# A Fast Reparameterization Procedure

Niklas Een, Alan Mishchenko

{een,alanmi}@eecs.berkeley.edu

Berkeley Verification and Synthesis Research Center
EECS Department
University of California, Berkeley, USA.

**Abstract.** Reparameterization, also known as range preserving logic synthesis, replaces a logic cone by another logic cone, which has fewer inputs while producing the same output combinations as the original cone. It is expected that a smaller circuit leads to a shorter verification time. This paper describes an approach to reparameterization, which is faster but not as general as the previous work. The new procedure is particularly well-suited for circuits derived by localization abstraction.

## 1 Introduction

The use of reparameterization as a circuit transformation in the verification flow was pioneered by Baumgartner et. al. in [1]. In their work, new positions for the primary inputs (PIs) are determined by finding a minimum cut between the current PI positions and the next-state variables (flop inputs). BDDs are then used to compute the *range* (or image) on the cut. Finally, a new logic cone with the same range (but fewer PIs), is synthesized from the BDDs and grafted onto the original design in place of the current logic cone. This is a very powerful transformation, but it has potential drawbacks: (i) the BDDs may blow up and exhaust the memory, (ii) the extracted circuit may be larger than the logic it replaces, and (iii) the runtime overhead may be too high.

In contrast, the proposed approach is based on greedy local transformations, capturing only a subset of optimization opportunities. However, memory consumption is modest, runtimes are very low, and the resulting design is always smaller, or of the same size, as the original design. It is shown experimentally that the proposed method leads to sizeble reductions when applied for circuits produced by localization abstraction [5].

## 2 Fast Reparameterization

The fast reparameterization algorithm is based on the following observation: if a node dominates[1] a set of PIs, and those PIs are sufficient to force both a zero and a one at that node, regardless of the values given to the other PIs and state variables, then that node can be replaced by a new primary input, while the unused logic cone driving

---

[1] A node $n$ dominates another node $m$ iff every path from $m$ to a primary output goes through node $n$.

the original node can be removed. The old primary inputs dominated by the given node are also removed by this procedure.

> **Example.** Suppose a design contains inputs $x_1$, $x_2$, and a gate $\mathrm{XOR}(x_1, x_2)$; and that furthermore, $x_1$ has no other fanouts besides this XOR-gate. Then, no matter which value $x_2$ takes, both a zero and a one can be forced at the output of the XOR by setting $x_1$ appropriately, and thus the XOR-gate can, for verification purposes, be replaced by a primary input.

The proposed method to find similar situations starts by computing all dominators of the netlist graph, then for each candidate node dominating at least one PI the following quantification problem is solved: "*for each* assignment to the non-dominated gates, *does there exist* a pair of assignments to the dominated PIs that results in a zero and a one at the candidate node". More formally, assuming that $x$ represents non-dominated gates ("external inputs") and $y_i$ represents dominated PIs ("internal inputs"), the following is always true for the node's function $\phi$:

$$\forall x \; \exists y_0, y_1 \; . \; \neg\phi(x, y_0) \land \phi(x, y_1)$$

Important features of this approach are:

(i) It is circuit based, while early work on reparameterization was based on transition relations [4].

(ii) In its simplest form, the proposed restructuring replaces some internal nodes by new primary inputs and remove dangling logic.

(iii) The analysis is completely combinational: no information on the reachable state-space is used.

(iv) If the property was disproved after reparametrization, it is straight-forward to remap the resulting counterexample to depend on the original primary inputs.

It is important to realize that by analyzing and applying reductions in topological order from PIs to POs, regions amenable to reparameterization are gradually reduced to contain fewer gates and PIs. By this process, the result of repeatedly applying local transformations can lead to a substantial global reduction. In the current implementation, the above formula is evaluated by exhaustive simulation of
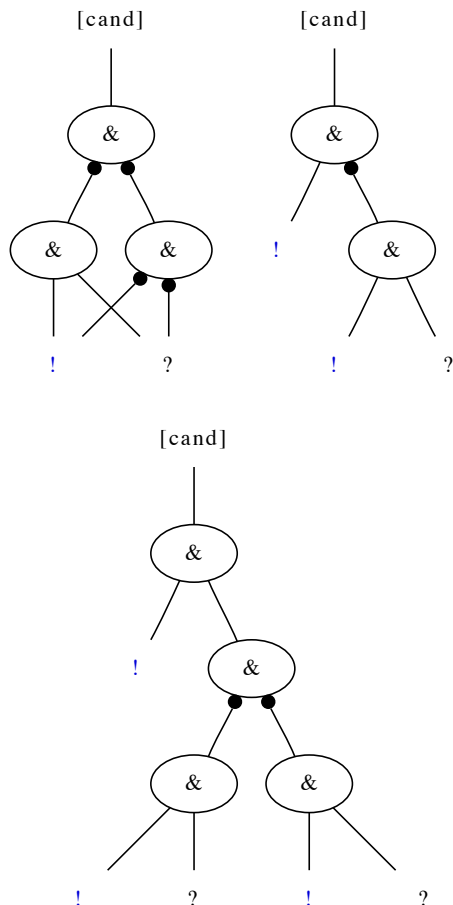
**Figure 1.** *Example of subgraphs that can be reduced.* Here "!" denote nodes we can control (a PI dominated by the output node); "?" denote nodes that can assume any value beyond our control. In the above examples, the output node "[cand]" can be forced to both a zero and a one by choosing the right values for the "!", regardless of the values of the "?". In such cases, the output node is replaced by a new PI.

the logic cone rooted in the given node while the cone is limited to 8 inputs. Limiting the scope to cones with 8 inputs and simulating 256 bit patterns (or eight 32-bit words) seems to be enough to saturate the reduction achievable on the benchmarks where the method is applicable.

Some typical reductions are shown in *Figure 1*. The graphs should be understood as sub-circuits of a netlist being reparameterized. The exclamation marks denote "internal" PIs dominated by the top-node, and hence under our control; and the question marks denote gates with fanouts outside the displayed logic cone, for which no assumption on their values can be made. The full algorithm is described in *Figure 2*.

**Counterexample reconstruction.** There are several ways that a trace on the reduced netlist can be lifted to the original netlist. For instance, the removed logic between the new PIs and the old PIs can be stored in a separate netlist. The trace on the reduced netlist can then be pro-

---

## Fast Reparameterization

– Compute dominators. For a DAG with bounded in-degree (such as an And-Inverter-Graph), this is a linear operation in the number of nodes (see *Figure 3*).

– For each PI, add all its dominators to the set of "candidates".

– For each candidate $c$, in topological order from inputs to outputs:

  - Compute the set $D$ of nodes dominated by $c$ (*Figure 4*).

  - Denote the PIs in $D$ "internal" inputs.

  - Denote any node outside $D$, but being a direct fanin of a node inside $D$, an "external" input (may be any gate type).

  - Simulate all possible assignments to the internal and external inputs. If *for all* external assignments *there exists* an internal assignment that gives a 0 at $c$, and *another* internal assignment that gives a 1 at $c$, then substitute $c$ by a new primary input.

**Figure 2.** *Steps of the reparameterization algorithm.*

---

## Compute Dominators

– Initialize all POs to dominate themselves

– Traverse the netlist in reverse topological order (from POs to PIs), and mark the children of each node as being dominated by the same dominator as yourself *unless* the child has already been assigned a dominator

– For already marked children, compute the "meet" of the two dominators, i.e. find the first common dominator. If there is no common dominator, mark the node as dominating itself.

**Figure 3.** *Review of the "finding direct dominators" algorithm for DAGs.* For a more complete treatment of this topic, see [8], or for a more precise description, the source code of "computeDominators()" in "ZZ/Netlist/StdLib.cc" of ABC-ZZ [11].

---

## Compute Dominated Area

```
area = {w_dom}          – init. set of gates to the dominator
count = [0, 0, …, 0]    – count is a map "gate → integer"
for w ∈ area:
      for v ∈ faninsOf(w):
            count[v]++
            if count[v] == num_of_fanouts[v]:
                  area = area ∪ {v}
```

**Figure 4.** *Find nodes used only by gate "w_dom".* This set is sometimes called *maximum fanout free cone* (MFFC). The outer for-loop over the elements of *area* is meant to visit all elements added by the inner for-loop. For this procedure, flops are treated as sinks, i.e. having no fanins.

jected onto the original PIs by rerunning the simulation used to produce the reparameterized circuit, and for each candidate pick an assignment that gives the correct value. But even simpler, one can just put the original netlist into a SAT solver and assert the values from the trace onto the appropriate variables and call the SAT solver to complete the assignment. In practice, this seems to always work well.

## 3   Improvements

The algorithm described in the previous section replaces internal nodes with inputs, and thus only removes logic. If we are prepared to forgo this admittedly nice property and occasionally add a bit of new logic, then nodes that are not completely controllable by their dominated inputs can still be reparameterized by the following method: for node with function $\phi(x, y)$, where $x$ are external inputs and $y$ are internal inputs, compute the following two functions:

$$\phi_0(x) \equiv \forall y. \neg \phi(x, y)$$
$$\phi_1(x) \equiv \forall y. \ \ \phi(x, y)$$

Using these two functions, $\phi$ can be resynthesized using a single input $y_{new}$ by the expression:

$$\neg \phi_0(x) \wedge (\phi_1(x) \vee y_{new})$$

In other words, if two or more inputs are dominated by the node $\phi$, a reduction in the number of inputs is guaranteed. Depending on the shape of the original logic, and how well new logic for $\phi_0$ and $\phi_1$ is synthesized, the number of logic gates may either increase or decrease. In our implementation, logic for $\phi_0$ and $\phi_1$ is created by the fast irredundant sum-of-product ("isop") proposed by Shin-ichi Minato in [10]. We greedily apply this extended method for all nodes with two or more dominated PIs, even if it leads to a blow-up in logic size. To counter such cases, fast logic synthesis can be applied after the reparameterization. Obviously, there are many ways to refine this scheme.

## 4   Future Work

Another possible improvement to the method is extending the reparameterization algorithm to work for multi-output cones. As an example, consider a two-output cone where the outputs can be forced to all four combinations {00, 01, 10, 11} by choosing appropriate values for dominated inputs. In such a case, the cone can be replaced by two free inputs. If some of the four combinations at the outputs are impossible under conditions expressed in terms of non-controllable signals, a logic cone can be constructed to characterize these conditions and reduce the number of PIs by adding logic similar to the case of a single-output cone.

## 5   Experiments

As part of the experimental evaluation, all benchmarks from the single-property track of the Hardware Modelcheck-

ing Competition 2012 were considered. Localization abstraction [5] was applied with a timeout of one hour to each benchmark and the resulting models meeting the following criteria were kept:

– At least half of the flops were removed by abstraction.

– The abstraction was accurate (no spurious counterexamples).

– At least one of the verification engines could prove the property within one hour.

The sizes of benchmarks selected in this way are listed in table *Table 1*. All those models were given to the reparameterization engine, both in *weak* mode and *strong* mode, the latter using the improvements described in section 3. The reparameterized models were also post-processed with a quick simplification method called "shrink" which is part of the ABC package [7]. The longest runtime for weak reparameterization was 16 ms, for strong reparameterization 28 ms and for the simplification phase 50 ms.[2] Reductions are listed in table *Table 2*.

For comparison, *Table 2* also include the results of running an industrial implementation of the BDD based algorithm of [1]. Because runtimes are significantly longer with this algorithm, they are given their own column. These results were given to us from IBM, and according to their statement "are not tweaked as much as they could be".

All benchmarks were given to three engines: *Property Directed Reachability* [2, 6], *BDD-based reachability* [3], and *Interpolation-based Model Checking* [9]. The complete table of results is given in *Table 3*. A slice of this table, showing only results for PDR, with and without (strong) reparameterization, is given in *Table 4* together with a scatter plot.

**Analysis.**   Firstly, we see a speedup of 100x-1000x over previous work in the runtime of the reparameterization algorithm itself, with comparable quality of results for the application under consideration (models resulting from localization abstraction). This means the algorithm can *always* be applied without the need for careful orchestration. Secondly, we see an average speedup of 2.5x in verification times when applying reparameterization in conjunction with PDR, which is also the best overall engine on these examples. For two benchmarks, *6s121* and *6s150*, BDD reachability do substantially better than PDR, and for the latter (where runtimes are meaningful) the speedup due to reparameterization is greater than 3x. Furthermore, for BDD reachability one can see that on several occasions (*6s30* in particular), reparameterization is completely crucial for performance. Finally, interpolation based modelchecking (IMC) seems to be largely unaffected by reparameterization.

---

[2]Benchmarks from HWMCC'12 are quite small. For comparison: running reparameterization on a 7 million gate design from one of our industrial collaborators took 4.1 s.

**Abstraction Phase**

| Design | #And | #PI | #FF | | | Depth |
|--------|------|-----|-----|---|------|-------|
| *6s102* | 6,594 | 72 | 1,121 | → | 56 | 23 |
| *6s121* | 1,636 | 99 | 419 | → | 110 | 19 |
| *6s132* | 1,216 | 94 | 139 | → | 113 | 7 |
| *6s144* | 41,862 | 480 | 3,337 | → | 236 | 18 |
| *6s150* | 5,448 | 146 | 1,044 | → | 323 | 103 |
| *6s159* | 1,469 | 13 | 252 | → | 18 | 4 |
| *6s164* | 1,095 | 91 | 198 | → | 93 | 16 |
| *6s189* | 36,851 | 479 | 2,434 | → | 259 | 18 |
| *6s194* | 12,049 | 532 | 2,389 | → | 198 | 45 |
| *6s30* | 1,043,139 | 32,994 | 1,195 | → | 128 | 32 |
| *6s43* | 7,408 | 30 | 965 | → | 310 | 25 |
| *6s50* | 16,700 | 1,570 | 3,107 | → | 207 | 52 |
| *6s51* | 16,701 | 1,570 | 3,107 | → | 209 | 65 |
| *bob05* | 18,043 | 224 | 2,404 | → | 146 | 106 |
| *bob1u05cu* | 32,063 | 224 | 4,377 | → | 146 | 106 |

**Table 1.** *Sizes of original designs and abstract models.* Column #FF shows how many flip-flops were turned into unconstrained inputs by localization abstraction. The removed FFs show up as PIs in the other tables. Last column shows the BMC depth used by the abstraction engine (see [5] for more details).

**Reparameterization**

| Design | No Reparam. | | Weak Rep. | | Strong Rep. | | BDD Reparam. | | |
|--------|------|-----|------|-----|------|-----|------|-----|---------|
| | #And | #PI | #And | #PI | #And | #PI | #And | #PI | Runtime |
| *6s102* | 6,594 | 1,137 | 1,247 | 331 | 1,188 | 267 | 1,283 | 285 | 71.91 s |
| *6s121* | 1,636 | 408 | 627 | 120 | 559 | 66 | 732 | 41 | 0.27 s |
| *6s132* | 1,216 | 120 | 1,108 | 62 | 1,102 | 55 | 2,731 | 36 | 0.80 s |
| *6s144* | 41,862 | 3,580 | 10,172 | 1,038 | 9,494 | 812 | 13,259 | 889 | 60.50 s |
| *6s150* | 5,448 | 867 | 3,062 | 506 | 2,213 | 89 | 2,231 | 46 | 1.81 s |
| *6s159* | 1,469 | 247 | 116 | 20 | 114 | 19 | 256 | 13 | 0.02 s |
| *6s164* | 1,077 | 196 | 661 | 109 | 499 | 41 | 3,413 | 40 | 11.61 s |
| *6s189* | 36,851 | 2,654 | 10,033 | 1,004 | 9,552 | 794 | 12,051 | 814 | 63.11 s |
| *6s194* | 12,049 | 2,723 | 1,366 | 184 | 1,348 | 166 | 1,612 | 121 | 10.37 s |
| *6s30* | 102,535 | 34,061 | 1,508 | 307 | 1,184 | 205 | 603 | 50 | 0.74 s |
| *6s43* | 7,408 | 685 | 3,451 | 304 | 3,218 | 202 | 17,691 | 101 | 2.07 s |
| *6s50* | 16,700 | 4,470 | 1,841 | 350 | 1,652 | 270 | 4,319 | 752 | 46.57 s |
| *6s51* | 16,701 | 4,468 | 1,828 | 350 | 1,639 | 268 | 1,255 | 62 | 16.34 s |
| *bob05* | 18,043 | 2,358 | 1,618 | 187 | 1,388 | 52 | 1,586 | 43 | 0.31 s |
| *bob1u05cu* | 32,063 | 4,455 | 1,618 | 187 | 1,388 | 52 | 1,586 | 43 | 0.33 s |

**Table 2.** *Effect of reparameterization on the abstracted models.* "Weak" reparameterization refers to the basic method described in section 2, "Strong" additionally includes the improvements discussed in section 3. Runtimes are omitted as the average CPU time was 4-8 ms (and the longest 28 ms). However, BDD based reparameterization is not as scalable as the method presented in this paper, and runtimes (typically between 100x-1000x longer) are listed for reference.

# References

[1] J. Baumgartner and H. Mony. **Maximal Input Reduction of Sequential Netlists via Synergistic Reparameterization and Localization Strategies**. In *Proc. of CHARME*, pages 222–237, 2005.

[2] Aaron Bradley. **IC3: SAT-Based Model Checking Without Unrolling**. In *Proc. of VMCAI*, 2011.

[3] R. E. Bryant. **Graph-Based Algorithms for Boolean Function Manipulation**. In *IEEE Transactions on Computers, vol. c-35, no.8, Aug.*, 1986.

[4] Pankaj Chauhan, Edmund Clarke, and Daniel Kroening. **A SAT-Based Algorithm for Reparameterization in Symbolic Simulation**. In *Proc. of DAC*, 2004.

[5] N. Een, A. Mishchenko, and N. Amla. **A Single-Instance Incremental SAT Formulation of Proof- and Counterexample-Based Abstraction**. In *FMCAD*, 2010.

[6] Niklas Een, Alan Mishchenko, and Robert Brayton. **Efficient Implementation of Property Directed Reachability**. In *Proc. of FMCAD*, 2011.

| | PDR | | | BDD reach. | | | IMC | | |
|---|---|---|---|---|---|---|---|---|---|
| | **NoRep.** | **Weak** | **Strong** | **NoRep.** | **Weak** | **Strong** | **NoRep.** | **Weak** | **Strong** |
| *6s102* | 1.7 | 0.4 | 0.4 | 121.2 | 90.2 | 204.3 | – | 2619.2 | – |
| *6s121* | 64.0 | 12.3 | 5.4 | 0.5 | 0.3 | 0.4 | 10.2 | 5.8 | 36.0 |
| *6s132* | 7.8 | 8.1 | 7.9 | 2327.3 | 1375.5 | – | 201.4 | 384.9 | 267.9 |
| *6s144* | 10.3 | 12.4 | 9.6 | – | – | – | 1177.5 | 942.1 | 1413.0 |
| *6s150* | – | 2323.7 | – | 539.3 | 143.6 | 189.1 | – | – | – |
| *6s159* | 0.0 | 0.0 | 0.0 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 |
| *6s164* | 7.0 | 34.1 | 6.8 | – | 33.6 | 0.4 | 4.4 | 8.5 | 3.0 |
| *6s189* | 11.9 | 7.8 | 8.6 | – | – | – | 738.0 | 396.5 | 366.0 |
| *6s194* | 4.7 | 4.6 | 3.7 | 307.5 | 42.5 | 742.2 | 798.9 | 1042.8 | 1103.3 |
| *6s30* | 15.9 | 45.3 | 12.3 | – | 17.6 | 49.2 | – | – | – |
| *6s43* | 13.6 | 11.7 | 9.1 | – | 1191.1 | 1390.4 | – | – | – |
| *6s50* | 14.7 | 10.8 | 5.4 | 941.6 | 241.0 | 1680.8 | 1795.6 | 820.0 | 2348.9 |
| *6s51* | 18.7 | 7.1 | 4.1 | 303.8 | 147.6 | 1891.2 | 1806.8 | 954.8 | 1737.8 |
| *bob05* | 0.3 | 0.3 | 0.3 | 2450.5 | 2371.4 | 1253.2 | 30.3 | 26.3 | 25.1 |
| *bob1u05cu* | 0.3 | 0.2 | 0.3 | – | 2157.6 | 1088.3 | 30.3 | 24.0 | 24.8 |

**Table 3.** *Full table of results.* Each design after abstraction is given to three engines: Property Directed Reachability (PDR), BDD-based reachability (BDD), and Interpolation-based Model Checking (IMC). Each engine is run on three versions of the model with no/weak/strong reparameterization applied. A dash represents a timeout after one hour.

| **Design** | **NoRep.** | **Rep.** |
|---|---|---|
| 6s102 | 1.66 | **0.40** |
| 6s121 | 64.00 | **5.43** |
| 6s132 | **7.82** | 7.90 |
| 6s144 | 10.34 | **9.56** |
| 6s159 | 0.04 | **0.03** |
| 6s164 | 6.97 | **6.83** |
| 6s189 | 11.86 | **8.59** |
| 6s194 | 4.71 | **3.72** |
| 6s30 | 15.92 | **12.29** |
| 6s43 | 13.63 | **9.15** |
| 6s50 | 14.66 | **5.44** |
| 6s51 | 18.72 | **4.08** |
| bob05 | 0.33 | **0.26** |
| bob1u05cu | 0.33 | **0.26** |
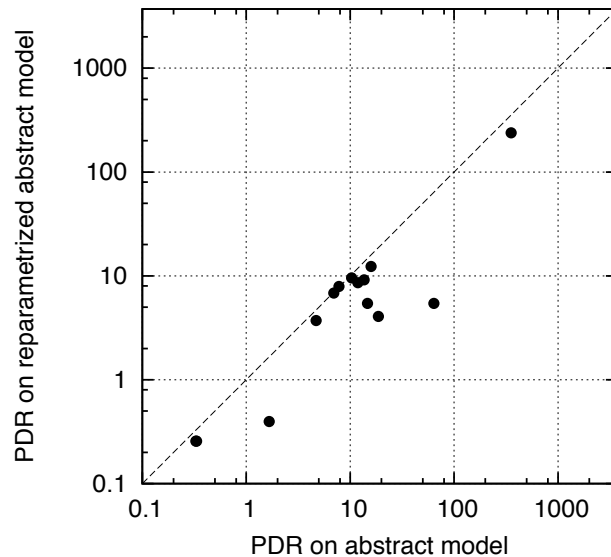


**Table 4.** *Runtime improvements for PDR.* Column "NoRep." shows runtimes (in seconds) for proving the property of each benchmark using PDR after abstraction, but without reparameterization; column "Rep." shows runtimes after reparameterization. The scatter plot on the right places these runtime pairs on a log-scale. The average speedup is 2.5x.

[7] Berkeley Logic Synthesis Group. **ABC: A System for Sequential Synthesis and Verification**. *http://www.eecs.berkeley.edu/~alanmi/abc/*, v00127p.

[8] T. Lengauer and R.E. Tarjan. **A Fast Algorithm for Finding Dominators in a Flowgraph**. In *ACM Transactions on Programming Languages and Systems, Vol. 1, No. 1, July*, pages 121–141, 1979.

[9] K. L McMillan. **Interpolation and SAT-based Model Checking**. In *Proc. of CAV*, 2003.

[10] S. Minato. **Fast Generation of Irredundant Sum-Of-Products Forms from Binary Decision Diagrams**. In *Proc. of SASIMI*, 1992.

[11] Berkeley Verification and Synthesis Research Center. **ABC-ZZ: A C++ framework for verification & synthesis**. *https://bitbucket.org/niklaseen/abc-zz*.