12-22-2020

Accelerating Parallel Verification via Complementary Property Partitioning and Strategy Exploration

Rohit Dureja
Iowa State University, dureja@iastate.edu

Jason Baumgartner
IBM Corporation

Robert Kanzelman
IBM Corporation

Mark Williams
IBM Corporation

Kristin Y. Rozier
Iowa State University, kyrozier@iastate.edu

Accelerating Parallel Verification via Complementary Property Partitioning and Strategy Exploration

Abstract

Industrial hardware verification tasks often require checking a large number of properties within a testbench. Verification tools often utilize parallelism in their solving orchestration to improve scalability, either in portfolio mode where different solver strategies run concurrently, or in partitioning mode where disjoint property subsets are verified independently. While most tools focus solely upon reducing end-to-end walltime, reducing overall CPU-time is a comparably-important goal influencing power consumption, competition for available machines, and IT costs. Portfolio approaches often degrade into highly-redundant work across processes, where similar strategies address properties in nearly-identical order. Partitioning should take property affinity into account, atomically verifying high-affinity properties to minimize redundant work of applying identical strategies on individual properties with nearly-identical logic cones. In this paper, we improve multi-property parallel verification with respect to both wall- and CPU-time. We extend affinity-based partitioning to guarantee complete utilization of available processes, with provable partition quality. We propose methods to minimize redundant computation, and dynamically optimize work distribution. We deploy our techniques in a sequential redundancy removal framework, using localization to solve non-inductive properties. Our techniques offer a median 2.4× speedup yielding 18.1% more property solves, as demonstrated by extensive experiments.

Keywords

Power demand, Scalability, Redundancy, Tools, Parallel processing, Task analysis, Portfolios

Disciplines

Computer Sciences | Systems Engineering and Multidisciplinary Design Optimization

Comments

This proceeding is published as Dureja, Rohit, Jason Baumgartner, Robert Kanzelman, Mark Williams, and Kristin Y. Rozier. "Accelerating Parallel Verification via Complementary Property Partitioning and Strategy Exploration." In *Proceedings of the 20th International Conference on Formal Methods in Computer Aided Design (*Strichman, O. and A. Ivrii, eds.) (2020): 16-25. DOI: 10.34727/2020/isbn.978-3-85448-042-6_8. Posted with permission.

Formal Methods in Computer-Aided Design 2020

Accelerating Parallel Verification via Complementary Property Partitioning and Strategy Exploration

Rohit Dureja*, Jason Baumgartner†, Robert Kanzelman†, Mark Williams† and Kristin Y. Rozier*
*Iowa State University, †IBM Corporation

***Abstract*—Industrial hardware verification tasks often require checking a large number of properties within a testbench. Verification tools often utilize parallelism in their solving orchestration to improve scalability, either in *portfolio* mode where different solver strategies run concurrently, or in *partitioning* mode where disjoint property subsets are verified independently. While most tools focus solely upon reducing end-to-end wall-time, reducing overall CPU-time is a comparably-important goal influencing power consumption, competition for available machines, and IT costs. Portfolio approaches often degrade into highly-redundant work across processes, where similar strategies address properties in nearly-identical order. Partitioning should take *property affinity* into account, atomically verifying high-affinity properties to minimize redundant work of applying identical strategies on individual properties with nearly-identical logic cones. In this paper, we improve multi-property parallel verification with respect to both wall- and CPU-time. We extend affinity-based partitioning to guarantee *complete* utilization of available processes, with provable *partition quality*. We propose methods to minimize redundant computation, and dynamically optimize work distribution. We deploy our techniques in a sequential redundancy removal framework, using *localization* to solve non-inductive properties. Our techniques offer a median 2.4× speedup yielding 18.1% more property solves, as demonstrated by extensive experiments.**

I. Introduction

Practical hardware and software verification often mandates checking a large number of properties on a given design. For example, *functional verification* involves checking a suite of low-level assertions and higher-level encompassing properties. *Equivalence checking* compares pairwise equality of each output across two designs, yielding a distinct property per output. *Redundancy removal* requires proving many gate-equalities throughout a design, each comprising a distinct property. Redundancy removal is the core procedure of equivalence checking, and is widely-used to boost verification scalability.

Each property has a distinct minimal *cone of influence* (COI), or fan-in logic of the signals referenced in the property. Verification of a group of properties requires resources proportional to the collective COI size, which is often exponential (after lighter logic reductions). Each property adds distinct logic to the group's collective COI; *affinity* refers to the degree of common vs. distinct logic in the COI. *Atomic verification*[1] of a group of low-affinity properties is thus often significantly slower than solving them one-at-a-time. Conversely, atomic verification of a high-affinity group saves considerable verification resource, as the effort expended for one property can benefit the others without significantly slowing them down [1, 2]. Parallel verification resource can be optimized to leverage these facts using affinity-based *property partitioning* [3], where each parallel process, or *worker*, runs the same strategy on a different property group.

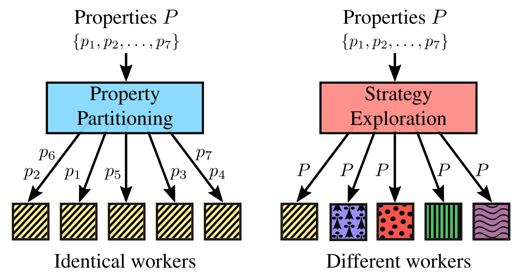


Fig. 1. Parallel verification: property partitioning vs. strategy exploration.

An alternate way to accelerate verification is by using a parallel portfolio (*strategy exploration*), where the same property group is concurrently verified using a different strategy per worker, as depicted in Fig. 1. However, portfolio approaches often degrade into highly-redundant work across processes, where similar algorithms address properties in nearly-identical order. Existing tools often independently use these modes in different contexts, particularly strategy exploration first running qualitatively-different strategies in available workers (e.g., BMC, IC3, interpolation) then padding differently-configured identical strategies in the remaining processes (e.g., IC3 with different heuristics). The latter yields increasingly-redundant CPU-time for diminishing gains in wall-time. These modes need not be mutually-exclusive: a strategy could partition within a worker, and partitioning could use different strategies for different groups. We explore the mutual optimization between property partitioning and strategy exploration, addressing the following challenges:

Property partitioning →

P1 Some workers are not utilized if the number of high-affinity groups is less than available workers.

P2 Some workers finish their tasks and idle (no more partitions to dispatch) while others degrade wall-time solving large or difficult groups, or run on slower machines.

Strategy exploration →

P3 Nearly-identical strategies verify the same properties concurrently yielding redundant computation; two or more workers would solve the same property at nearly the same time.

[1] *Atomic verification* refers to running a set of single-process verification engines (called a *strategy*) on a group of properties. *Serial verification* refers to beginning one atomic task after another finishes, using a single process. *Concurrent* or *parallel verification* refers to dispatching multiple atomic tasks on concurrently-running parallel processes.

https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_8

P4 A worker gets stuck on the first difficult property inhibiting progress; easy properties go unexplored.
P5 When using a round-robin resource-constrained approach to avoid **P4**, a worker fails to solve a difficult property in the allocated time even after several repetitions.

Contributions: We optimize parallel verification using complementary *property partitioning* and *strategy exploration*, in terms of both wall- and CPU-time. **(1)** We present a scalable property partitioning algorithm (Sect. III-A), extending [3] to guarantee *complete* utilization of available processes with provable partition quality. **(2)** We propose parallel scheduling improvements (Sect. III-B), such as resource-constrained irredundant group iteration, incremental repetition, and group decomposition to dynamically cope with more-difficult groups or slower workers. **(3)** We address irredundant strategy exploration of a localization portfolio in a sequential redundancy removal framework (Sect. IV), which we have found to be the most-scalable strategy to prove non-inductive redundancies. **(4)** We additionally propose improvements to *semantic group partitioning* within localization (Sect. IV-C). To our knowledge, this is the first published approach to mutually-optimize property partitioning and strategy exploration within a multi-property localization portfolio.

A. Related Work

Despite the prevalence of parallel verification tools and multi-property testbenches, little research has addressed mutual optimization of parallel partitioning and strategy exploration. Furthermore, most approaches optimize wall-time alone without considering CPU-time, treating additional CPUs as free horsepower to fill with slightly-modified strategies without attempting to minimize redundant computation.

Methods to group properties based on COI similarity are either computationally-prohibitive [1, 2, 4], or do not optimally utilize available parallel processes [3]. They may generate fewer groups than processes, or lose affinity guarantees when requiring *number of groups* as an algorithmic parameter.

Much prior work addresses ways to parallelize specific algorithms in a *single-property* context [5]–[7]. Other work incrementally reuses information between properties to accelerate specific algorithms [8]–[11]. These are complementary to our work, and can be used as strategies therein.

Much complementary research has addressed sequential redundancy removal, using scalability-boosting strategies including induction [12]–[14], simulation [15, 16], and synergistic transformation and verification algorithms [16, 17]. The benefit of parallelizing inductively-provable redundancies has been noted in [18, 19], though little work addresses parallelizing non-inductive redundancies. Localization is a powerful scalability boost to redundancy removal [14, 16, 20] and property checking [21]–[24]. Prior work is focused mostly upon single-property single-process contexts [21]–[24], or solely upon parallel property partitioning [3]. This work is complementary to ours: we extend state-of-the-art solutions for both, to mutually-optimized parallel verification.

II. Preliminaries

The design under verification is represented as a *netlist* N, which is a tuple $\langle\langle V, E\rangle, F\rangle$ where $\langle V, E\rangle$ is a directed graph with vertices V representing *gates*, and edges $E \subseteq V \times V$ representing interconnections between gates. Function $F : V \rightarrow \mathit{types}$ assigns vertices to gate types: constants, primary inputs, combinational logic such as *AND* gates, and sequential logic such as *registers*. A *state* is a valuation to the registers. Certain gates are labeled as *properties*. The *fan-in* (*fan-out*) of gate u is the set of gates which may be reached by traversing edges backward (forward) from u. The fan-in of property p is called the *cone of influence* (COI) of p. Registers and inputs in the COI are called *support variables*. The number of support variables in the COI is its *size*. A *strongly connected component* (SCC) is a set of interconnected gates such that there is a non-empty directed path between every pair of gates in the same SCC. A *merge* of gate u onto gate v consists of moving the output edges of u onto v, then eliminating u from the netlist by treating u as a rename for v.

A. Affinity Analysis

Property grouping algorithms represent support variable information as a *Boolean bitvector* per property [25]. Every support variable in the netlist is indexed to a unique position in the bitvector, set to "1" if and only if the support variable is in the COI of the property. The length of such a bitvector is equal to the total number of support variables in the netlist, and all bitvectors have the same length. The COI size of the property is the number of bits set to "1". These bitvectors may be compared to determine relative property *affinity*. Properties p_1, p_2 with bitvectors bv_1, bv_2 respectively have

$$0 \leq \text{affinity}(p_1, p_2) = 1 - \frac{\text{hamming}(bv_1, bv_2)}{\text{length}(bv_1)} \leq 1.0$$

where hamming(bv_1, bv_2) is the Hamming distance between bv_1 and bv_2, and length(bv_1) is the number of support variables in the netlist [3]. The *distance* between p_1, p_2 equals the Hamming distance between their bitvectors, i.e., dist$(p_1, p_2) =$ hamming(bv_1, bv_2). A *group* g is a set of properties, with a single property g^* therein representing its *center*. The *quality* $Q(g)$ of a group is the minimum affinity between any property in g vs. its center g^*:

$$Q(g) = \min(\{\text{affinity}(p, g^*) \mid \forall p \in g\})$$

It is desirable that property partitioning algorithms guarantee group quality to be greater than a specifiable threshold.

B. High-Affinity Property Grouping

Three-leveled grouping [3] (Fig. 2) utilizes support bitvectors of properties to generate high-affinity groups. The algorithm takes the desired grouping level (l) and affinity threshold (t). It groups properties based upon: a) *Level-1*: identical bitvectors (identical support variables); b) *Level-2*: common large SCCs (containing t% netlist support variables) in the COI; and c) *Level-3*: small Hamming distance between support bitvectors, scalably identified by equivalence-classing *mapped*

```
structural_grouping (Properties P, Netlist N, Level l, Affinity t)
Groups G = P  # each property in singleton group
if l ≥ 1 : grouping_level_1 (G, N)  # identical COI
if l ≥ 2 : grouping_level_2 (G, N, t)  # large SCCs in COI
if l ≥ 3 : grouping_level_3 (G, N, t)  # Hamming distance
return G  # return high-affinity groups
```

Fig. 2. Algorithm to group properties based on structural affinity [3].

bitvectors using threshold-aware mapping functions. Higher levels yield progressively fewer but larger groups.

Straightforward grouping approaches such as pairwise comparison are computationally prohibitive [25], requiring at least quadratic resource with respect to number of properties. Despite being conceptually a quadratic-resource algorithm, bitvector equivalence-classing [3] consumes near-linear runtime and memory in practice, enabling scalable online partitioning with provable quality bounds [3]. Bitvectors are computed during a linear sweep of the netlist, and have size proportional to the number of SCCs plus non-SCC support variables. SCC computation has linear runtime [26]. With efficient implementation, this entire process consumes a few seconds on netlists with millions of support variables and properties: e.g. computing bitvectors in topological netlist order, and garbage-collecting bitvectors as soon as all fanout references have been processed [25].

A priori knowledge of solvers may dictate the ideal grouping level. For example, BDD-based reachability is highly sensitive to COI size, and thus may prefer level=1. BMC may prefer level=3 with lower affinity. Localization may prefer level=1, =2, or =3 depending on subsequent solvers. In many contexts, the caller can set level=3 and allow Fig. 2 to determine group count and size, especially when using the techniques of Sect. III-B and Sect. IV-C to decompose difficult groups.

Theorem 1 ([3]). *Level-1 grouping generates property groups G such that $\forall g \in G : Q(g) = 1.0$.*

Theorem 2 ([3]). *Given affinity t, level-2 grouping generates property groups G such that $\forall g \in G : Q(g) \geq t$.*

Theorem 3 ([3]). *Given affinity t, level-3 grouping generates property groups G such that $\forall g \in G : Q(g) \geq 3 * t - 2$.*

Note that desired number of property groups is not an algorithmic parameter; affinity analysis determines the optimal number of groups respecting configurable quality bounds. For more details on leveled grouping, we refer the reader to [3].

III. Grouping for Parallel Verification

Many organizations have large clusters of computers for load-balancing of tasks such as verification. The maximum number of available workers for a given task (n) is often known, e.g. the maximum number of organizational job submissions allowed per user, minus how many that user wishes to reserve for other tasks. Existing scalable grouping algorithms [3] may generate fewer high-affinity groups than n (**P1**). While partitioning a high-affinity group may yield redundant CPU-time (similar effort expended on nearly-identical COIs), it may benefit wall-time due to disparate difficulty of properties therein: e.g. one may be inductive, and another require deep sequential analysis. Traditional clustering algorithms can be configured to produce $\geq n$ groups, though are computationally prohibitive for online use and may not yield affinity guarantees if n does not align with the given netlist.

```
structural_grouping_parallel (Properties P, Netlist N, Level l,
                              Affinity t, Workers n)
Level l_c = 0  # current grouping level
Groups G = singletons(P)  # initialize to singleton groups
if |G| ≤ n : return G  # fewer properties than workers
if l ≥ 1 : grouping_level_1 (G, N), l_c = 1  # identical COI
if l ≥ 2 and |G| ≥ n :  # else fewer groups than workers
    grouping_level_2 (G, N, t), l_c = 2  # large SCCs in COI
if l ≥ 3 and |G| ≥ n :  # else fewer groups than workers
    grouping_level_3 (G, N, t), l_c = 3  # Hamming distance
if |G| < n :  # fewer groups than available workers
    rebalance (G, N, l_c, t, n)  # distribute groups, see Fig. 4
assert ( |G| ≥ n)  # guaranteed to hold
return G  # return high-affinity groups
```

Fig. 3. Property grouping guaranteed to generate at least $\min(n, |P|)$ high-affinity groups for n parallel workers.

```
rebalance (Groups G, Netlist N, Level l_c, Affinity t, Workers n)
if l_c == 1 :  # divide large level-1 groups in half
    halve_groups (G, n)  # see Fig. 5
else  # rollback minimal-quality level-2 & level-3 groups
    rollback_groups (G, N, l_c, t, n)  # see Fig. 6
```

Fig. 4. Algorithm to subdivide high-affinity groups for n workers.

A. Property Grouping Algorithm

Fig. 3 shows our extension to leveled grouping [3] (Fig. 2), guaranteeing generation of at least $\min(n, |P|)$ provable-affinity groups. Each property is returned as a singleton if there are fewer than n properties. Otherwise, grouping is performed in three levels that iteratively generate fewer, larger groups. Later levels are skipped if the number of generated groups becomes less than n at any level. The algorithm then rebalances as needed by fine-grained affinity analysis: subdividing large or lower-affinity groups to generate at least $\min(n, |P|)$ property groups. As discussed in Sect. III-B, this procedure is beneficial even after initial partitioning to subdivide a difficult group into provably high-affinity subgroups.

The rebalancing algorithm is shown in Fig. 4. It subdivides groups based on the grouping level l_c that generated fewer groups than n. For level-1, quality is already 100% so division is based on number of properties in the group (Fig. 5). Groups with the most properties are halved until at least $\min(n, |P|)$ groups are generated. Finer-grained analysis may be integrated if desired, e.g. considering affinity of combinational gates in the combinational fan-in of these properties. Group rollback for higher levels is more intricate (Fig. 6), with the goal of *improving* group quality. A group with minimal quality is conservatively subdivided until at least $\min(n, |P|)$ groups are generated. A minimal-quality group is split to yield smaller,

halve_groups (**Groups** G, **Workers** n)
while $|G| < n$:
    Group g = pick largest non-singleton group from G
    $G = (G \setminus g) \cup$ halve_group (g) *# see below*

halve_group (**Group** g)
return {first half of g, second half of g} *# split in half*

Fig. 5. Algorithm for subdividing large level-1 groups in half.

rollback_groups (**Groups** G, **Netlist** N, **Level** l_c, **Affinity** t, **Workers** n)
while $|G| < n$:
    Group g = pick minimal-quality non-singleton group from G
    $G = (G \setminus g) \cup$ rollback_group (g, N, l_c, t) *# see below*

rollback_group(**Group** g, **Netlist** N, **Level** l_c, **Affinity** t)
Groups G = singletons(g) *# split g to singletons*
grouping_level_1 (G, N) *# level-1*
if $|G|$ == 1 : G = halve_group ($g \in G$) **return** G *# $|G|$ == 2*
else if $|G|$ == 2 : **return** G *# g had two 100% quality subgroups*
rollback_group_level (G, N, t, 2) *# level-2*
if $|G|$ == 2 : **return** G
if l_c == 3 : rollback_group_level (G, N, t, 3) *# level-3*
return G *# $|G|$ == 2*

rollback_group_level (**Groups** G, **Netlist** N, **Affinity** t, **Level** l)
Groups $G_c = G$ *# local copy of G*
Group g_0, $g_1 = \emptyset$ *# temporary groups, initially empty*
if l == 2 : grouping_level_2 (G_c, N, t) *# level-2*
else : grouping_level_3 (G_c, N, t) *# level-3*
if $|G_c|$ == 1 : *# G_c is one group containing all properties in G*
    g_0 = $g \in G$ containing center property g_c^*
    # extract most-distant property into distinct subgroup
    g_1 = $g \in G$ s.t. dist(g_0^*, g^*) == $max(\{$dist(g_0^*, g_i^*) $\mid \forall g_i \in G\})$
    for each group $g \in G$: *# merge groups to minimize distance*
        if dist(g_0^*, g^*) $\leq$ dist(g_1^*, g^*) : add properties in g to g_0
        else : add properties in g to g_1
    $G = \{g_0, g_1\}$ *# note $Q(g_0), Q(g_1) \geq Q(g_c)$, see Thm. 4*
else : $G = G_c$ *# $|G| \geq 2$*

Fig. 6. Algorithm for subdividing minimal-quality groups.

higher-quality subgroups. This process has negligible runtime, reuses precomputed support bitvectors and requires only a few milliseconds on the largest netlists.

The rebalancing procedure generates groups with quality bounds per Theorems 1, 2 and 3. Note that arbitrarily subdividing level-2,-3 groups without careful affinity consideration might violate affinity thresholds, because the quality of group g is measured with respect to its center property g^*. Assume that we generate subgroups g_0 and g_1 from g. If g^* is in g_0, we trivially have $Q(g_0^*) \geq Q(g^*)$ for any properties subgrouped with g^*. However, no such claim can be made about g_1; its properties might have been nearer to g^* than to each other. It is thus desirable to subdivide the most-distant property g_1^* from g^* to improve vs. risk degrading the resulting quality of both subgroups. Moreover, simply rolling back a higher level group to lower-level subgroups risks generating more groups than necessary, e.g., one level-2 group rolled back to ten level-1 groups. The algorithm in Fig. 3 generates *a minimal number* $|G|$ of high-affinity groups with provable affinity bounds, where $|G| \geq \min(n, |P|)$.

Theorem 4. *Given a group g, the rollback_group procedure subdivides g into two disjoint subgroups g_0 and g_1 such that $Q(g_0) \geq Q(g)$ and $Q(g_1) \geq Q(g)$.*

Proof. (Sketch) The algorithm returns two 100% affinity groups when properties in g generate at most two level-1 subgroups. Otherwise, the greatest-Hamming-distance property $g_1^* \in g$ from g's center property g^* is identified. Subgroup g_0 inherits g^* as its center, and g_1 inherits g_1^* as its center. Remaining properties in g are added to g_0 vs. g_1 to minimize distance from g_0^* vs. g_1^*, ensuring provable quality bounds. □

Corollary 4.1. *Given affinity t and level l, grouping for parallelism (Fig. 3) generates groups G such that $\forall g \in G$: a) $Q(g) = 1.0$ if $l = 1$, b) $Q(g) \geq t$ if $l = 2$, and c) $Q(g) \geq 3 * t - 2$ if $l = 3$.*

Proof. The proof follows per Theorems 1, 2 and 3 when no rebalancing occurs. Otherwise, rebalancing divides group g in to smaller groups based on: (i) $l = 1$, level-1 subgroups are generated and $Q(g) = 1.0$ per Theorem 1; (ii) $l = 2$, levels-1 or 2 subgroups are generated and $Q(g) \geq t$ per Theorems 2 and 4; and (iii) $l = 3$, levels-1, 2 or 3 subgroups are generated and $Q(g) \geq 3 * t - 2$ per Theorems 3 and 4. □

Theorem 5. *Given groups G over a set of properties P, and workers n with $|G| < n$ and $|P| \geq n$, rebalancing generates property groups G' such that $|G'| = n$.*

Proof. Both halve_group and rollback_group subdivide a non-singleton group g into exactly two subgroups, and iterate until $|G'| \geq n$. Therefore, the number of groups increases by exactly one in every iteration, unless all groups become singleton which cannot happen until $|G'| = |P| \geq n$. □

Corollary 5.1. *Given a set of properties P and n workers, grouping for parallelism (Fig. 3) generates groups G from P such that $|G| \geq \min(n, |P|)$.*

Proof. The proof trivially holds when $\geq n$ groups or $|P| \leq n$ singletons are generated without rebalancing. Otherwise, the proof holds per Theorem 5 when rebalancing occurs. □

B. Group Distribution Heuristics

We propose three heuristics to optimally utilize parallel workers, used on-the-fly by a *manager* that dispatches property groups and dynamically adjusts based upon worker feedback. When partitioning is supported by an engine within a strategy (e.g. a localization engine [3]), there might be multiple managers partitioning an identical or overlapping set of properties. It is sometimes beneficial to use a hierarchy of managers: the *root* might use lower-affinity partitioning onto parallel strategies, with higher-affinity partitioning within a strategy.

Iteration order (I): Fig. 3 orders groups deterministically, and thus distributed managers within a strategy will likely verify common properties in the same order. This results in redundant CPU-time, where two or more strategies may solve the same property at nearly the same time (**P3**). The root manager could instead dispatch disjoint properties to different workers, though there are motivations for building get_next_group (**Groups** G, **Netlist** N, **Level** l_c, **Affinity** t)

Group g = pick unsolved or inactive group from G
if g == null : **return** null *# all group are solved or active*
if unsolved(g) **and** inactive(g) : **return** g *# dispatch group*
if unsolved(g) : *# decompose (new groups are unsolved and inactive)*
    if l_c == 1 : $G = (G \setminus g) \cup$ halve_group(g) *# see Fig. 5*
    else $G = (G \setminus g) \cup$ rollback_group(g, N, l_c, t) *# see Fig. 6*
else remove g from G *# group is already solved*
goto 1 *# pick next group to dispatch*

Fig. 7. *Manager* routine to dispatch unsolved groups using decomposition.

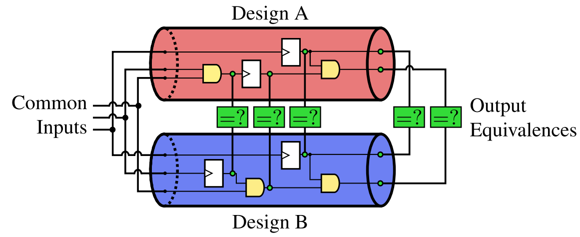


Fig. 8. Sequential equivalence checking uses redundancy removal to eliminate gate-equivalences between two logic designs. Each speculated gate-equality requires verifying a property called a *miter* (depicted as green box =?).

intelligence into distributed managers working on the entire property set, such as enabling incrementality and data sharing across properties [8]–[11]. To minimize redundant work, the manager may be augmented with options to iterate common groups in different orders: 1) smallest to largest COI (*forward*); 2) largest to smallest COI (*backward*); and 3) *random* to heuristically minimize concurrent solving of the same group while more groups than workers remain unsolved. If all properties are of comparable difficulty, running two identical strategies with opposite group ordering effectively halves wall-time with almost no redundant CPU-time. This approach can yield superlinear irredundant speedup when different strategies are tailored for easier vs more-difficult properties: a lighter strategy can iterate *forward* heuristically addressing easier properties first (the heavier strategy would be slower for these), while the heavier strategy can iterate *backward* addressing more-difficult properties first (the lighter strategy might be unable to solve these).

Controlled repetition (R): Each worker solves groups one-at-a-time. Encountering a difficult group inhibits overall progress (**P4**). Easier groups might follow, which when solved might speed-up incremental verification of the previous difficult group. Furthermore, solving easy properties sooner benefits other workers, allowing them to focus on fewer difficult groups. It is thus beneficial to impose time-limits per group within certain *fast* strategies. The manager must be capable of pruning already-solved properties (possibly solved by different workers), and repeating groups up to a configurable maximum allowed repetitions (to reduce redundant CPU-time). It may be beneficial to increase resource limits between repetitions, possibly after n repetitions with no progress. Engine incrementality is fairly important when imposing time-limits and repetition, to minimize redundant CPU-time.

Decomposition (D): Some groups are more difficult than others, either because they are large (e.g., many properties), or because individual properties therein are more difficult (e.g., having a very-deep counterexample). Some workers might be slower than others, possibly due to varying machine load. A common wall-time degradation occurs when fewer difficult groups than workers remain, and previously-active workers become idle (**P2**). This heuristic decomposes unsolved groups and dispatches them to idle workers, to accelerate convergence despite imposing some redundant CPU-time. Rather than redundantly dispatching an entire unsolved group, this heuristic utilizes the algorithms of Fig. 5 and Fig. 6 to subdivide unsolved groups to smaller and higher-affinity groups, eventually becoming singletons. Smaller groups are easier for idle workers to redundantly solve (**P5**), benefiting but not preempting active workers (which might be on the verge of solves). The corresponding manager with decomposition is shown in Fig. 7. A group is *inactive* when no worker is currently verifying it. Solved properties and groups are discarded; groups with *unsolved* properties are subdivided and redundantly dispatched. Singleton groups are not redundantly dispatched, being *inactive* after the first dispatch.

IV. Localization for Redundancy Removal

Industrial hardware designs are often rife with redundancy, e.g. to boost the performance of semiconductor devices, and to implement features such as error resilience, security, initialization logic and post-silicon observability. Verification testbenches yield additional netlist redundancies, due to *input constraints* restricting the set of stimulus applied to the design, and due to redundancies arising between the design and synthesized properties. Equivalence checking can be viewed as verifying a *composite netlist* comprising two designs as per Fig. 8. *Sequential redundancy removal* [12]–[14, 16]–[18, 27] (Fig. 9) is the process of proving that equivalence-classes of gates evaluate to equal or opposite values in all reachable states; each speculated redundancy entails solving a property called a *miter*. When a miter is proven, the corresponding redundant gates can be merged. This COI reduction is highly beneficial to verification scalability, and is the core procedure of sequential equivalence checking (SEC).

Various heuristics control the scope of equivalence-class candidates affecting runtime vs. reduction (Fig. 9 Step 1): e.g. whether to consider only registers vs. all gate types; whether to prune classes to reflect *corresponded signal names* or require per-class candidates spanning both designs in an equivalence-checking context (Fig. 8) [14, 20]. A *speculatively-reduced netlist* (Steps 2-3) accelerates verification of the miters. Techniques such as BMC and guided simulation are typically used to falsify miters; then induction proves the easier miters; and finally multi-engine strategies prove the difficult miters or find difficult counterexamples (Steps 4,5). Failed proofs (falsified miters or inconclusive results) cause a *refinement* of the equivalence classes to separate unproven miters' gates, then another expensive proof iteration is performed. Our goal is to minimize inconclusive proofs to achieve maximum netlist reduction with

```
redundancy_removal (Netlist N)
1: Guess the redundancy candidates - sets of equivalence classes of gates
   in N, where gate u in class Q(u) is suspected equivalent to every other
   gate v in the same equivalence class.
2: Select a representative gate R(Q(u)) from each class Q(u).
3: Construct the speculatively-reduced netlist by replacing source gate u
   of every edge (u, v) ∈ E by R(Q(u)). Additionally, for each gate v,
   add a miter property asserted when v ≢ R(q(v)).
4: Attempt to prove that each miter is unassertable.
5: If a miter cannot be proven unassertable, refine the equivalence classes
   to separate the corresponding gates, and goto Step 2.
6: For all unassertable miters, merge the corresponding gates onto the
   representative to eliminate redundancy.
```

Fig. 9. Generic sequential redundancy removal framework [16].

```
fast_lossy_localization (Group g, unsigned n, Timeout T)
1: Netlist L = load_incremental_abstraction(g) # initially empty
2: unsigned k = load_incremental_bmc_depth(g) # initially 0
3: while elapsed_time() ≤ T and unsolved(g) :
4:     localize_bmc (g, L, k, unchanged)  # see below
       # check if netlist unchanged for last n bmc steps
5:     if unchanged < n : k = k + 1, goto 4  # increment depth
6:     run_proof_strategy(L, g, T - elapsed_time())
7: save_incremental_data (G, k, L)  # timeout: save incremental data

localize_bmc (Group g, Netlist L, unsigned k, unsigned unchanged)
1: bool stop = 0  # some properties fail at depth k
2: while not stop : # loop until all properties pass at depth k
3:     Gates c = {}, stop = 1  # cutpoints to refine, initially empty
4:     for each Property p ∈ g :
5:         Result r = run_bmc(L, p, k)  # run bmc with depth k
6:         if r == unsat : continue  # property passes
7:         if cex not spurious : report_solved(p, cex), continue
8:         stop = 0 # property fails
9:         Gates d = cutpoints_to_refine(), c = c ∪ d
10:    if not stop : refine_abstraction(L, c), unchanged = 0
11:    else unchanged + = 1  # no change in abstraction
```

Fig. 10. *Fast-and-Lossy* localization with incremental repetition of high-affinity property groups.

minimal wall- *and CPU*-time, using a parallel localization portfolio. Note that even if a testbench has only a single property, redundancy removal will often create thousands of miters. The large number of miters often tremendously benefit from parallel processing, as noted for combinational redundancy removal [19] and induction [18]. These miters are distributed throughout the netlist, making affinity partitioning particularly beneficial. Since practical netlists comprise a diversity of logic, different miters benefit from different strategies.

The proof or counterexample of a property often only depends on a small subset of logic in its COI. *Localization* [21]–[24] is a powerful abstraction method to reduce COI size by replacing irrelevant gates by *cutpoints* or unconstrained primary inputs. Since cutpoints can simulate the behavior of the original gates and more, the abstracted netlist over-approximates the behavior of the original netlist: abstract proofs imply original proofs, but abstract counterexamples might be spurious. Abstraction *refinement* eliminates cutpoints deemed responsible for spurious counterexamples, re-introducing previously-eliminated logic. It is desirable that the abstract netlist be as small as possible to enable scalable verification, while being immune to spurious counterexamples.

Localization is often essential to solve non-inductive miters, leveraging speculative reduction to abstract nearly all logic except for differently-implemented yet functionally-equivalent logic *between* speculated equivalences [14, 16]. Without localization, the COI of a miter may be very large despite speculative reduction. This large COI size may choke even fairly-scalable provers such as IC3. While the benefits of localization for sequential redundancy removal are well-known [17], prior work considered only single-process miter verification, aside from use of a standard parallel model-checking portfolio to solve miters [20]. Ours is the first to optimize a parallel localization portfolio in this (or any multi-property) context, using property partitioning and irredundant scheduling procedures (Figs. 3 and 7), along with the following complementary strategies tailored for easier vs. difficult properties. Note that substrategies in either may be employed by the other.

A. Fast-and-Lossy Localization

Fast-and-Lossy localization (Fig. 10) attempts to quickly discharge easier property groups, using timeouts to skip difficult groups. If the group is not solved within the allotted time, verification data (e.g., the current abstract netlist and achieved BMC depth) is saved for incremental reuse to accelerate later repetition. Skipped groups can be repeated as-is, or rebalanced (Fig. 7) after several repetitions of no progress. Note that repeating a group as-is may likely proceed further upon repetition, by incrementally skipping earlier processing and since a different worker might have solved some properties therein. *Fast-and-Lossy* localization uses counterexample-based refinement sometimes with quick proof-based abstraction (PBA), possibly yielding larger abstract netlists that are more-difficult to prove but with less time expended in BMC itself [23] for faster performance on easier groups. When ready to prove (i.e., no refinements occur for n consecutive BMC steps), abstracted groups are passed to a sequence of lighter reduction engines then IC3 [5, 28]) under a modest time-limit (e.g. $\leq 300s$) which can be increased across repetitions (**R**).

B. Aggressive Localization

Aggressive localization (Fig. 11) is aimed at solving difficult properties, where *Fast-and-Lossy* may fail due to larger-than-necessary abstractions, insufficient reductions prior to IC3, or small group time-limits. *Aggressive* never repeats groups, so either imposes no time limit whatsoever, or a large time-limit as shown applied to semantically-partitioned (Sec. IV-C) subgroups but iterated and increased until the group is solved. *Aggressive* typically uses a hybrid of counterexample-based refinement and PBA run after every unsatisfiable BMC result, to yield smaller abstractions than the former alone to accelerate subsequent proofs at the expense of more runtime spent in BMC itself [23]. When ready to prove (i.e., no refinements occur for n consecutive BMC steps), abstracted groups are passed to a sequence of heavy reduction engines (including nested induction-only sequential redundancy removal across

```
aggressive_localization (Group g, unsigned n, bool pba, bool semantic,
                         Affinity t, Timeout T, Multiplier m)
1:  Netlist L = initial_abstraction(g)  # initially empty
2:  unsigned k = 0  # bmc depth
3:  localize_bmc (g, L, k, unchanged)  # see Fig. 10
4:  if semantic : collect_support_info (...)  # see Sect. IV-C
5:  if pba : minimize L using proof-based abstraction
    # check if netlist unchanged for last n bmc steps
6:  if unchanged < n : k = k + 1, goto 3  # increment depth
7:  Groups Ĝ = semantic ? structural_grouping (g, L, 3, t) : G
    # Sort via (I) mode (Sect. III-B): forward, backward, or random
8:  Sort Ĝ by abstract COI size
9:  for each unsolved group ĝ ∈ Ĝ :
10:     while elapsed_time() ≤ T and unsolved(ĝ) :
11:         run_proof_strategy(L, ĝ, T - elapsed_time())
12: if unsolved groups remain : T = T × m, goto 9
```

Fig. 11. *Aggressive* localization with semantic partitioning, counterexample- and proof-based abstraction.

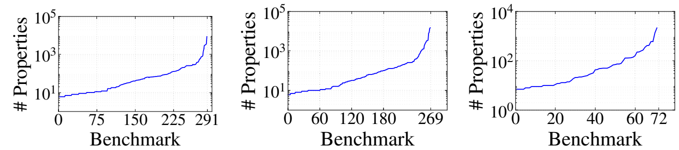
all gates, which might be too expensive to converge on large netlists before localization) followed by IC3 [5, 28]).

C. Semantic Partitioning

Semantic partitioning [3] refers to re-partitioning a group whose *unabstracted COI* was high-affinity, yielding subgroups of high affinity with respect to *abstract COI* as correlates to subsequent verification complexity. Abstract COI information is mined onto support bitvectors on a per-property basis as cutpoints are refined (Fig. 11 Step 4), considering minimized counterexamples for individual properties despite incrementally using the same BMC instance for the entire group. The group is partitioned into smaller, high-localized-affinity subgroups (Step 7) before attempting to prove.

Improvements to semantic partitioning vs. [3]: Per-property abstract-COI bloat may arise during counterexample analysis, because the group must be mutually refined to be free of spurious counterexamples. Eager partitioning (as soon as any diverged abstract COI occurs) *could* circumvent this ambiguous bloat, though often severely hurts performance since intermediate abstract-COI differences often reconverge. In practice, lazy partitioning deferred until modest BMC time limits are exceeded is far superior (particularly since BMC often benefits from level=3 lower affinity), retaining high-affinity atomic verification benefits. Abstract-COI ambiguities can be largely corrected during proof analysis, by analyzing a distinct proof per property. Incremental data should be saved when semantically re-partitioning, to minimize restart penalty.

Difficult sub-groups are susceptible to delaying easier later sub-groups. Subgroups should be ordered as per **(I)** mode (Sect. III-B): *forward*, *backward*, and *random*, configured differently in parallel strategies for better portfolio performance with less redundant CPU-time. Subgroups are verified in the chosen order using controlled repetition (**R**) and large *Aggressive* time-limits (Steps 9–11). We recommend T $\geq$ 1h multiplying 2$\times$ at each iteration (Step 12) and overriding to *unlimited* when a single sub-group remains.



(a) Set B1 (b) Set B2 (c) Set B3

Fig. 12. Number of properties per benchmark set.

V. Experimental Results

We evaluate our techniques within the post-induction proof strategy of a sequential redundancy removal framework (Fig. 9). To eliminate *noise* such as different counterexamples yielding different equivalence-classes (Step 5), we snapshot the speculatively-reduced netlist after ten minutes of induction, before the final iteration of a six-hour eight-process semi-formal bug-hunting [29] and localization portfolio to eliminate most incorrect and easier [27] miters. The following experiments[2] are run on these snapshotted netlists (pruning those with fewer miters than processes), yielding three benchmark sets. Set **B1** (Fig. 12a) are the most-difficult 291 of 1822 proprietary SEC benchmarks, where initial equivalence classes comprise original properties and *name corresponded* register pairs. Set **B2** (Fig. 12b) has 269 netlists derived from the former, including a large equivalence class for registers without name correlation. Set **B3** has 72 netlists from the SINGLE property HWMCC 2017 benchmarks, comprising a large initial equivalence class of all registers. Our techniques are implemented within *RuleBase: Sixthsense Edition* [30].

A. Localization Portfolio

We select our localization portfolio (Table I) from extensive evaluation of 36 single-process localization configurations and 30 subsequent proof strategies, exploring options such as enabling vs. disabling PBA [23]; different levels of property grouping vs. no grouping [3]; enabling vs. disabling semantic partitioning (Sect. IV-C); and different policies for group iteration **(I)**, repetition **(R)**, and decomposition **(D)** (Sect. III-B). The best-performing collection is chosen, maximizing *complementary* unique solves. *Aggressive* localization (Sect. IV-B) primarily uses both counterexample- and proof-based abstraction, yielding smallest abstract netlists solved with a single-process heavy strategy of combinational rewriting; input elimination [31]–[33] which is especially powerful after localization due to inserted cutpoints; min-area retiming [34]; a nested induction-only gate-based sequential redundancy removal; then IC3. *Fast-and-Lossy* localization (Sect. IV-A) uses counterexample-based refinement mainly with no or lighter PBA for faster BMC, yielding larger abstract netlists solved using light combinational rewriting, input elimination, then IC3. The former is fastest for difficult properties; the latter for easier properties.

We compare four 6-process localization portfolios derived from Table I. The localization configuration and subsequent solving strategy of each process is identical across portfolios,

[2] Detailed results available at http://temporallogic.org/research/FMCAD20

TABLE I
SIX-PROCESS COMPLEMENTARY LOCALIZATION PORTFOLIO.

#	Localization Strategy	Grouping Level	Semantic	Iteration (I)	Repetition (R)	Decomposition (D)
S1	*Fast-and-Lossy*	Level-1	✗	Forward	✓	✗
S2	*Fast-and-Lossy*	Level-1	✗	Reverse	✓	✓
S3	*Fast-and-Lossy*	Level-3	✓	Forward	✓	✓
S4	*Aggressive*	Level-1	✗	Forward	✗	-
S5	*Aggressive*	Level-1	✗	Reverse	✗	-
S6	*Aggressive*	Level-3	✓	Forward	✗	-

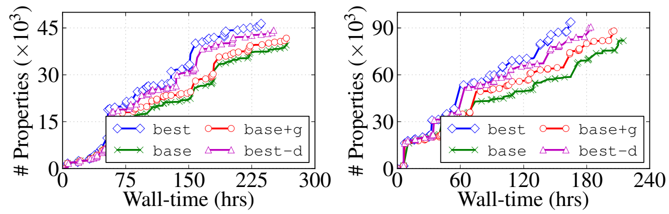
except for adherence to the illustrated scheduling differences as discussed below. For greater portfolio value, each process includes localization configuration differences beyond the illustrated scheduling distinction in Table I. **S1** only performs counterexample-based refinement; **S2** and **S3** also perform PBA. **S2** vs. **S3** perform hybrid counterexample-based refinement with light PBA (modest time limit) after every unsatisfiable BMC step vs. only before the subsequent solving strategy, respectively. Abstract-netlist gates remaining after PBA are considered *committed* and cannot be eliminated in later PBA steps [21] in **S2**, but not **S3**. **S3** utilizes a minimal unsatisfiable core to further reduce the abstract netlist. **S4-S6** are identical to **S1-S3**, respectively, without imposed time-limits and modulo the above-mentioned post-localization solving strategy differences. To highlight our individual contributions, we compare four variants of this portfolio:

1) `base`: No property grouping or incremental repetition of properties; all processes iterate properties in forward order. This represents a standard state-of-the-art localization portfolio approach *without property grouping*, e.g., before [3].
2) `base+g` extends `base` with affinity property grouping, including semantic partitioning in one *Fast-and-Lossy* and one *Aggressive* strategy. This represents a state-of-the-art localization portfolio *with property grouping*, e.g., as per [3] though with our semantic refinement improvements of Sect. IV-C.
3) `best-d` extends `base+g` with incremental repetition **(R)** and irredundant iteration order **(I)**, to reduce CPU-time.
4) `best` extends `best-d` with decomposition **(D)**.

Processes S1-S6 are generic online localization strategies. Multi-property localization *without affinity-partitioning* generally yields poor/noncompetitive performance [3], eroding most of its scalability benefit, especially for difficult miters. (Recall that these benchmarks pre-filter easier miters, using induction and semi-formal bug-hunting.) Therefore, `base` and `base+g` are highly-competitive 6-process localization portfolios, for online "first-run-of-a-testbench." Industrial verification tools may use more processes for large testbenches, and may post-process data from prior/ongoing runs to accelerate future results. This level of sophisticated benchmark-specific orchestration is valuable, though does not readily benefit "first-run-of-a-testbench" and introduces noise in experiments hence are not used herein. We optimize runtime of a generic 6-process localization portfolio without per-benchmark customization.

B. Experiment Setup

Our experiments run on a computing grid with identical x86 Linux nodes. Each benchmark run uses a 6-process portfolio



(a) Set B1 (b) Set B2

Fig. 13. #Properties solved vs. wall-time for **B1** and **B2**; 6-hour time limit.

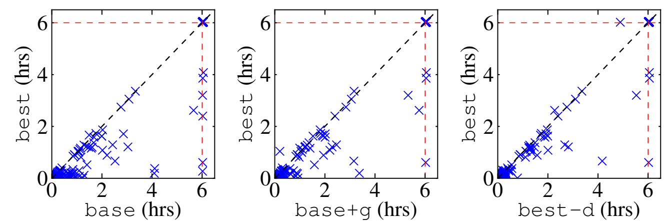


Fig. 14. **best** vs. baselines for **B1** (points below diagonal are in favor).

(Table I); each process **S1-S6** runs on a single identical CPU core on the same host-machine. Each process eagerly cancels solved properties across all processes in that portfolio, to reduce redundant computation.

While most prior research and competitions focus solely upon optimizing wall-time, our techniques additionally benefit CPU-time. Traditionally, Fast-and-Lossy (unlike Aggressive) processes terminate early, leaving unsolved difficult properties. In these experiments, `base` and `base+g` augment Fast-and-Lossy processes to naively repeat identically-configured **S1-S3** with identical resource limits per group (whereas `best-d` and `best` add incremental-repetition **(R)** with resource-doubling across repetitions), until all properties are solved or global timeout. This naive repetition is wasteful in practice, yielding highly-redundant CPU-time for marginal benefit. However, disabling naive repetition in these experiments yielded 3.2% fewer solves in `base` and `base+g` vs. `best-d` and `best`, which arguably unfairly penalized them as state-of-the-art solutions *before our contributions*. Therefore, **S1-S6** in each portfolio continue working until all processes terminate, hence CPU-time is approximately 6× wall-time in these experiments.

C. Proprietary Benchmarks

Fig. 13 shows the number of properties solved vs. wall-time for **B1** and **B2**. `best` is the clear winner, solving 18.1% (15.3%) more properties in 17.2% (22.9%) less time for **B1** (**B2**, respectively) compared to `base`. Affinity-grouping significantly improves performance of `base+g` over `base`. Level-3 grouping with our semantic partitioning improvements (Sect. IV-C) benefits *Aggressive*, atomically solving properties in fewer, larger high-abstract-affinity groups compared to level-1,-2. Incremental repetition and irredundant iteration allows `best-d` to solve 8.1% more properties than `base+g`, less-severely hindered by difficult groups. `best` yields additional solves through decomposition of difficult groups after five incremental repetitions of no progress, solving all properties in 4 vs. 6 benchmarks in **B1** vs. **B2** that time out with

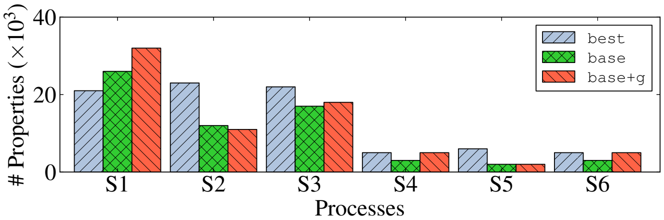
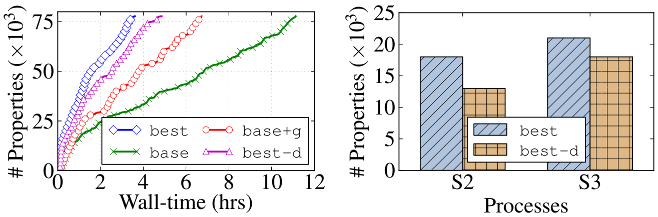


Fig. 15. #Properties solved on **B2** per process of Table I.

TABLE II
UTILITY OF AGGRESSIVE STRATEGY PROCESSES IN A PORTFOLIO.

Portfolio	Set B1		Set B2	
	#Solved	Time (h)	#Solved	Time (h)
3× *Fast-and-Lossy*, 3× *Aggressive*	**46,844**	**236**	**93,806**	**165**
6× *Fast-and-Lossy* (modified `best`)	41,702	275	91,639	184



(a) Properties vs. time

(b) Work distribution

Fig. 16. #Properties solved vs. wall-time for **big**: (a) by all portfolios; (b) per process of Table I within `best` and `best-d`.

other portfolios. Fig. 14 details per-**B1**-benchmark runtimes of `best`, yielding a median speedup of 2.4×, 2.0× and 1.5× vs. `base`, `base+g`, and `best-d`, respectively.

Fig. 15 shows the distribution of properties solved per process (Table I) within these portfolios. The percentage solved by each *Fast-and-Lossy* (and *Aggressive*) process is nearly uniform in `best`, showing near-optimal irredundant work distribution. In contrast, without **(I)** and **(R)**, `base` and `base+g` have highly-uneven distributions due largely to parallel processes addressing the same groups concurrently. While the number of solved (easier) miters is considerably larger with *Fast-and-Lossy*, we emphasize how critical the *Aggressive* solution of difficult miters is to the overall redundancy removal process. If any are left unsolved, Fig. 9 Step 5 will forgo attempting to merge the corresponding gates, thereby weakening netlist reductions, risking unsolved SEC, and hurting runtime by requiring yet another expensive proof iteration with refined equivalence classes [14] – where fanout miters often become more-difficult than those unsolved in prior iterations. Table II shows the number of properties solved by `best`, and a modified `best` portfolio with all *Fast-and-Lossy* strategy processes where **S4**-**S6** are identical to **S1**-**S3** respectively, but without imposed time-limits and iterating groups in opposite order. Without *Aggressive* processes in the portfolio, modified `best` solves 10.9% (2.31%) fewer properties in 16.5% (11.51%) more time for **B1** (**B2**).

To further highlight the value of decomposition **(D)**, Fig. 16b illustrates an additional **big** benchmark containing 77728 properties partitioned into 9958 level-1 and level-2, and 2991 level-3 high-affinity groups. Fig. 16a shows the number of properties solved by each portfolio vs. time. `best` is 3.0× faster than `base`. Fig. 16b shows the number of properties solved by two *Fast-and-Lossy* processes of `best` and `best-d`; decomposition enables **S2** and **S3** in `best` to collectively solve 25.2% more properties than `best-d`.

D. HWMCC Benchmarks

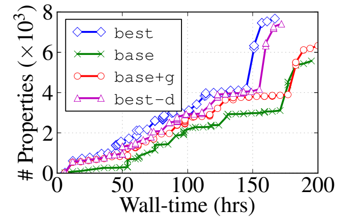


Fig. 17. #Solved vs. wall-time for **B3**.

Fig. 17 shows the number of properties solved by each portfolio for set **B3**. `best` is again the winner, solving 3054 more properties in less time than `base`. Incremental repetition and irredundant iteration is particularly beneficial in this set: several benchmarks have counterexamples that are discovered in earlier group repetitions, enabling *Aggressive* and later *Fast-and-Lossy* repetitions to direct resource upon more-difficult provable miters.

VI. CONCLUSIONS AND FUTURE WORK

We focus upon boosting the scalability of multi-property parallel verification, with application to sequential redundancy removal using a localization portfolio. Our contributions optimize both wall-time and CPU-time, orchestrating via complementary strategy exploration and property partitioning. **(1)** We extend scalable affinity-based property partitioning to guarantee *complete* utilization of available processes with provable partition affinities. **(2)** We propose improvements to the scheduling of parallel processes, such as resource-constrained irredundant iteration, incremental repetition, and decomposition of difficult groups. **(3)** We deliver a carefully-optimized localization portfolio, self-tailoring to irredundantly address a range of property difficulties through a synergistic balance of *Fast-and-Lossy* vs. *Aggressive* configurations. **(4)** We propose improvements to *semantic group partitioning* within localization, boosting scalability by enabling the BMC within localization to benefit from larger and slightly-lower affinity groups, then optimally sub-dividing those groups before solving the localized properties. To our knowledge, this is the first published approach to optimize both property partitioning and strategy exploration within a multi-property localization portfolio. Experiments confirm that this solution works well across large suites of benchmarks.

Note that our mutually-optimized partitioning vs. strategy-exploration orchestration offers broad insights early in an ongoing verification-tool run, whereas traditional orchestration typically explores only easier (smaller-COI) properties or only a subset of strategies early in the run. Exploring how this insight may enable dynamic benchmark-specific customized orchestration *during* an ongoing run is a promising future direction, e.g. dynamically adjusting which strategy is used per process and partition. Exploring these techniques across a broader set of engines, and exploring incrementality of strategies across localization and equivalence-class refinements, are additional promising research directions.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback and suggestions. We thank Alexander Ivrii for assistance in implementing various localization features and for feedback on early drafts of this paper. We thank Raj Kumar Gajavelly for providing benchmarks and assistance with experimental evaluation. This work is partially supported by National Science Foundation CAREER Award CNS-1664356.

REFERENCES

[1] G. Cabodi, P. E. Camurati, C. Loiacono, M. Palena, P. Pasini, D. Patti, and S. Quer, "To split or to group: from divide-and-conquer to subtask sharing for verifying multiple properties in model checking," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 20, pp. 313–325, Jun 2018.

[2] G. Cabodi and S. Nocco, "Optimized model checking of multiple properties," in *Design, Automation and Test in Europe (DATE)*, pp. 1–4, Mar 2011.

[3] R. Dureja, J. Baumgartner, A. Ivrii, R. Kanzelman, and K. Y. Rozier, "Boosting verification scalability via structural grouping and semantic partitioning of properties," in *Formal Methods in Computer Aided Design (FMCAD)*, pp. 1–9, Oct 2019.

[4] R. Dureja and K. Y. Rozier, "More scalable LTL model checking via discovering design-space dependencies (D^3)," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (D. Beyer and M. Huisman, eds.), (Cham), pp. 309–327, Springer International Publishing, Apr 2018.

[5] A. R. Bradley, "SAT-based model checking without unrolling," in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, (Berlin, Heidelberg), p. 7087, Springer-Verlag, 2011.

[6] S. Chaki and D. Karimi, "Model checking with multi-threaded IC3 portfolios," in *Verification, Model Checking, and Abstract Interpretation (VMCAI)* (B. Jobstmann and K. R. M. Leino, eds.), (Berlin, Heidelberg), pp. 517–535, Springer Berlin Heidelberg, Jan 2016.

[7] M. Marescotti, A. Gurfinkel, A. E. J. Hyvärinen, and N. Sharygina, "Designing parallel PDR," in *Formal Methods in Computer-Aided Design (FMCAD)*, (Austin, Texas), p. 156163, FMCAD Inc, Oct 2017.

[8] Z. Khasidashvili, A. Nadel, A. Palti, and Z. Hanna, "Simultaneous SAT-based model checking of safety properties," in *Hardware and Software, Verification and Testing (HVC)* (S. Ur, E. Bin, and Y. Wolfsthal, eds.), (Berlin, Heidelberg), pp. 56–75, Springer Berlin Heidelberg, 2006.

[9] Z. Khasidashvili and A. Nadel, "Implicative simultaneous satisfiability and applications," in *Hardware and Software: Verification and Testing (HVC)* (K. Eder, J. Lourenço, and O. Shehory, eds.), (Berlin, Heidelberg), pp. 66–79, Springer Berlin Heidelberg, 2012.

[10] R. Dureja and K. Y. Rozier, "FuseIC3: An algorithm for checking large design spaces," in *Formal Methods in Computer Aided Design (FMCAD)*, pp. 164–171, Oct 2017.

[11] J. Marques-Silva, "Interpolant learning and reuse in SAT-based model checking," *Electronic Notes in Theoretical Computer Science*, vol. 174, no. 3, pp. 31 – 43, 2007.

[12] C. A. J. van Eijk, "Sequential equivalence checking without state space traversal," in *Design, Automation and Test in Europe (DATE)*, pp. 618–623, Feb 1998.

[13] P. Bjesse and K. Claessen, "SAT-based verification without state space traversal," in *Formal Methods in Computer-Aided Design (FMCAD)* (W. A. Hunt and S. D. Johnson, eds.), (Berlin, Heidelberg), pp. 409–426, Springer Berlin Heidelberg, Oct 2000.

[14] H. Mony, J. Baumgartner, V. Paruthi, and R. Kanzelman, "Exploiting suspected redundancy without proving it," in *Design Automation Conference (DAC)*, pp. 463–466, Jun 2005.

[15] K. Debnath, R. Murgai, M. Jain, and J. Olson, "SAT-based redundancy removal," in *Design, Automation and Test in Europe (DATE)*, pp. 315–318, Mar 2018.

[16] H. Mony, J. Baumgartner, A. Mishchenko, and R. Brayton, "Speculative reduction-based scalable redundancy identification," in *Design, Automation and Test in Europe (DATE)*, pp. 1674–1679, Apr 2009.

[17] J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman, and G. Janssen, "Scalable sequential equivalence checking across arbitrary design transformations," in *2006 International Conference on Computer Design*, pp. 259–266, Oct 2006.

[18] A. Mishchenko, M. Case, R. Brayton, and S. Jang, "Scalable and scalably-verifiable sequential synthesis," in *International Conference on Computer-Aided Design*, 2008.

[19] V. N. Possani, A. Mishchenko, R. P. Ribas, and A. I. Reis, "Parallel combinational equivalence checking," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Oct 2019.

[20] R. Brayton, N. Een, and A. Mishchenko, "Using speculation for sequential equivalence checking," in *International Workshop on Logic and Synthesis (IWLS)*, Jun 2012.

[21] A. Mishchenko, N. Een, R. Brayton, J. Baumgartner, H. Mony, and P. Nalla, "GLA: Gate-level abstraction revisited," in *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1399–1404, March 2013.

[22] K. L. McMillan and N. Amla, "Automatic abstraction without counterexamples," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (H. Garavel and J. Hatcliff, eds.), (Berlin, Heidelberg), pp. 2–17, Springer Berlin Heidelberg, 2003.

[23] N. Amla and K. L. McMillan, "A hybrid of counterexample-based and proof-based abstraction," in *Formal Methods in Computer-Aided Design (FMCAD)* (A. J. Hu and A. K. Martin, eds.), (Berlin, Heidelberg), pp. 260–274, Springer Berlin Heidelberg, 2004.

[24] P. Chauhan, E. Clarke, J. Kukula, S. Sapra, H. Veith, and D. Wang, "Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis," in *Formal Methods in Computer-Aided Design (FMCAD)* (M. D. Aagaard and J. W. O'Leary, eds.), (Berlin, Heidelberg), pp. 33–51, Springer Berlin Heidelberg, 2002.

[25] G. Cabodi, P. Camurati, and S. Quer, "A graph-labeling approach for efficient cone-of-influence computation in model-checking problems with multiple properties," *Software: Practice and Experience*, vol. 46, no. 4, pp. 493–511, 2016.

[26] R. Tarjan, "Depth first search and linear graph algorithms," in *SIAM Journal on Computing*, 1972.

[27] M. Case, J. Baumgartner, H. Mony, and R. Kanzelman, "Optimal redundancy removal without fixedpoint computation," in *Formal Methods in Computer-Aided Design (FMCAD)*, pp. 101–108, Oct 2011.

[28] N. Een, A. Mishchenko, and R. Brayton, "Efficient implementation of property directed reachability," in *Formal Methods in Computer-Aided Design (FMCAD)*, (Austin, TX), pp. 125–134, FMCAD Inc, 2011.

[29] P. K. Nalla, R. K. Gajavelly, J. Baumgartner, H. Mony, R. Kanzelman, and A. Ivrii, "The art of semi-formal bug hunting," in *International Conference on Computer-Aided Design (ICCAD)*, (New York, NY, USA), ACM, 2016.

[30] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, "Scalable automated verification via expert-system guided transformations," in *Formal Methods in Computer-Aided Design (FMCAD)* (A. J. Hu and A. K. Martin, eds.), (Berlin, Heidelberg), pp. 159–173, Springer Berlin Heidelberg, 2004.

[31] J. Baumgartner and H. Mony, "Maximal input reduction of sequential netlists via synergistic reparameterization and localization strategies," in *Correct Hardware Design and Verification Methods*, Oct 2005.

[32] N. Eén and A. Mishchenko, "A fast reparameterization procedure," in *International Workshop on Design and Implementation of Formal Tools and Systems*, 2013.

[33] R. K. Gajavelly, J. Baumgartner, A. Ivrii, R. L. Kanzelman, and S. Ghosh, "Input elimination transformations for scalable verification and trace reconstruction," in *Formal Methods in Computer-Aided Design (FMCAD)*, 2019.

[34] A. Kuehlmann and J. Baumgartner, "Transformation-based verification using generalized retiming," in *Computer Aided Verification (CAV)* (G. Berry, H. Comon, and A. Finkel, eds.), (Berlin, Heidelberg), pp. 104–117, Springer Berlin Heidelberg, 2001.