Early Deep Bug Discovery via Re-run Acceleration and Parallel Multi-Depth Bounded Model Checking (BMC) Exploration

Jungwoo Seo[1] (jwoo.seo@samsung.com), Dongyoung Kim[1] (dongzer0.kim@samsung.com),
Sungjin Park[2] (sungjinpark@siemens.com), Mark Eslinger[3] (mark.eslinger@siemens.com),
Juyeon Son[1] (jy1020.son@samsung.com), Gyehyun Na[1] (gh.na@samsung.com),
Dongeun Lee[1] (dongeun1.lee@samsung.com)
[1] Samsung Electronics Co. Ltd., Korea, [2] Siemens EDA, Seoul, Korea, [3] Siemens EDA, Santa Clara, USA

***Abstract*- Formal verification of complex and long-sequence designs inevitably encounters inconclusive results when proofs cannot be completed within bounded resources. Traditional approaches require engineers to apply abstractions, reduce models, or rely on manual multi-run and deeper target techniques to push formal engines deeper, which is both time-consuming and resource intensive. This paper presents a methodology that combines Re-run Acceleration and dynamic resource reallocation to the Bounded Model Checking (BMC) engine. Re-run Acceleration leverages previously stored verification results, replaying them for properties whose Cone of Influence (COI) has not changed, thereby eliminating redundant computation and accelerating subsequent runs. The saved processor budget is then redirected toward BMC's parallel multi-depth exploration. When neither proven nor uncoverable results are produced within a predefined time window, the methodology reallocates 100% of available processor resources to the BMC engines. This enables the solvers to aggressively explore deeper state space in parallel, significantly increasing the likelihood of detecting deep-seated bugs earlier in the verification timeline. In our memory design, this approach resulted in the detection of approximately 4 additional results within the same overall runtime compared to conventional flows.**

I. Introduction

Formal verification has become an essential component of modern semiconductor design flows, offering exhaustive state-space exploration and stronger confidence in functional correctness compared to simulation. However, as design complexity and sequential depth continue to increase, inconclusive results are inevitable. When proofs cannot be completed within allotted time with available computational resources, engineers must often resort to manual interventions such as abstraction tuning, decomposition, or targeted bug hunting, all of which demand significant engineering effort and slow down bug discovery [1].

In practice, inconclusive results signify that the formal engines have reached their practical depth limits, implying that deeper corner-case bugs, if present in the design, may remain undetected. Merely extending runtime or increasing engine count is inefficient. A smarter allocation of available resources is required.

To address these challenges, we propose a resource-adaptive formal verification methodology that combines Re-run Acceleration with Dynamic BMC Resource Reallocation. Re-run Acceleration eliminates redundant computations by replaying stable results from a previous run, while dynamic BMC allocation enables the tool to shift computational focus toward deeper bounded model checking once proof progress stagnates. Together, these mechanisms reduce inconclusive results and accelerate the discovery of deep-seated bugs within the same runtime budget, providing both higher verification efficiency and earlier design insight.

II. Background

A. *Re-run Acceleration*

Re-run Acceleration enhances formal analysis efficiency by using design knowledge and results from previous runs. Results from previous runs including proven, failed, covered, and uncoverable properties are accelerated in the current run. During each verification cycle, formal results and design knowledge are stored in a database. When subsequent formal analysis is performed on the same design and property set, properties with the above results are typically solved very quickly. The speedup typically goes from days to minutes and hours to seconds for properties that would normally have longer runtimes. This mechanism eliminates redundant proofs, shortens turnaround time, and frees computational capacity [2].

Since the results of the proven properties are obtained quickly, they can be utilized as helper assertions earlier in the process. This enables Re-run Acceleration to direct computational effort toward more difficult properties from an earlier stage of verification.

Historically, formal tools would record which engine solved a given property and then, in subsequent runs, only use those engines relevant to solving the given set of properties. While this would give better results than an initial run, it would still require the engine to do a complete analysis, which could still be a long run time to prove or fail a property. The Re-run Acceleration technique has been around for close to a decade at this point and expands on this concept. During a formal run the engines analyze the design and as part of their analysis find various design information and invariants. This data initially may take a long time to find but can usually be verified quickly. This data for each solved property comprises a results certificate. On subsequent runs, the data is verified and then quickly produces the previous result. The ability of Re-run Acceleration to successfully replay results depends on design and property consistency. Re-run Acceleration works best when the design/properties are the same, hence was mainly used in regressions. The more a design and properties change the greater chance it can affect replay eligibility. Therefore, Re-run Acceleration provides the greatest benefit in an iterative environment involving minor design or verification environment changes.

A practical application of Re-run Acceleration arises after a bug has been found and fixed. When verification is re-executed on the updated database, most properties other than the one that produced the Counter Example (CEX) can typically be replayed without re-solving. This allows the verification effort to focus primarily on the specific property that triggered the bug, while all unaffected results are instantly restored through Re-run Acceleration replay. As a result, post-fix regression can be completed much faster while maintaining full consistency with previous verification states.

B. Parallel Multi-depth Exploration Bounded Model Checking (BMC)

Bounded Model Checking (BMC) explores a design's state space up to a finite depth through SAT/SMT based translation. Modern formal engines enhance this process with parallel multi-depth exploration, enabling simultaneous analysis at different depths (e.g., N, N+1, N+2) across multiple solver instances [3]. This strategy accelerates the discovery of sequential bugs that might otherwise appear only in very deep cycles. And that's why many vendors have been offering BMC engines equipped with various strategies.

Parallel multi-depth exploration is dynamically activated when sufficient computational HW resources is available beyond the number of active property targets. In such conditions, the engine allocates remaining processors to deeper states exploration, allowing BMC solvers to traverse extended transition paths in parallel. This mechanism is primarily applied to enhance CEX detection, effectively exposing deep-state bugs that are often unreachable in conventional static configurations.

C. Dynamic Resource Allocation

Dynamic resource allocation provides the adaptive layer that links proof-oriented and bug hunting engines. The proof-oriented engine is an engine specialized in finding proven and uncoverable. The bug hunting engine, on the contrary, cannot find proven, but is specialized in finding deep CEX or covered. As verification progresses, a monitoring mechanism tracks the rate of proven or uncoverable updates. When progress stagnates within a configurable time window, all available processors are dynamically reassigned to BMC engines. This automatic reallocation effectively switches the system from proof mode to bug-hunting mode, enabling deeper state exploration without manual intervention.

In practice, this mechanism operates through an auxiliary control process that runs in parallel with formal verification. The process continuously monitors runtime status and automatically applies user-defined configuration changes when predefined conditions are met, such as the absence of new proven or uncoverable updates. This external coordination allows dynamic engine reallocation to be performed seamlessly during execution, without interrupting the ongoing verification flow.

This approach ensures efficient use of computational resources. Early phases focus on proof convergence and helper generation, while later phases prioritize deep exploration. This result is an intelligent and adaptive verification flow that optimizes both regression efficiency and bug detection capability.

D. Counterexample (CEX)

In formal verification, a counterexample (CEX) is fully specified execution trace generated by the verification engine that formally proves the violation of a property. The CEX consists of a concrete sequence of input stimuli, internal signal valuations, and state transitions that drive the design from a legal initial state to state where the property is falsified.

Beyond indicating a failure, a CEX provides direct insight into the root cause of the violation by capturing the exact conditions under which the failure occurs. This allows engineers to replay and analyze the failing scenario in detail. If the environmental assumptions used during verification are valid, the CEX exposes a real design bug that must be fixed. Otherwise, the CEX reveals an environmental modeling issue, such as missing or overly permissive constraints, which can lead to false failure.

III. MOTIVATION

In formal verification, inconclusive results are inevitable as design complexity and sequential depth grows. When proofs cannot be completed within the time and resources available, engineers are forced to decide whether to continue pushing for proofs or to switch into bug-hunting mode. In practice, this transition is often manual and reactive – occurring only after significant time has already been spent on unproductive proof analysis.

Such inconclusive states typically indicate that the formal engines have reached their depth limits, while deeper corner-case bugs may remain hidden. Extending runtime alone rarely resolves these cases efficiently; instead, a more targeted use of computational resources is required.

Therefore, our motivation is to establish a verification methodology that can dynamically recognize stagnation and reallocate available resources from proof-oriented engines to deeper BMC exploration. By doing so, the verification process can pivot from attempting to prove properties that are unlikely to converge toward aggressively searching for deep bugs earlier within the same runtime budget.

The proposed approach transforms the traditional "prove-until-timeout" flow into a resource-adaptive process that balances proof convergence and bug discovery depth.

IV. RESOURCE ADAPTIVE FORMAL VERIFICATION FLOW

Scaling challenges in formal verification arise not only from redundant regression but also from insufficient depth exploration. The proposed flow integrates two complementary techniques: Re-Run Acceleration and Dynamic BMC Resource Reallocation to achieve both efficiency and aggressive bug hunting.

A. Re-run Acceleration

We applied Re-run Acceleration to the general formal database (DB) setup process. In the initial verification environment (initial DB), when a CEX occurred for a deadlock assertion, debugging revealed that several control-point constraints were incorrect, which we subsequently corrected. To verify the modified behavior, we created Minor Fix DB1 that included more than 100 new cover properties. In the Minor Fix DB1, the deadlock assertion was proven within 9 hours.

However, Minor Fix DB1 still contained two inconclusive cover properties. To resolve these, we created Minor Fix DB2 by modifying several cover properties and adding an additional assumption. All results were obtained within 7 hours 40 minutes, but further review showed that one assumption of a specific option was over-constrained. Therefore, we removed that assumption and created Minor Fix DB3. The full verification of DB3, without any reuse, completed in 8 hours 16 minutes and produced all results including the deadlock proven. The results for each DB version are shown as reference in Figure 1 below.

By applying Re-run Acceleration across the entire process from the Initial DB through Minor Fix DB3, we measured the performance of each iteration and compared it with the baseline. As shown in Figure 1, the results illustrate the performance differences when Re-run Acceleration is applied versus when each DB version is verified independently. In the first version (DB1), few properties were replayed, and the performance improvement was minimal. In DB2, a higher number of properties were successfully replayed, resulting in noticeable runtime reduction. Finally, in DB3, the improvement became significant, as all properties were solved within only 20 minutes compared to over 8 hours without Re-run Acceleration.

Overall, Re-run Acceleration demonstrated progressively greater efficiency and accuracy as minor updates accumulate, and more verification data is stored in the database.

B. Dynamic Resource Reallocation for enabling parallel multi-depth BMC engines

We evaluated how bug-finding performance changes when applying the BMC engine alone to deadlock assertions. In a verification environment containing 93 assertions and cover properties, a CEX result was obtained in 157 hours under a typical mixed engine configuration. To measure the effect of applying BMC exclusively, we re-ran the same case with a single deadlock checker, allocating 100% of the resources to BMC. Contrary to expectations, no CEX result was detected even after more than 200 hours of runtime. Even in other scenario tests, verifying the existing Inconclusive result with a single property did not reveal any additional improvements. This is shown in Case I and Case II in Table I.

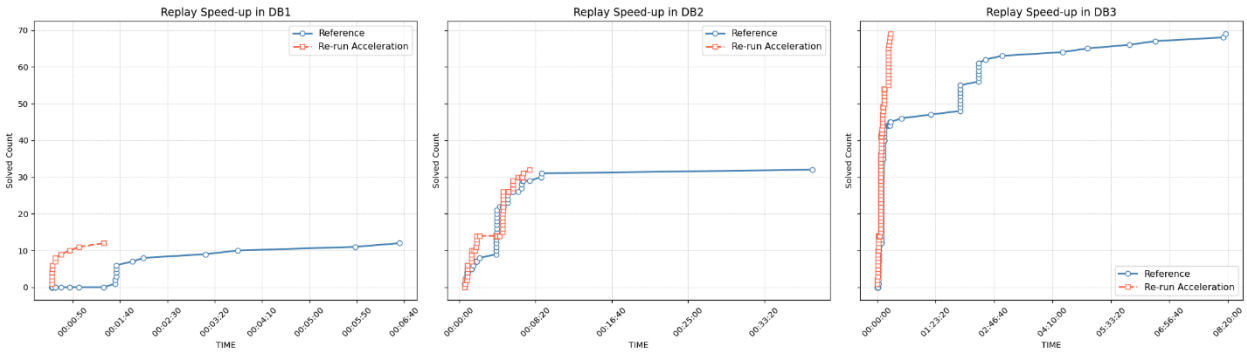

Figure 1. Re-run Acceleration speed-up for replayed properties as DB is updated

Repeated experiments confirmed this behavior. When all properties were solved together, the deadlock bug was consistently discovered, but when only a single target was verified, the result became inconclusive. The same phenomenon was observed in other designs where CEX occurred at deeper cycles. That's because some assertions change to assumptions after they are proven and reduce the COI for primary assertion. That's called Assume-Guarantee [4]. These findings suggest that dedicating time and resources to first obtain proven or uncoverable results, rather than relying solely on BMC analysis from the beginning, allows the BMC engines to achieve deeper and more effective exploration later.

To identify the optimal time to spend on proof analysis, we analyzed one year of formal verification data from previous Flash Memory projects to understand when proven or uncoverable results typically saturate. In most cases, over 97% of proven, vacuously proven, and uncoverable results were obtained within the first 48 hours. In a large environment with more than 500 properties, results continued to appear intermittently at intervals ranging from 1-2 hours up to 10-20 hours, and some proven results still emerged even after 200 hours.

Based on this observation, we evaluated no-progress trigger conditions to determine when to switch resources toward BMC. After running for the initial 48 hours, we tested additional runs with no-progress conditions set to 12 and 24 hours. The 24-hour setting achieved more than 99.5% coverage of proven, vacuously proven, and uncoverable results providing an efficient balance between proof completion and deep-state exploration.

To automate this process, a TCL control script was implemented to monitor verification progress and dynamically reconfigure engines when the defined no-progress condition was met, enabling adaptive allocation between proof-oriented and BMC engines.

```
  onerror {StopVerify 1}

  set cnt 0

  while {cnt is 0} {
      if VerifyDone {
          exit
      }
      if ElapsedTime >= 48 hours {
          increase cnt to 1
      } else {
          wait 10 mins
      }
}

  while {cnt is 1} {
      if VerifyDone {
          exit
      }
      if Num of Proven and Uncoverable is not changed {
          if ElapsedTime without update >= 24 hours {
              Delete All Engines
              Add BMC Engine
              increase cnt to 2
          } else {
              wait 10 mins
          }
```

```
      } else {
          Update Num of Proven and Uncoverable counts
          wait 10 mins
      }
}
```

Figure 2. Pseudo Code of Dynamic Resource Allocation

By applying Dynamic Resource Allocation for BMC engines to Case I and Case II, we observed improved results as shown in Table I below. During the first 72 hours we primarily focused on proven, uncoverable, and vacuously proven results. By using an assume-guarantee approach, tool reduced the COI of the remaining assertions and then focusing on BMC, we observed performance improvements. Case I shows that the inconclusive property detected a CEX at 102 hours. And this flow also shortened the CEX detection TAT in Case II.

TABLE I
PERFORMANCE BETWEEN ALL ENGINES AND DYNAMIC RESOURCE ALLOCATION FOR BMC ENGINES

	Reference w/ all Props		Reference w/ 1 target		Dynamic Resource Allocation for BMC engines + no progress 24h	
	Result	Turnaround time (TAT)	Result	Turnaround time (TAT)	Result	Turnaround time (TAT)
Case I	Inconclusive	200:00:00	Inconclusive	200:00:00	CEX	102:46:07
Case II	CEX	157:52:37	Inconclusive	200:00:00	CEX	87:16:59

Both Case I and II resulted as inconclusive when tested with only a single target. This is because they cannot utilize other proven or covered targets. This testing shows that bug detection is sometimes dependent on proven and covered results.

The second test for utilizing Dynamic Resource Allocation for BMC engines was conducted in an environment where no CEX was observed. We used typical mixed engines and 32 cores for 400 hours, but Reference showed a convergence rate of 67.7%, as shown in Table II. After applying this methodology, we were able to identify 10 additional CEX results. However, two proven items that were found under the general engine configuration could no longer be reproduced. This indicates a trade-off between proven and CEX outcomes.

When applying Dynamic Resource Allocation for BMC, it is important to determine the primary verification objectives. This includes deciding whether to prioritize convergence on proven properties or to focus on discovering CEX results deeper in the state space. Selecting the appropriate target based on the verification goal enables more effective use of computational resources and maximizes the benefits of the methodology.

TABLE II
TRADE-OFF BETWEEN PROVEN AND CEX PERFORMANCE

	Reference	Dynamic Resource Allocation for BMC engines + noprogress 24h
Inconclusive	79	71
Possibly vacuously proven	1	1
Covered	21	21
Uncoverable	1	1
Proven	140	138
CEX	0	10
Vacuously proven	6	6
Convergence	67.7%	71.0%

In the proposed flow, the absence of both proven and uncoverable results within a configurable time window triggers dynamic reassignment of all available processors to BMC engines instances. This enables the BMC engines to utilize their full parallel capacity, allowing deep bugs to be exposed much earlier than with conventional fixed-allocation strategies.

C. Resource-Adaptive Verification Flow (Combined Methodology)

As illustrated in Figure 3 below, the proposed methodology integrates the Re-run Acceleration phase with the Dynamic Resource Reallocation for BMC engines phase to achieve both efficiency and depth in formal verification. Each phase has distinct strengths and limitations. It accelerates regression by eliminating redundant work and maximizing reuse across interactive runs. However, even though properties with

unchanged COI are quickly resolved, this does not guarantee that the remaining complex targets will converge faster or that deep bugs will be exposed within limited resources.

In contrast, the BMC engines provide powerful deep-state exploration, but their efficiency depends heavily on timing and resource allocation. Enabling Dynamic Resource Reallocation too late prolongs turnaround time for bug detection, whereas activating it too early may reduce the number of proven results that can later serve as helper assertions.

To balance this behavior, the Resource-Adaptive Verification flow leverages Re-run Acceleration to quickly establish a stable baseline of proven and uncoverable results. As shown in the upper (blue) region of Figure 3, it replays prior results, reduces active targets, and optimizes resource usage to focus on unsolved properties. Once progress is stabilized in this phase, the available capacity is redirected to the dynamic resource allocation for BMC engines stage, shown in orange. At this point, 100% of the CPU resources are reassigned to the BMC engines to enable parallel multi-depth exploration. The number of 32 cores and 400 hours of time limit are not changed. However, we can improve the results by using the same resources more efficiently.

This sequential transition allows early convergence for shallow properties while aggressively pursuing deeper exploration in later stages. Through this process, Re-run Acceleration prepares the environment for efficient bug hunting by ensuring that solver resources are fully utilized. When progress stalls, this phase activates its full parallel capability to uncover deep-seated bugs faster than conventional fixed-allocation flows.

Overall, the unified flow transforms the conventional trade-off between breadth and depth into a complementary sequence: breadth is achieved early through reuse and rapid convergence, and depth is achieved later through parallel multi-depth exploration.

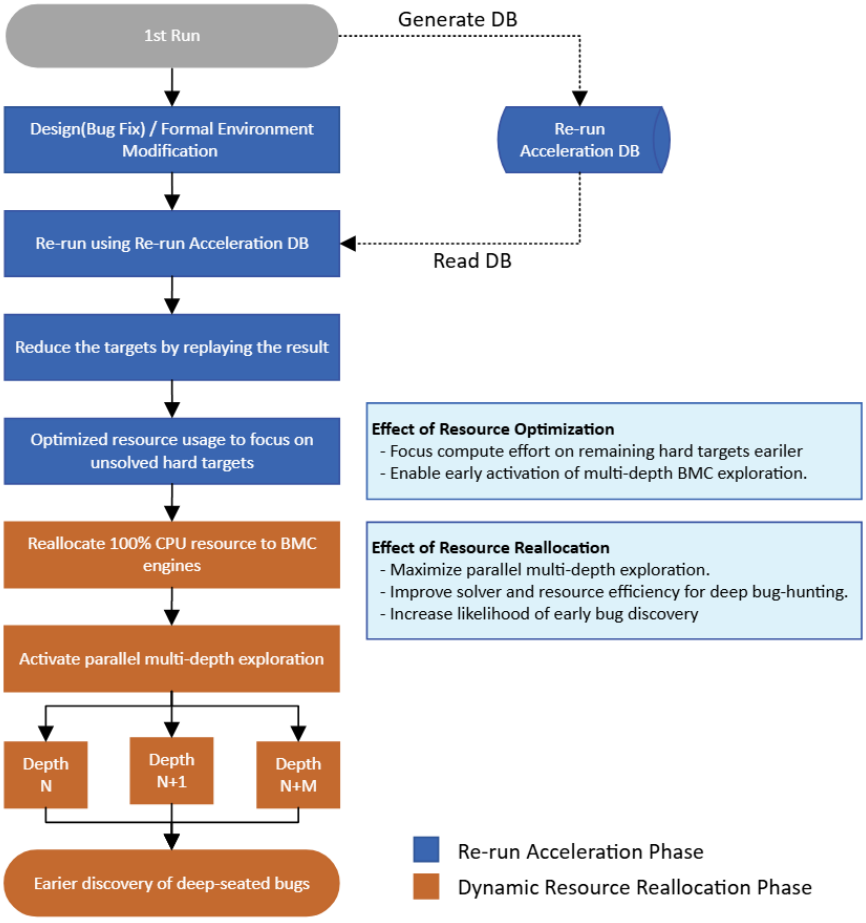

Figure 3. Flow Diagram of Resource Adaptive Verification Flow

V. Experimental Results

A. *Case Study 1: Reset Sweep*

Reset Sweep verifies that when a reset signal occurs during the read/program/erase sequence, all FSMs transition to the IDLE state within a specific number of cycles. If we verify the reset sweep for all sequences at once, the number of cases considered becomes too large, increasing the probability of the assertion being inconclusive. Therefore, in our Flash Memory verification environment, we split the three scenarios—read, program, and erase—and apply appropriate constraints to each to create a targeted verification environment. For example, in the read scenario, the verification limits the CMDs that can occur in the read sequence using assume statements.

If no bugs are found in each sequence, all constraints are then combined, and verification for all CMD combinations is performed. At this stage, since the All CMD verification already includes the operations of read/program/erase that were verified earlier, some portions of the verification are unnecessarily duplicated. Therefore, by recording a Re-run Acceleration save for individual scenarios and then loading those results during the All CMD verification, performance can be improved.

In Table III, we used 32 Cores and typical mixed engines for 400 hours in Reference, but it left 139 Inconclusive. When applying our method to the All CMD verification, out of a total of 534 active targets (assertions and covers), 5 proven, 15 covered, and 254 vacuity check pass (covered) were replayed from the read/program/erase Re-run Acceleration database. Then Dynamic Resource Allocation for maximizing multi-depth exploration was enabled after 72 hours, one additional CEX result was found at 95 hours. However, the overall performance did not show a significant advantage compared to the reference. This is because, when it is enabled, the number of remaining inconclusive cases is too large. 32 cores are not enough to gain effectiveness of parallel multi-depth exploration compared to 139 inconclusive.

TABLE III
RE-RUN ACCELERATION AND DYNAMIC BMC FOR RESET ALL CMD

	Reference	Resource Adaptive Verification Flow
Inconclusive	139	135
Covered	17	17
Uncoverable	1	1
Proven	368	371
CEX	3	4
Possibly vacuously proven	1	1
Convergence	73.7%	74.5%

B. *Case Study 2: Bug Fix and Re-run over DB release*

We also tested the deadlock erase scenario for another product. This scenario verifies that no deadlock occurs during the erase command sequence by ensuring all related finite state machines (FSM) complete their operations and transition correctly under erase conditions.

In the reference, an environmental bug was discovered in the first DB1 within 20 minutes, and when re-verifying the fixed DB2, a real bug was found after 316 hours. Applying the Resource Adaptive Verification methodology to this case, we replayed two proven and two vacuity check pass (covered) results from DB1. After Dynamic Resource Allocation was enabled 72 hours later to maximize multi-depth exploration, the real deadlock bug was discovered after 195 hours. As shown in Figure 4, Resource Adaptive verification reduced the time to uncover the deep bug by 120 hours.

We then applied the test to DB3, which incorporated the fix for the bug found in DB2. When we verified this DB3 using a conventional formal approach for 400 hours, we obtained results for 32 out of the 38 active targets, while the Deadlock Checker remained inconclusive. At that time, the average radius of the remaining properties was 1206.7, where radius refers to the depth in the design's state space that must be explored to prove or disprove a property. We also loaded the Re-run Acceleration database from DB2 and ran it on DB3, but unfortunately no properties were replayed. After the 400 hours has elapsed, the verification again yielded the same 32 property results as the reference, and the Deadlock Checker was still inconclusive. The average radius of remaining properties at this point was 1225.3.

These observations indicate that when updating from DB2 to DB3, extensive internal design and constraint changes altered the COI of all properties, resulting in no properties being eligible for replay via Re-run Acceleration. Ultimately, only the effect of Dynamic Resource Allocation for maximizing multi-depth exploration was applied in DB3.

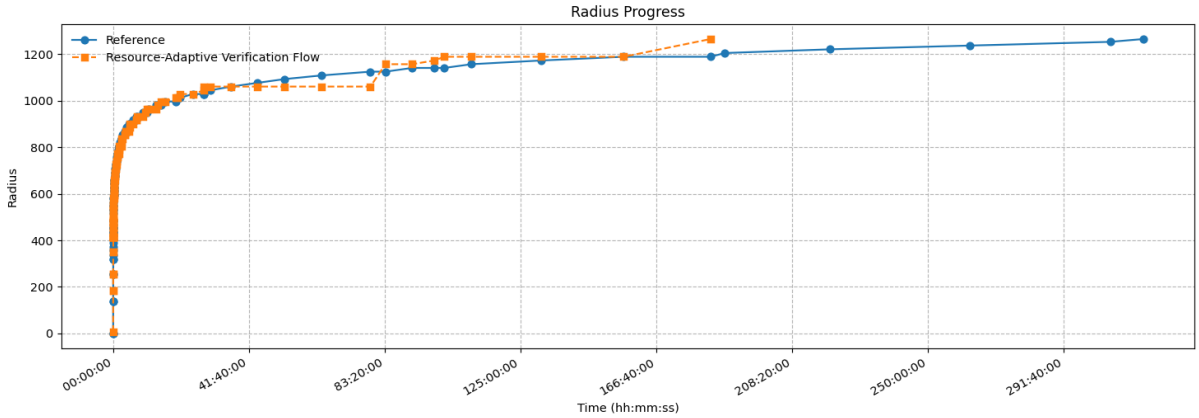

Figure 4. Radius progress while finding deep bug (Reference vs Resource Adaptive Verification Flow)

VI. CONCLUSION

A formal verification flow that combines Re-run Acceleration with Dynamic Resource Reallocation for parallel multi-depth exploration has been introduced. By replaying stable results and leveraging the BMC engine's parallel multi-depth exploration, the proposed methodology reduces inconclusive results and exposes deep bugs significantly earlier.

In this paper, we demonstrated that the proposed Resource Adaptive Verification flow is particularly effective when applied to the initial verification environment setup process, where minor design or constraint changes occur frequently. It is also effective when a fix is applied after a bug is discovered and the process is re-executed. On our memory design, the proposed methodology uncovered three proven and one CEX results within the same runtime, or found the same CEX results faster, showing that resource-optimized formal verification achieves both faster regression and deeper bug discovery.

On the other hand, in situations where the design and constraints change significantly, we observed that the number of properties that can be replayed through Re-run Acceleration decreases sharply. Therefore, future work will focus on developing methods to maintain performance even when structural elements such as module hierarchy or instance names.

REFERENCES

[1] Handling Inconclusive Assertions in Formal Verification, Siemens Verification Academy.
[2] Questa Verify Property Rerun Acceleration, Siemens Support Center.
[3] Questa Verify Property User Guide – Run Multi-Process Analysis
[4] Henzinger, T.A., Qadeer, S., Rajamani, S.K.: "You assume, we guarantee: Methodology and case studies" In: Hu, A.J., Vardi, M.Y. (eds.) Computer Aided Verification. pp. 440{451. Springer Berlin Heidelberg, Berlin, Heidelberg (1998)