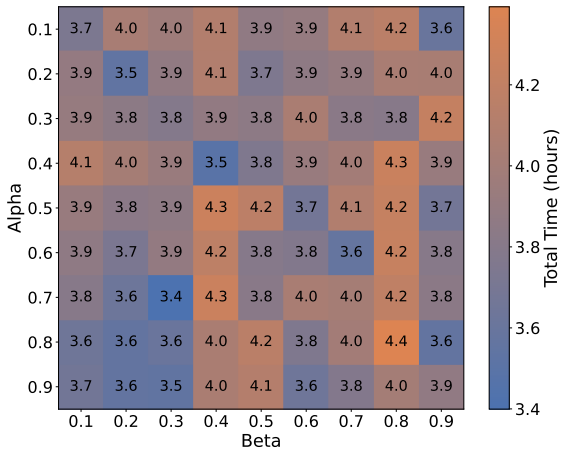

Figure 4: Total Time Taken Across All Benchmarks in Hours Corresponding To Each (α, β) Combination

leads to better performance. This observation further motivates the need for selectively applying BAP at individual frames.

3.2 Limitations of SOTA

The current SOTA research in selective per-frame BAP [20] is based on an MAB-based RL approach. In [20], the available actions for the agent are two:

- applying BAP at the current frame.
- proceeding without BAP.

At each frame, the agent selects one of these two actions, determining whether BAP will be applied. Further analysis of the action selection strategy showed that the methodology is skewed towards **not** performing BAP at a frame. We apply the SOTA methodology for the same set of benchmarks from Fig. 1 and analyze the selection strategy. Fig. 3 shows the percentage of times applying BAP was avoided by the SOTA methodology. For benchmarks B1 to B6, applying BAP was avoided by the selection strategy for 97.14%, which is 34 out of 35 frames. For benchmark B7, BAP is not applied for 97.22% of the time which is 35 out of 36 frames. To determine the cause of this, we also analyzed the algorithm used in [20]. While the algorithm introduces a bias towards not selecting BAP with proper intuition, the hyperparameter that controls the bias is dependent on another hyperparameter with which it has no relation. This analysis further motivated us to propose a stronger selection strategy that applies BAP with better efficiency.

4 Methodology

We propose a MAB-based RL framework to decide whether BAP should be applied at each frame during incremental BMC. Alg. 1 outlines the MAB-based decision algorithm. In the algorithm, both Q and N are two-element arrays representing 2 actions of the agent

- **Action 0** represents the action of not performing BAP.
- **Action 1** represents the action of performing BAP.

The Q array stores the estimated value of each action, and the N array represents the number of times a particular action has been

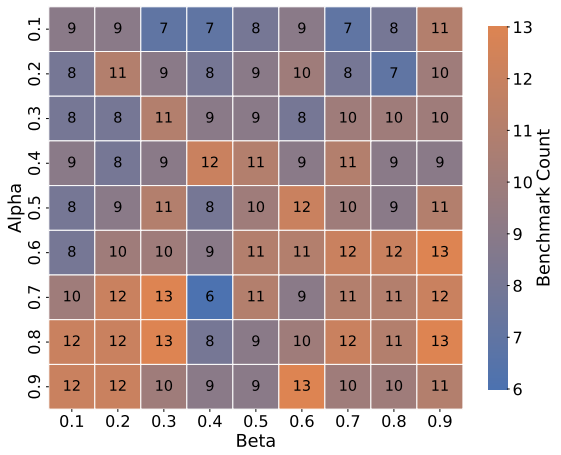

Figure 5: Number of Better Performing Benchmarks (Out of 17) Corresponding to Each (α, β) Combination

Algorithm 1 RL based selective BAP

Data: Q[], N[], timesteps
Input: Action, Average Propagation Rate, α, β
Output: Action
  $reward \leftarrow$ Average Propagation Rate
  if Action == APPLY_BAP **then**
    $Q[0] \leftarrow Q[0] + \frac{(\alpha * reward)}{N[0]}$
  end if
  $N[Action] \leftarrow N[Action] + 1$
  $Q[Action] \leftarrow (\beta * Q[Action]) + \frac{(reward - (\beta * Q[Action]))}{N[Action]}$
  $timesteps \leftarrow timesteps + 1$
  $max\_upper\_bound = -\infty$
  for $i \leftarrow 0$ to 1 **do**
    $upper\_bound \leftarrow Q[i] + \sqrt{\frac{2 * \log(t+1)}{N[i]+1}}$
    if $upper\_bound > max\_upper\_bound$ **then**
      $max\_upper\_bound \leftarrow upper\_bound$
      $Action \leftarrow i$
    end if
  end for
  return $Action$

selected. The reward function is based on *Average Propagation Rate*, motivated by the intuition that efficient branching decisions will result in increased *Boolean Constraint Propagation (BCP)* within the SAT solver, as also used in the SOTA [20]. Alg. 1 extends the standard UCB [2] algorithm with two modifications made for the selective BAP setting.

- **Action 1 Penalty:** This modification penalizes action 1 when it is selected by giving a reward to action 0. The intuition behind this is that aggressively applying action 1 (perform BAP) can negatively affect performance. Thus action 1 should win *convincingly* over action 0 to get selected. This change is highlighted in yellow in algorithm Alg. 1. The reward given to action 0 when action 1 is selected is proportional to the observed reward, adjusted by the hyper-parameter α.
- **Rotting Bandits:** The standard reward update rule for MAB based algorithms is $Q[i] = Q[i] + \frac{reward - Q[i]}{N[i]}$. We modify the standard reward updation mechanism with a recency-aware scheme inspired by *rotting bandits* [17]. The rotting bandit approach decays the reward while updating it to prioritise the influence of relatively recent BMC frames. The decay factor in the algorithm is the hyper-parameter β, and the relevant portion is highlighted in red in Alg. 1.

4.1 Hyperparameter Selection

To determine the ideal values of the hyperparameters α and β, we first define the possible ranges of both the parameters.

- **Beta**: The rotting bandits [17] introduces the **decay** of rewards as a function of time. Since β controls the temporal decay factor, its value must lie between $(0, 1)$. In our experiments we restrict the range of β to $[0.1, 0.9]$ to avoid extremely aggressive or negligible decay behavior.
- **Alpha**: α decides the magnitude of the *bonus* reward given to Action 0 when Action 1 is selected. This is to penalize the selection of Action 1. Assigning a *bonus* reward greater than the observed reward would extremely bias the policy against applying BAP. Thus, α is also restricted to the range $(0, 1)$ and by the same intuition as for β, we further restrict the range to $[0.1, 0.9]$.

After identifying these ranges for the hyperparameters, we consider sample points $\{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}$ for both the parameters to uniformly cover the search space. Since the (α, β) pair results in a search space of only 81 parameter combinations, we perform a systematic search for the best performing parameter configuration.

Figs. 4 and 5 show the results of this exhaustive search. The heatmap Fig. 4 represents the cumulative runtime of 17 systematically selected from HWMCC'25 benchmarks in hours, and the heatmap Fig. 5 shows the number of benchmarks that outperform the SOTA corresponding to the configuration. The top two configurations of (α, β) are $(0.7, 0.3)$ and $(0.4, 0.4)$ with a total time of 3.4 and 3.5 hours respectively, and a better performing benchmark count of 13 and 12, respectively. Based on this analysis, we select the two top-performing configurations $(0.7, 0.3)$ and $(0.4, 0.4)$ for the rest of our experiments.

5 Experimental Evaluation

All the experiments are conducted on a Linux machine equipped with 32 gigabytes of RAM and an Intel Core i7-7700 CPU running at 3.60GHz. We integrate the proposed methodology into the SOTA tool ABC's [11] `bmc3g` engine, which has the Glucose [5] SAT solver integrated. All the experiments, including `bmc3g` considered for this work, also include the *Decision History Reuse* heuristic implemented in [20]. The timeout limit was set to 3600 seconds following the standard HWMCC evaluation setting.

Table 1: Results on HWMCC'25 SAT Benchmarks

BM	Size	bmc3g	with BAP	SOTA	(0.4,0.4)	(0.7,0.3)
SS1	26295	1842.53	2479.6	1849.73	1644.62	1753.74
SS2	4232	1721.86	1374.48	1731.08	1503.95	1526.02
SS3	39468	37.27	41.57	37.28	22.06	29.4
SS4	6255	1650.12	1707.87	1644.81	1391.41	1999.92
SS5	52326	26.81	30.32	26.8	31.66	31.72
SS6	14531	1786.05	1189.74	1790.85	1469.39	1364.53
SS7	14118	26.09	16.64	26.19	10.42	22.13
SS8	52067	2618.14	1774.4	2621.98	1757.99	1939.9
SS9	8275	1347.42	1405.14	1349.8	1505.26	1993.75
SS10	17570	21.08	15.57	20.88	11.38	13.18
SS11	18945	116.17	93.67	116.22	106.15	114.24
SS12	5025	22.07	30.09	22.06	36.27	29.86
SS13	9665	43.61	40.27	43.2	49.24	40.96
SS14	1545	20.28	25.45	20.28	21.8	16.36
SS15	2700	1934.5	1746.58	1929.09	1724.29	1713.93
SS16	1575242	1184.75	1576.22	1352.12	1211.83	1488.92
SS17	287370	46.63	61.34	52.99	52.01	55.37
SS18	1571051	30.61	23.18	32.6	31.48	31.24
SS19	319118	282.04	275.51	263.58	315.41	317.85
SS20	2347769	30.4	43.17	33.36	30.27	30.18
SS21	193978	22.01	15.87	19.28	15.1	17.44
SS22	287252	148.98	160.52	190.74	164.58	149.22
bmc3g HM					1.25	1.11
with BAP HM					1.18	1.04
SOTA HM					1.25	1.11
bmc3g GM					1.41	1.30
BAP GM					1.07	0.99
SOTA GM					1.13	1.05

5.1 Selection of Benchmarks

We first collected all the benchmarks classified as SAT in the publicly available HWMCC'25 [1, 10] results. Then, we evaluate them using the default `bmc3g` engine in ABC with a timeout of 3600 seconds. The benchmarks that report UNDET under the timeout are selected as the UNDET benchmarks in our benchmark suite. The remaining benchmarks are included as SAT benchmarks. Among the SAT benchmarks, the ones that report SAT in under 20 seconds are excluded from the list. We have also excluded the *microban* benchmarks from this list to maintain a manageable workload.

5.2 Evaluation Metric

As discussed in Section 2.1, the primary objective when improving a BMC engine is to speed up *time to SAT* for SAT benchmarks and to explore to a larger depth for UNDET benchmarks. System-level speedup is traditionally evaluated using the *Geometric Mean (GM)*. A recent work [15] discusses the demerits of using GM and instead shows how *Harmonic Mean (HM)* provides a more accurate characterization of speedup. The authors of [15] propose two methods of calculating HM speed up - *Equal Time Speedup (ETS)* and *Equal Work Speedup (EWS)*. In this work, we adopt the EWS variant since it is used when a 'fixed amount of work needs to be completed in the minimum possible time'. In our case, a counterexample is to be found in the minimum possible time. The EWS HM evaluates speed up by calculating the HM of the times reported by two different strategies and then considering the ratio between them. Along with this, we also report the traditional GM speedup for the SAT benchmarks. For UNDET benchmarks, we report the *Arithmetic Mean (AM)* of the depth improvements over the baselines.

Table 2: Results on HWMCC'25 UNDET Benchmarks

BM	Size	bmc3g	with BAP	SOTA	(0.4,0.4)	(0.7,0.3)
SU1	14999	38	36	38	36	38
SU2	40916	38	38	38	37	35
SU3	77972	37	36	37	36	35
SU4	22388	37	36	37	38	36
SU5	400095	34	33	34	34	35
SU6	202592	36	35	36	35	35
SU7	34821	37	38	37	38	38
SU8	35460	33	31	33	32	32
SU9	5227	123	144	121	172	153
SU10	23029	1984	836	1984	1984	1524
SU11	34171	3	3	3	3	3
SU12	628275	15	15	15	15	15
SU13	19352	21	22	21	21	22
SU14	37927	22	22	22	22	22
SU15	2664	28	28	28	28	28
SU16	9799	26	26	26	26	26
SU17	10416	30	30	30	30	30
SU18	591517	39	39	39	39	39
SU19	577700	45	45	45	45	45
bmc3g AM					2.37	-22.89
with BAP AM					62	36.74
SOTA AM					2.47	-22.79

Table 3: Results on SAT Benchmarks from SOTA

BM	Size	bmc3g	with BAP	SOTA	(0.4,0.4)	(0.7,0.3)
IS1	89677	13.45	17.46	13.51	13.52	13.6
IS2	1359703	430.45	279.24	343.47	167.42	169.47
IS3	2347769	30.25	43.03	33.46	30.14	30.23
IS4	13687	1351.27	1470.14	1347.94	1440.04	1571.73
IS5	8275	1349.03	1402.72	1343.09	1509.21	1985.84
IS6	1574869	344.56	450.46	329.29	563.72	369.22
IS7	557954	173.12	173.99	172.39	174.17	173.01
IS8	557905	97.36	97.75	98.37	98.37	97.71
IS9	557904	547.34	551.04	547.69	550.37	546.53
IS10	1092699	1906.56	2538.96	1305.99	2123.7	1401.12
IS11	302832	1217.79	1431.43	1218.05	1350.5	1318.5
IS12	295328	56.8	53.08	54.83	61.81	60.77
bmc3g HM					1.00	1.00
with BAP HM					1.21	1.21
SOTA HM					1.02	1.02
bmc3g GM					0.99	1.04
with BAP GM					1.08	1.13
SOTA GM					0.95	0.99

Table 4: Results on UNDET Benchmarks from SOTA

BM	Size	bmc3g	with BAP	SOTA	(0.4,0.4)	(0.7,0.3)
IU1	7443	6385	9891	7260	11325	6270
IU2	461	8227	7929	8048	8121	8234
IU3	1792	4951	4837	4942	4921	4943
IU4	448	323568	331867	325895	324151	324939
IU5	976	10781	9680	10795	10193	10235
IU6	3457	63235	44315	73004	73936	73110
IU7	3088	75631	75158	75505	75435	75468
IU8	3108	11945	9192	12386	9346	8133
IU9	1931	10130	10157	10161	10168	10136
IU10	39733	103678	103814	103636	103638	103955
bmc3g AM					1270.3	689.2
with BAP AM					2439.4	1858.3
SOTA AM					-39.8	-620.9

5.3 Results on Selected Benchmarks

Tabs. 1 and 2 show the results of our methodology compared to the baselines for the selected benchmarks. We compare against three

baselines: the default ABC flow without BAP (bmc3g), uniform BAP application at every frame (with BAP), and the SOTA selective BAP approach [20] (SOTA). Our methodology is represented by the columns $(0.4, 0.4)$ and $(0.7, 0.3)$, which correspond to the (α, β) values described in Section 4.1. Additionally, the benchmarks and their sizes are shown in the columns BM and Size, respectively. The size here is set as the sum of latches and the number of *AND* gates in the design. The benchmarks are given *identifier (ID)*s to save space, and these IDs are mapped to the actual benchmark names in Appendix A.

Tab. 1 shows the results for the SAT benchmarks from the selected benchmarks suite. A (α, β) value of $(0.4, 0.4)$ achieves a HM speed up of 1.25x, 1.18x, and 1.25x over baselines bmc3g, with BAP, and SOTA, respectively. On the other hand, $(0.7, 0.3)$ achieves 1.11×, 1.04×, and 1.11× harmonic mean speed up over baselines bmc3g, with BAP, and SOTA respectively. While $(0.4, 0.4)$ performs better in terms of EWS HM, $(0.7, 0.3)$ performs better for 13 benchmarks while the former performs better for 12 benchmarks. Note that the improvements over both bmc3g and SOTA are similar, indicating that SOTA performs comparably to the baseline while also showing the efficiency of our approach. We also compare our methodology using the traditional GM speed up. Compared to ABC's bmc3g engine, we achieve up to 1.41× GM speed up, and compared to SOTA, we achieve 1.13× speed up.

Tab. 2 shows the results for the UNDET benchmarks for the selected benchmark suite. The $(0.4, 0.4)$ combination consistently performs better than the $(0.7, 0.3)$ combination over all the baselines. While for the baselines bmc3g and SOTA, the $(0.7, 0.3)$ combination performs poorly, the $(0.4, 0.4)$ shows improved performance, exploring 2.37, 2.47 greater depths on average, respectively. Compared to the *BAP* baseline, the combinations $(0.4, 0.4)$ and $(0.7, 0.7)$ performs better exploring 62 and 36.74 greater depths on an average. We also observed an interesting outlier benchmark from this suite, *pals_opt-floodmax.5_overflow.ufo.UNBOUNDED.pals.aig*, which is omitted from Tab. 2 due to its anomalous behavior. For our methodology and all baselines except uniformly applied BAP, the benchmark reported UNDET after reaching only 9 frames within the 3600-second timeout. In contrast, the uniformly enabled *with BAP* configuration explored up to 513 frames and eventually reported SAT.

5.4 Results on Benchmarks from SOTA

Additionally, along with the systematically selected benchmarks, we also show results on the benchmarks reported in the SOTA work. Tabs. 3 and 4 show the results for the SAT and UNDET benchmarks respectively. On SAT benchmarks, both of the combinations $(0.4, 0.4)$ and $(0.7, 0.7)$ show comparable performance to *bmc3g*, and *with BAP* achieving 1.0× and 1.02× improvement respectively. Compared to *with BAP*, our methodology shows 1.21× improvement. In this context, we also justify the efficiency of our methodology through the characteristics of the benchmarks from SOTA. Fig. 6 shows the bar chart of the time taken for *w/o BAP* (bmc3g) and *with BAP* with the dotted orange bars representing the time for *with BAP* and the hatched blue bars representing the time for *w/o BAP* (bmc3g). We observe that, for most benchmarks from the SOTA benchmark suite, uniformly disabling BAP performs either better than or comparable to uniformly enabling it. Hence,

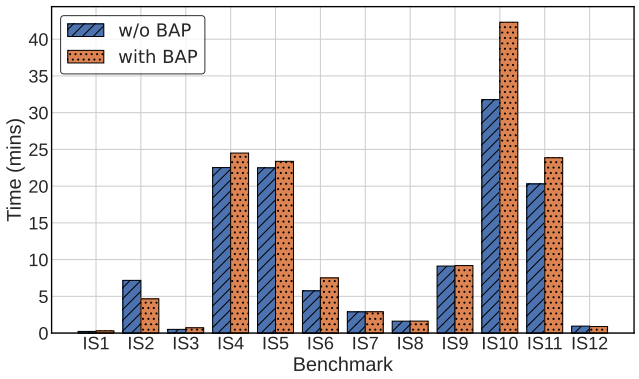

Figure 6: Comparison for Benchmarks from SOTA Paper

the strong bias toward disabling BAP in the SOTA methodology appears favorable for this particular benchmark distribution. From Tab. 3, our methodology achieves performance comparable to both *bmc3g* and the SOTA approach, while outperforming the uniformly enabled *with BAP* baseline by 1.21×. These results suggest that our adaptive strategy effectively balances the trade-off between uniformly enabling and uniformly disabling BAP. Tab. 4 shows the results of our methodology for the UNDET benchmarks selected in [20]. While our methodology achieves a higher average depth compared to *bmc3g* and *with BAP*, we do not achieve better performance compared to the SOTA. We gain 1270.3 and 2439.4 average depth over *bmc3g* and *with BAP*, respectively, using $(0.4, 0.4)$ and 689.2 and 1858.3, respectively, using $(0.7, 0.3)$.

6 Conclusion

In this work, we addressed the problem of selectively applying BAP during incremental BMC. Through detailed analysis, we showed that the effectiveness of applying BAP varies significantly across benchmarks and even across BMC frames making static approaches to apply BAP suboptimal. To address this, we proposed an improved MAB-based RL approach to selectively apply BAP across BMC frames. We also did an detailed and exhaustive study of the hyperparameters that guided the learning strategy used in this work. Experimental results on systematically selected benchmarks from HWMCC'25 show that our methodology improves time-to-SAT and exploration depths compared to existing baselines and prior SOTA approaches for selective BAP application.

7 Acknowledgements

This work was supported by the European Horizon MSCA Doctoral Network REACT "Self-AwaRe NEuromorphic ArChiTectures: Security, Reliability and Energy-Efficiency" under Grant No. 101226463. Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

Funded by the European Union

References

[1] 2025. HWMCC'25 Benchmarks and Results. https://zenodo.org/records/17428464. Zenodo dataset containing benchmarks, certificates, log files, and raw results for HWMCC'25.

[2] Rajeev Agrawal. 1995. Sample Mean Based Index Policies with O(log n) Regret for the Multi-Armed Bandit Problem. *Advances in Applied Probability* 27, 4 (1995), 1054–1078. http://www.jstor.org/stable/1427934

[3] Gilles Audemard and Laurent Simon. 2009. GLUCOSE: a solver that predicts learnt clauses quality. (01 2009).

[4] Gilles Audemard and Laurent Simon. 2012. Refining Restarts Strategies for SAT and UNSAT. In *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming - Volume 7514*. Springer-Verlag, Berlin, Heidelberg, 118–126.

[5] Gilles Audemard and Laurent Simon. 2018. On the Glucose SAT Solver. *International Journal on Artificial Intelligence Tools* 27, 01 (2018), 1840001. arXiv:https://doi.org/10.1142/S0218213018400018 doi:10.1142/S0218213018400018

[6] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. 2018. Bounded Model Checking. (6 2018). doi:10.1184/R1/6603944.v1

[7] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '99)*. Springer-Verlag, Berlin, Heidelberg, 193–207.

[8] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. 2024. CaDiCaL 2.0. In *Computer Aided Verification: 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24–27, 2024, Proceedings, Part I* (Montreal, QC, Canada). Springer-Verlag, Berlin, Heidelberg, 133–152. doi:10.1007/978-3-031-65627-9_7

[9] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. 2024. CaDiCaL, Gimsatul, IsaSAT and Kissat Entering the SAT Competition 2024. In *Proc. of SAT Competition 2024 – Solver, Benchmark and Proof Checker Descriptions (Department of Computer Science Report Series B, Vol. B-2024-1)*, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda (Eds.). University of Helsinki, 8–10.

[10] Armin Biere, Nils Froleyks, and Mathias Preiner. 2025. Hardware Model Checking Competition 2025. In *2025 Formal Methods in Computer-Aided Design (FMCAD)*. 1–1. doi:10.34727/2025/isbn.978-3-85448-084-6_6

[11] Robert Brayton and Alan Mishchenko. 2010. ABC: an academic industrial-strength verification tool. In *Proceedings of the 22nd International Conference on Computer Aided Verification* (Edinburgh, UK) *(CAV'10)*. Springer-Verlag, Berlin, Heidelberg, 24–40. doi:10.1007/978-3-642-14295-6_5

[12] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. 2001. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design* 19 (01 2001), 7–34. doi:10.1023/A:1011276507260

[13] Martin Davis, George Logemann, and Donald Loveland. 1962. A machine program for theorem-proving. *Commun. ACM* 5, 7 (July 1962), 394–397. doi:10.1145/368273.368557

[14] Martin Davis and Hilary Putnam. 1960. A Computing Procedure for Quantification Theory. *J. ACM* 7, 3 (July 1960), 201–215. doi:10.1145/321033.321034

[15] Lieven Eeckhout. 2025. Use Equal-Work or Equal-Time Speedup, Not Geomean Speedup. 276–285. doi:10.1109/ISPASS64960.2025.00033

[16] Tor Lattimore and Csaba Szepesvari. 2019. Bandit Algorithms. (2019). https://tor-lattimore.com/downloads/book/book.pdf

[17] Nir Levine, Koby Crammer, and Shie Mannor. 2017. Rotting Bandits. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/97d98119037c5b8a9663cb21fb8ebf47-Paper.pdf

[18] J.P. Marques Silva and K.A. Sakallah. 1996. GRASP-A new search algorithm for satisfiability. In *Proceedings of International Conference on Computer Aided Design*. 220–227. doi:10.1109/ICCAD.1996.569607

[19] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. 2001. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*. 530–535. doi:10.1145/378239.379017

[20] Satyam Shubham, Sutirtha Bhattacharyya, Ansuman Banerjee, and Raj Kumar Gajavelly. 2025. New Branching Heuristics for Incremental Bounded Model Checking. In *2025 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, Vol. 1. 1–6. doi:10.1109/ISVLSI65124.2025.11130348

[21] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction* (second ed.). The MIT Press. http://incompleteideas.net/book/the-book-2nd.html

[22] G. S. Tseitin. 1983. *On the Complexity of Derivation in Propositional Calculus*. Springer Berlin Heidelberg, Berlin, Heidelberg, 466–483. doi:10.1007/978-3-642-81955-1_28

A Appendix: Benchmark Mapping

The detailed names of the benchmarks with their corresponding ID name used in the paper are listed here.

Table 5: Mapping of Benchmarks from Tab. 1

ID	Benchmark
SS1	arbitrated_top_n2_w64_d32_e0
SS2	arbitrated_top_n2_w8_d32_e0
SS3	arbitrated_top_n3_w128_d16_e0
SS4	arbitrated_top_n3_w8_d32_e0
SS5	arbitrated_top_n4_w128_d16_e0
SS6	arbitrated_top_n4_w16_d32_e0
SS7	arbitrated_top_n4_w32_d16_e0
SS8	arbitrated_top_n4_w64_d32_e0
SS9	arbitrated_top_n4_w8_d32_e0
SS10	arbitrated_top_n5_w32_d16_e0
SS11	circular_pointer_top_w128_d16_e0
SS12	circular_pointer_top_w32_d16_e0
SS13	circular_pointer_top_w64_d16_e0
SS14	circular_pointer_top_w8_d16_e0
SS15	circular_pointer_top_w8_d32_e0
SS16	pals_lcr.7.1.ufo.BOUNDED-14.pals+Problem12_label01
SS17	yosyshq_appnote_123_veer_axi-p62
SS18	pals_lcr.3.ufo.BOUNDED-6.pals+Problem12_label00
SS19	Problem05_label47+token_ring.13.cil-2
SS20	Problem13_label23
SS21	Problem14_label18
SS22	yosyshq_appnote_123_veer_axi-p73

Table 6: Mapping of Benchmarks from Tab. 2

ID	Benchmark
SU1	arbitrated_top_n2_w8_d128_e0
SU2	arbitrated_top_n3_w16_d128_e0
SU3	arbitrated_top_n3_w32_d128_e0
SU4	arbitrated_top_n3_w8_d128_e0
SU5	arbitrated_top_n4_w128_d128_e0
SU6	arbitrated_top_n4_w64_d128_e0
SU7	arbitrated_top_n5_w16_d64_e0
SU8	circular_pointer_top_w128_d32_e0
SU9	level_16
SU10	ponylink-slaveTXlen-sat
SU11	prodbin-ll_unwindbound10
SU12	ridecore
SU13	shift_register_top_w128_d16_e0
SU14	shift_register_top_w128_d32_e0
SU15	shift_register_top_w16_d16_e0
SU16	shift_register_top_w32_d32_e0
SU17	shift_register_top_w8_d128_e0
SU18	yosyshq_appnote_123_cv32e40x-p114
SU19	yosyshq_appnote_123_cv32e40x-p88

Table 7: Mapping of Benchmarks from Tab. 3

ID	Benchmark
IS1	ILA_Piccolo_LHU_problem
IS2	Problem101_label01
IS3	Problem13_label23
IS4	arbitrated_top_n2_w32_d32_e0
IS5	arbitrated_top_n4_w8_d32_e0
IS6	pals_lcr.5.1.ufo.UNBOUNDED.pals+Problem12_label04
IS7	rocket_1761
IS8	rocket_2045
IS9	rocket_5541
IS10	yosyshq_appnote_123_cv32e40x-p104
IS11	yosyshq_appnote_123_veer-p08
IS12	yosyshq_appnote_123_veer_axi-p62

Table 8: Mapping of Benchmarks from Tab. 4

ID	Benchmark
IU1	Problem02_label10
IU2	bin-suffix-5
IU3	counter_bit_width_large
IU4	fib_30
IU5	kalman_bit_width_small
IU6	qspiflash_dualflexpress_divthree-p132
IU7	qspiflash_qflexpress_divfive-p025
IU8	qspiflash_qflexpress_divfive-p131
IU9	simple_vardep_1
IU10	stack-p2