An Abstraction Algorithm for the Verification of Generalized C-Slow Designs

Jason Baumgartner[1], Anson Tripp[1], Adnan Aziz[2], Vigyan Singhal[3], and Flemming Andersen[1]

[1] IBM Corporation, Austin, Texas 78758, USA,
{jasonb,ajt,fanders}@austin.ibm.com
[2] The University of Texas, Austin, Texas 78712, USA,
adnan@ece.utexas.edu
[3] Tempus Fugit, Inc., Albany, California 94706, USA
vigyan@home.com

Abstract. A c-slow netlist N is one which may be retimed to another netlist N', where the number of latches along each wire of N' is a multiple of c. Leiserson and Saxe [1, page 54] have shown that by increasing c (a process termed *slowdown*) and retiming, any design may be made systolic, thereby dramatically decreasing its cycle time. In this paper we develop a new fully-automated abstraction algorithm applicable to the verification of generalized c-slow flip-flop based netlists; the more generalized topology accepted by our approach allows applicability to a fairly large class of pipelined netlists. This abstraction reduces the number of state variables and divides the diameter of the model by c; intuitively, it folds the state space of the design modulo c. We study the reachable state space of both the original and reduced netlists, and establish a c-slow bisimulation relation between the two. We demonstrate how CTL* model checking may be preserved through the abstraction for a useful fragment of CTL* formulae. Experiments with two components of IBM's Gigahertz Processor demonstrate the effectiveness of this abstraction algorithm.

1 Introduction

Leiserson and Saxe [1,2] have defined a c-slow netlist N as one which is retiming equivalent to another netlist N', where the number of latches along each wire of N' is a multiple of c. Netlist N' may be viewed as having c equivalence classes of latches; latches in class i may only fan out to latches in class $(i+1)$ mod c. Each equivalence class of latches of N' contains data from an independent stream of execution, and data from two or more independent streams may never arrive at any netlist element concurrently. They demonstrate that designs may be made systolic through *slowdown* (increasing c), and how this process dramatically benefits the cycle time of such designs. We have observed at IBM a widespread use of such pipelining through slowdown, for example, in control logic which routes tokens to and from table-based logic (e.g., caches and instruction dispatch logic).

This use has been hand-crafted during the design development, not achieved via a synthesis tool.

The purpose of our research is to develop a sound and complete abstraction algorithm for the verification of a generalized class of flip-flop (FF) based c-slow designs, which eliminates all but one equivalence class of FFs and reduces the diameter of the model by a factor of c. To the best of our knowledge, this paper is the first to exploit the structure of c-slow designs for enhanced verification. Our motivating example was an intricate five-stage pipelined control netlist with feedback, and with an asynchronous interrupt to every stage. This interrupt input prevents the classification of this design as c-slow by the definition in [2], but our generalization of that definition enables us to classify and perform a c-slow abstraction upon this example. Due to its complexity and size, model checking this netlist required enormous computational complexity even after considerable manual abstraction.

Retiming itself is insufficient to achieve the results of c-slow abstraction. Retiming is not guaranteed to reduce the diameter of the model, and does not change the number of latches along a directed cycle. It should also be noted that a good BDD ordering cannot achieve the benefits of this abstraction. For example, assume that we have two netlists, N and N', where N is equivalent to N' except that the FF count along each wire of N' is multiplied by c. One may hypothesize that a BDD ordering which groups the FFs along each wire together may bound the BDD size for the reachable set of N to within a factor of c of that of N'; however, we have found counterexamples to this hypothesis. A better ordering groups all variables of a given class together, as the design could be viewed as comprising c independent machines in parallel. However, such ordering will not reduce the number of state variables nor the diameter of the model.

A related class of research has considered the transformation of level-sensitive latch-based netlists to simpler edge-sensitive FF-based ones. (Refer to [3] for a behavioral definition of these latch types.) Hasteer et al. [4] have shown that multi-phase netlists may be "phase abstracted" to simpler FF-based netlists for sequential hardware equivalence. Baumgartner et al. [5] have taken a similar approach (called "dual-phase abstraction" for a 2-phase design) for model checking. This approach preserves initial values and provides greater reduction in the number of state variables than the phase abstraction from [4] alone. Phase abstraction is fundamentally different than c-slow abstraction. For example, in multi-phase designs, only one class of latches updates at each time-step. Furthermore, the initial values of all but one class of latches will be overwritten before propagation. Therefore, given a FF-based design, these approaches are of no further benefit.

The remainder of the document is organized as follows. Section 2 provides our definition of c-slow netlists, and introduces a 3-slow netlist N. In Section 3 we introduce our abstraction algorithm, as well as the abstracted version of N. We demonstrate the correctness of the abstraction in Section 4. In particular we demonstrate a natural correspondence between the original and abstracted designs which we refer to as a c-slow bisimulation, and discuss the effect of the abstraction upon CTL* model checking. In Section 5 we introduce algorithms to determine the maximum c and to translate traces from the abstracted model to traces in the original netlist. We provide experimental results in Section 6, and in Section 7 we summarize and present future work items.

2 C-Slow Netlists

In this section we provide our definition of c-slow netlists, which is a more general definition than that of [2]. We assume that the netlist contains no level-sensitive latches; if it does, phase abstraction [5] should be performed to yield a FF-based netlist. We further assume that the netlist has no gated-clock FFs; if it does, it is at most 1-slow (since an inactive gate mandates that the next state of the FF be equivalent to its present state). Each c-slow netlist N is comprised of c equivalence classes of FFs characterized as follows.

Definition 1. A *c-slow netlist* is one whose gates and FFs may be c-colored such that:

1. Each FF is assigned a color i.
2. All FFs which have FFs of color i in their support have color $(i + 1)$ mod c.
3. All gates which have FFs of color i in their support have color i.

There are several noteworthy points in the above definition. First, since no gate may contain FFs of more than one color in its support, the design may not reason about itself except "modulo c". Intuitively, it is this property which allows us to "fold" the design to a smaller domain of a single coloring of FFs.

Second, the coloring restrictions apply only to FFs and to gates which have FFs in their support. Hence, it is legal for primary inputs to fanout to FFs of multiple colors, which makes our definition more general than that in [2]. This generality has shown great potential in extending the class of netlists to which we may apply the c-slow abstraction; the two design fragments of the Gigahertz Processor from our sample set which were found to be c-slow would not have been classifiable as such without this generality.

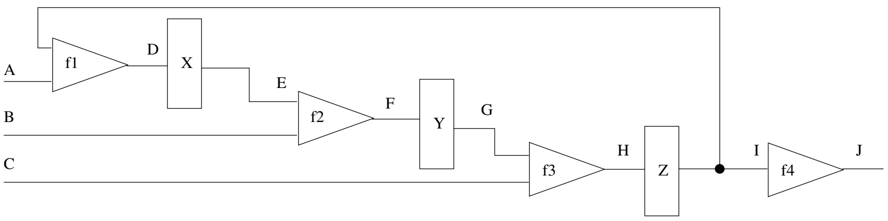

Fig. 1. 3-Stage Netlist N

Consider the generic 3-slow netlist N depicted in Figure 1. We assume that this netlist represents a composition of the design under test and its environment. We also assume that no single input will fan out to FFs of different colors. Later in this section we demonstrate that we may soundly alter netlists which violate this assumption by splitting such inputs into functionally identical yet distinct cones, one per color. All nets may be vectors. We arbitrarily define the color of FFs X as 0, of Y as 1, and of Z as 2.

Consider the first several time-steps of symbolic reachability analysis of N in Table 1 below. The symbolic values a_i, b_i, and c_i represent arbitrary values producible at inputs A, B, and C respectively. The symbol i_j represents the initial value of the FFs of color j.

time	0	1	2	3	4	5
A	a_0	a_1	a_2	a_3	a_4	a_5
B	b_0	b_1	b_2	b_3	b_4	b_5
C	c_0	c_1	c_2	c_3	c_4	c_5
D	$f1(a_0, i_2)$	t_{1_1}	t_{0_2}	$f1(a_3, t_{2_2})$	t_{1_3}	$f1(a_5, t_{0_3})$
E	i_0	$f1(a_0, i_2)$	t_{1_1}	t_{0_2}	$f1(a_3, t_{2_2})$	t_{1_3}
F	$f2(b_0, i_0)$	t_{2_1}	t_{1_2}	$f2(b_3, t_{0_2})$	t_{2_3}	$f2(b_5, t_{1_3})$
G	i_1	$f2(b_0, i_0)$	t_{2_1}	t_{1_2}	$f2(b_3, t_{0_2})$	t_{2_3}
H	$f3(c_0, i_1)$	t_{0_1}	t_{2_2}	$f3\big(c_3, t_{1_2}\big)$	t_{0_3}	$f3(c_5, t_{2_3})$
I	i_2	$f3(c_0, i_1)$	t_{0_1}	t_{2_2}	$f3(c_3, t_{1_2})$	t_{0_3}
J	$f4(i_2)$	$f4\big(f3(c_0, i_1)\big)$	$f4(t_{0_1})$	$f4(t_{2_2})$	$f4\big(f3(c_3, t_{1_2})\big)$	$f4(t_{0_3})$

$$t_{0_1} = f3\big(c_1, f2(b_0, i_0)\big) \qquad t_{0_2} = f1(a_2, t_{0_1}) \qquad t_{0_3} = f3\big(c_4, f2(b_3, t_{0_2})\big)$$
$$t_{1_1} = f1\big(a_1, f3(c_0, i_1)\big) \qquad t_{1_2} = f2(b_2, t_{1_1}) \qquad t_{1_3} = f1\big(a_4, f3(c_3, t_{1_2})\big)$$
$$t_{2_1} = f2\big(b_1, f1(a_0, i_2)\big) \qquad t_{2_2} = f3(c_2, t_{2_1}) \qquad t_{2_3} = f2\big(b_4, f1(a_3, t_{2_2})\big)$$

Table 1. Reachability Analysis of N

At any instant in time, each signal may only be a function of the value generated by a given "data source" at every c-th cycle. By data source, we refer to inputs and initial values. This observation illustrates the motivation behind this abstraction; our transformation yields a design where each signal may concurrently (nondeterministically) be a function of each data source at each cycle.

3 Abstraction of C-Slow Netlists

In this section we illustrate the structural modifications necessary and sufficient to perform the c-slow abstraction.

The algorithm for performing the abstraction is as follows. Color $c-1$ FFs are abstracted by replacement with a mux selected by a conjunction of a random value with *first_cycle*; the output of this mux passes through a FF, and the initial value of this FF is defined by the value which would appear at its input if *first_cycle* were asserted. Color 0 FFs are abstracted by replacement with a mux which is selected by *first_cycle*. All others are abstracted by replacement with a mux which is selected by a conjunction of a random value with *first_cycle*. If the netlist is a feed-forward pipeline, all variables may be replaced in this last manner, thereby converting the sequential netlist to a combinational one.

Consider netlist N' shown in Figure 2 below, which represents our c-slow abstraction of N. We initialize Z' with the value which would appear at its inputs if *first_cycle* were 1. The new inputs *ND* and *ND'* represent unique nondeterministic values. We will utilize one copy of this netlist (with *first_cycle* tied up) to generate the initial values for Z'. This initial value will be utilized in a second copy of this netlist where *first_cycle* is tied down, which is the copy that we will model check. Intuitively, the nondeterministic initial value allows the output of the rightmost mux to take all possible values observable at net I in N during the first c cycles. Thereafter, straightforward reachability analysis will ensure correspondence of N and N'.

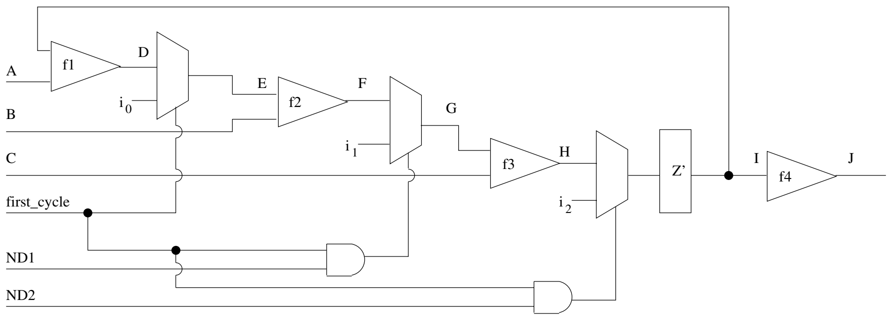

Fig. 2. Abstracted 3-Stage Netlist N'

The first two cycles of symbolic simulation of N' is provided in Table 2. To compensate for the "nondeterministic initial state set" induced by this abstraction, we have split this analysis into three *timeframes*; cycles 0_0 and 1_0 represent the timeframe induced by the first element of the set – that element further being a function only of the initial values of the FFs of color 0, cycles 0_1 and 1_1 induced by the second element, etc. This analysis illustrates the manner in which the design will be treated by the reachability engine.

There are several critical points illustrated by this table. First, the values present at FFs of color $c - 1$ in N', at time i along the path induced by initial values of color j, are equivalent to those present upon their correspondents in N at time $3i + j$. This may be inductively expanded by noting that i_j in the above two tables need not correlate to initial values, but to any reachable state.

time	0_0	1_0	0_1	1_1	0_2	1_2
A	a_2	a_5	a_1	a_4	a_0	a_3
B	b_3	b_6	b_2	b_5	b_1	b_4
C	c_4	c_7	c_3	c_6	c_2	c_5
D	t_{0_2}	$f1(a_5, t_{0_3})$	t_{1_1}	t_{1_3}	$f1(a_0, i_2)$	$f1(a_3, t_{2_2})$
E	t_{0_2}	$f1(a_5, t_{0_3})$	t_{1_1}	t_{1_3}	$f1(a_0, i_2)$	$f1(a_3, t_{2_2})$
F	$f2\big(b_3, t_{0_2})\big)$	t_{0_4}	t_{1_2}	$f2(b_5, t_{1_3})$	t_{2_1}	t_{2_3}
G	$f2\big(b_3, t_{0_2})\big)$	t_{0_4}	t_{1_2}	$f2(b_5, t_{1_3})$	t_{2_1}	t_{2_3}
H	t_{0_3}	$f3(c_7, t_{0_4})$	$f3\big(c_3, t_{1_2}\big)$	$f3\big(c_6, f2(b_5, t_{1_3})\big)$	t_{2_2}	$f3(c_5, t_{2_3})$
I	t_{0_1}	t_{0_3}	$f3(c_0, i_1)$	$f3\big(c_3, t_{1_2}\big)$	i_2	t_{2_2}
J	$f4(t_{0_1})$	$f4(t_{0_3})$	$f4\big(f3(c_0, i_1)\big)$	$f4\big(f3\big(c_3, t_{1_2}\big)\big)$	$f4(i_2)$	$f4(t_{2_2})$

$$t_{0_4} = f2\big(b_6, f1(a_5, t_{0_3})\big)$$

Table 2. Reachability Analysis of N'

Another point is that, in order to "align" the values present at FFs of color $c-1$, we have been forced to temporally skew the inputs. For example, comparing any state at time i in N' with state $3i + j$ in N, we see that the inputs which fan out to all but color 0 FFs are skewed forward by an amount equal to the color of the FFs in their transitive fanout. While this may seem bizarre, recall our assumption that the netlist is a composition of the environment and design. Therefore, without loss of generality, the inputs may only be combinational free variables or constants. In either case, the values they may hold at any times i and j are equivalent. For each value that input vector A may take at time i (denoted A_i), there is an equivalent value that A may take at any time j, and vice-versa. This fact is crucial to the correctness of our abstraction.

To further demonstrate this point, we now discuss how we handle inputs which fan out to FFs of multiple colors. We split such "multi-color" input logic cones into a separate cone per color. To illustrate the necessity of this technique, assume that inputs A and B were tied together in N – a single input AB. Thus, values a_i and b_i are identical. Tying these together in N' would violate the correspondence; for example, state t_{3_2} is a function of a_3 and b_4, each being a distinct value producible by input AB. Clearly any state producible in N', where a_3 and b_4 are equivalent, would be producible in N. However, any state in N where a_3 and b_4 differ would be unproducible in N', *unless* we split AB into two functionally equivalent, yet distinct, inputs – one for color 0 and the other for color 1. Such splitting of shared input cones is straightforward, and may be performed automatically in $O(c \cdot nets)$ time. This logic splitting may encompass combinational logical elements, and even logic subcircuits including FFs which themselves are c-slow. The exact theory of such splitting of sequential cones is still under development, though may be visualized by noting that peripheral latches upon multi-color inputs clearly need not limit c.

4 Correctness of Abstraction

In this section we define our notion of correspondence between the original and abstracted netlists, which we term a c-slow bisimulation (inspired by Milner's bisimulation relations [6]). We will relate our designs to Kripke structures, which are defined as follows.

Definition 2. A *Kripke structure* $\mathcal{K} = \langle S, S_0, \mathcal{A}, \mathcal{L}, R\rangle$, where S is a set of *states*, $S_0 \subseteq S$ is the set of *initial states*, $\mathcal{A}$ is the set of *atomic propositions*, $\mathcal{L} : S \mapsto 2^{\mathcal{A}}$ is the *labeling function*, and $R \subseteq S \times S$ is the *transition relation*.

Our designs are described as Moore machines (using Moore machines, instead of the more general Mealy machines [7], simplifies the exposition for this paper, though our implementation is able to handle Mealy machines by treating outputs as FFs). We use the following definitions for a Moore machine and its associated structure (similar to Grumberg and Long [8]).

Definition 3. A *Moore machine* $\mathcal{M} = \langle L, S, S_0, I, O, V, \delta, \gamma\rangle$, where L is the set of *state variables (FFs)*, $S = 2^L$ is the set of *states*, $S_0 \subseteq S$ is the set of *initial states*, I is the set of *input variables*, O is the set of *output variables*, $V \subseteq L$ is the set of *property visible FFs*, $\delta \subseteq S \times 2^I \times S$ is the *transition relation*, and $\gamma : S \mapsto 2^O$ is the *output function*.

We uniquely identify each state by a subset of the state variables; intuitively, this subset represents those FFs which evaluate to a 1 at that state. We will define V to be the FFs of color $c-1$.

Definition 4. The *Kripke structure associated with a Moore machine* $\mathcal{M} = \langle L, S, S_0, I, O, V, \delta, \gamma\rangle$ is denoted by $K(M) = \langle S^K, S_0^K, \mathcal{A}, \mathcal{L}, R\rangle$, where $S^K = 2^{L \cup I}$, $S_0^K = \{s \in S^K : s - I \in S_0\}$, $\mathcal{A} = V$, $\mathcal{L} = S^K \mapsto 2^V$, and $R\big((s,x),(t,y)\big)$ iff $\delta(s,x,t)$.

Intuitively, we define S_0^K as the subset of S^K which, when projected down to the FFs, is equivalent to S_0. In the sequel we will use M to denote the Moore machine as well as the Kripke structure for the machine.

Definition 5. A *c-transition* of a Kripke structure M is a sequence of c transitions from state s_i to s_{i+c}, where $R(s_j, s_{j+1})$ for all j such that $0 \leq j < c$.

Definition 6. Let M be the Kripke structure of a c-slow machine. We define the *extended initial state set* of M, $S_{init} \subseteq S$, as the set of all states reachable within $c-1$ time-steps from the initial state of M.

Definition 7. Let M and M' be two Kripke structures. A relation $G \subseteq S \times S'$ is a *c-slow-bisimulation* relation if $G(s, s')$ implies:

1. $\mathcal{L}(s) = \mathcal{L}'(s')$.

2. for every $t \in S$ such that there exists a $c - transition$ of M from state s to t, there exists $t' \in S'$ such that $R'(s', t')$ and $G(t, t')$.
3. for every $t' \in S'$ such that $R'(s', t')$, there exists $t \in S$ such that there exists a $c - transition$ of M from state s to t, and $G(t, t')$.

We say that a c-slow-bisimulation exists from M to M' (denoted by $M \prec M'$) iff there exists a c-slow bisimulation relation G such that for all $s \in S_{init}$ and $t' \in S'_0$, there exist $s' \in S'_0$ and $t \in S_{init}$ such that $G(s, s')$ and $G(t, t')$.

An infinite path $\pi = (s_0, s_1, s_2, \ldots)$ is a sequence of states such that any two successive states are related by the transition relation (i.e., $R(s_i, s_{i+1})$). Let π^i denote the suffix path $(s_i, s_{i+1}, s_{i+2}, \ldots)$. We say that a c-slow-bisimulation relation exists between two infinite paths $\pi = (s_0, s_1, s_2, \ldots)$ and $\pi' = (s'_0, s'_1, s'_2, \ldots)$, denoted by $G(\pi, \pi')$, iff for all $i \geq 0$, we have that $G(s_{c \cdot i}, s'_i)$.

Lemma 1. *Let s and s' be states of structures M and M', respectively, such that $G(s, s')$. For each infinite path π starting at s and composed of transitions of M, there exists an infinite path π' starting at s' and composed of transitions of M' such that $G(\pi, \pi')$. Similarly, for each infinite path π' starting at s' and composed of transitions of M', there exists an infinite path π starting at s and composed of transitions of M such that $G(\pi, \pi')$.*

The bisimilarity ensures that the reachable set of the original and abstracted netlists will be equivalent. Thus, properties such as $AG\phi$ and $EF\phi$, where ϕ is a boolean property, are trivially preserved using this abstraction, provided that all formula signals refer to nets of the same color. This may seem limiting: a simple formula such as $AG(latchIn \rightarrow AX\big(latchOut)\big)$ reasons about two differently colored nets. However, such a formula may be captured by an automaton which transitions upon $latchIn \equiv 1$ to a state where it samples $latchOut$; the new "single-colored" formula asserts that this sampled value $\equiv 1$. Note that such transformations of CTL to automata are commonplace for *on-the-fly* model checking [9].

One approach to transforming properties for this abstraction is to synthesize the original property (for the unabstracted design), and compose it into the model prior to abstraction. The structural abstraction will thereby also abstract the property. Logic synthesis algorithms may need to be tuned for optimal use of the c-slow abstraction. For example, $p \ \rightarrow \ AXAXAXq$ may likely be synthesized as a counter (counting the number of cycles which have occurred since p). However, such a direct translation would violate the c-slow topology of the design. We therefore propose a pipelined "one-hot" translation which would, for example, specifically introduce N state variables rather than $\lceil log_2(N) \rceil$ for AX^N, if it is determined that the design has a c-slow topology. Translation of design substructures which violate c-slowness, but may be safely replaced with substructures which do not, is an important topic which is not limited to property automata. We did not implement such a "c-slow-friendly translator" for our experimentation, as our properties were relatively simple and in many cases implemented directly via automata anyway, though we feel that one could be readily automated.

A second approach to abstracting properties is by a dedicated transformation algorithm. Both approaches place substantial constraints upon the fragment of CTL* that our abstraction may handle, as is reflected by the following definition.

Definition 8. A c-slow reducible (CSR) subformula ϕ and its c-slow reduction $\Omega(\phi)$ are defined inductively as follows.

– every atomic proposition p is a CSR state formula $\phi = p$, and $\Omega(\phi) = p$. (Note that these atomic propositions are limited to the property-visible nets V, which are nets of color $c-1$.)
– if p is a CSR state formula, so is $\phi = \neg p$, and $\Omega(\phi) = \neg\Omega(p)$.
– if p and q are CSR state formulae, so is $\phi = p \wedge q$, and $\Omega(\phi) = \Omega(p) \wedge \Omega(q)$.
– if p is a CSR path formula, then $\phi = \mathsf{E}p$ is a CSR state formula, and $\Omega(\phi) = \mathsf{E}\Omega(p)$.
– if p is a CSR path formula, then $\phi = \mathsf{A}p$ is a CSR state formula, and $\Omega(\phi) = \mathsf{A}\Omega(p)$.
– each CSR state formula ϕ is also a CSR path formula ϕ.
– if p is a CSR path formula, so is $\phi = \neg p$, and $\Omega(\phi) = \neg\Omega(p)$.
– if p and q are CSR path formulae, so is $\phi = p \wedge q$, and $\Omega(\phi) = \Omega(p) \wedge \Omega(q)$.
– if p is a CSR path formula, so is $\phi = \mathsf{X}^{\mathsf{c}}p$, and $\Omega(\phi) = \mathsf{X}\Omega(p)$. (Note that strings of less than c X operators must be flattened via translation to automata.)

If p is a CSR subformula, then $\phi = AG\ p$ and $\phi = EF\ p$ are CSR state formulae, for which $\Omega(\phi) = AG\ \Omega(p)$ or $\Omega(\phi) = EF\ \Omega(p)$, respectively. These last transformations are not recursively applicable; EF and AG are only applicable as the first tokens in the formula. Furthermore, CSR subformulae are themselves insufficient for model checking using this abstraction; the top-level quantification is necessary to break the dependence on the initial state of the concrete model.

Note that $pU\ q$ is not a CSR formula, since it entails reasoning across consecutive time-steps. However, since the design may only reason about itself modulo c, we have not found it beneficial to use such a property to verify the design. Furthermore, we have found it useful to use a *modulo-c* variant of the U operator for such designs. We define $pU_c\ q$ as true along a path $(s_i, s_{i+1}, ...)$ iff there exists a j such that $s_j \models q$, and for all states at index k where $(j \bmod c) \equiv (k \bmod c)$ and $k < j$, we have that $s_j \models p$. Similar approaches apply to the general use of G and F.

It is noteworthy that every property which we had verified of the designs reported in our experimental results was suitable for c-slow abstraction. As per the above discussion, we suspect that all meaningful properties of such netlists will either be directly suitable for c-slow abstraction, or may be strengthened to be made suitable.

Theorem 1. *Let s and s' be states of M and M', and $\pi = (s_0, s_1, s_2, \ldots)$ and $\pi' = (s'_0, s'_1, s'_2, \ldots)$ be infinite paths of M and M', respectively. If G is a c-slow-bisimulation relation such that $G(s, s')$ and $G(\pi, \pi')$, then*

1. *for every c-slow-reducible CTL* state formula ϕ, $s \models \phi$ iff $s' \models \Omega(\phi)$.*
2. *for every c-slow-reducible CTL* path formula ϕ, $\pi \models \phi$ iff $\pi' \models \Omega(\phi)$.*

Proof. The proof is by induction on the length of the formula ϕ. Our induction hypothesis is that Theorem 1 holds for all CTL* formulae ϕ' of length $\leq n$.

Base Case: $n = 0$. There are no CTL* formulae of length 0, hence this base case trivially satisfies the induction hypothesis.

Inductive Step: Let ϕ be an arbitrary CTL* formula of length $n + 1$. We will utilize the induction hypothesis for formulae of length $\leq n$ to prove that Theorem 1 holds for ϕ. There are seven cases to consider.

1. If ϕ is a state formula and an atomic proposition, then $\Omega(\phi) = \phi$. Since s and s' share the same labels, we have that $s \models \phi \Leftrightarrow s' \models \phi \Leftrightarrow s' \models \Omega(\phi)$.
2. If ϕ is a state formula and $\phi = \neg\phi_1$, then $\Omega(\phi) = \neg\Omega(\phi_1)$. Using the induction hypothesis (since the length of ϕ_1 is exactly one less than that of ϕ), we have that $s \models \phi_1 \Leftrightarrow s' \models \Omega(\phi_1)$. Consequently, we have that $s \models \phi \Leftrightarrow s \not\models \phi_1 \Leftrightarrow s' \not\models \Omega(\phi_1) \Leftrightarrow s' \models \Omega(\phi)$.
3. If ϕ is a state formula and $\phi = \phi_1 \wedge \phi_2$, then $\Omega(\phi) = \Omega(\phi_1) \wedge \Omega(\phi_2)$. By definition, we know that $s \models \phi \Leftrightarrow \big((s \models \phi_1) \text{ and } (s \models \phi_2)\big)$. Using the induction hypothesis, since the lengths of ϕ_1 and ϕ_2 are strictly less than the length of ϕ, we have that $(s \models \phi_1) \Leftrightarrow s' \models \Omega(\phi_1)$, and also that $(s \models \phi_2) \Leftrightarrow s' \models \Omega(\phi_2)$. Therefore, we have that $s \models \phi \Leftrightarrow \big(s' \models \Omega(\phi_1) \text{ and } s' \models \Omega(\phi_2)\big) \Leftrightarrow s' \models \Omega(\phi)$.
4. If $\phi = \mathsf{E}\phi_1$, then $\Omega(\phi) = \mathsf{E}\Omega(\phi_1)$. The relation $s \models \phi$ is true iff there exists an infinite path σ beginning at state s such that $\sigma \models \phi_1$. Consider any σ beginning at s (regardless of whether $\sigma \models \phi_1$), and let σ' be an infinite path beginning at state s' such that $G(\sigma, \sigma')$. Such a σ' must exist by Lemma 1. Using the induction hypothesis, since the length of ϕ_1 is exactly one less than the length of ϕ, we have that $\sigma \models \phi_1 \Leftrightarrow \sigma' \models \Omega(\phi_1)$. This implies that $s \models \phi \Leftrightarrow s' \models \Omega(\phi)$.
5. If $\phi = \mathsf{A}\phi_1$, then $\Omega(\phi) = \mathsf{A}\Omega(\phi_1)$. The expression $s \models \phi$ is true iff for every infinite path σ beginning at state s, we have that $\sigma \models \phi_1$. For every infinite path σ beginning at s, consider the infinite path σ' beginning at state s' such that $G(\sigma, \sigma')$. Such a path must exist by Lemma 1. Applying the induction hypothesis, since the length of ϕ_1 is exactly one less than the length of ϕ, we have that $\sigma \models \phi_1 \Leftrightarrow \sigma' \models \Omega(\phi_1)$. This implies that $s \models \phi \Leftrightarrow s' \models \Omega(\phi)$.
6. Suppose ϕ is a path formula which is also a state formula. Since we have exhausted the possibilities for state formulae in the other cases, we conclude that $\pi \models \phi \Leftrightarrow \pi' \models \Omega(\phi)$.
7. If $\phi = \mathsf{X}^{\mathsf{c}}\phi_1$, then $\Omega(\phi) = \mathsf{X}\Omega(\phi_1)$. Note that expression $\pi \models \phi$ is true iff $\pi^c = (s_c, s_{c+1}, s_{c+2}, \ldots) \models \phi_1$. Since $G(\pi^c, \pi'^1)$, and using the induction hypothesis (since the length of ϕ_1 is less than the length of ϕ), we have that $\pi^c \models \phi_1 \Leftrightarrow \pi'^1 = (s'_1, s'_2, s'_3, \ldots) \models \Omega(\phi_1)$. This is equivalent to $\pi \models \phi \Leftrightarrow \pi' \models \mathsf{X}\Omega(\phi_1)$, and also to $\pi \models \phi \Leftrightarrow \pi' \models \Omega(\phi)$.

Note that one by-product of this abstraction is that it extends the initial state set of M' to encompass all states reachable within $c-1$ time-steps from the initial state of M. The above cases prove preservation of CTL* model checking along each path, and with respect to each state. The necessity of prefixing all c-slow reducible CTL* formulae with EF or AG prevents this extended initial set from becoming visible to properties.

Theorem 2. If N' is a c-slow abstraction of N, then $N \prec N'$.

Proof. Our induction hypothesis is that Theorem 2 holds for all c-slow netlists, where $c \leq n$. If $c < 2$, no c-slow abstraction will be performed.

Base Case: $n = 2$.

As depicted in Figure 3, in this case N has two sets of FFs, where X is color 0, and Z is color 1. The abstraction (used to generate N') is performed as described in Section 3.

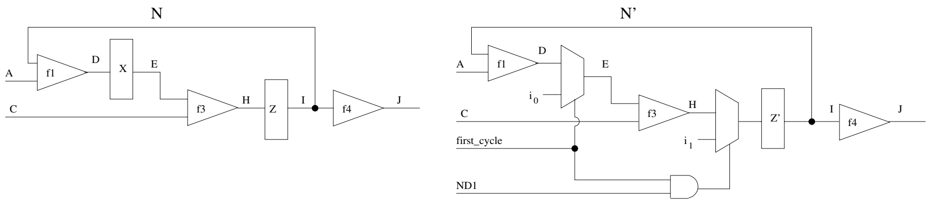

Fig. 3. Original and Abstracted Netlists, N and N', Base Case (C=2)

This proof may be readily completed by enumerating an inductive symbolic simulation for the two netlists (as in Tables 1 and 2), and demonstrating their bisimilarity. This analysis is omitted here due to space constraints.

Inductive Step:

This step shows that if the bisimulation holds for c up to n, then it also holds for $c = n+1$. The induction relies upon the the addition of "intermediate" colored sets of FFs, which are all replaced with MUXes whose selects are conjunctions of a unique random value (for that color) with *first_cycle*, depicted in Figures 4 and 5 as the "inductive units". A set of zero or more inductive units is depicted as a cloud within these two figures, and used in the obvious manner to bring c (along with the explicitly depicted inductive unit) up to $n + 1$.

This proof also may be completed by an inductive symbolic simulation.

The proof of correctness for feed-forward pipelines is omitted due to space constraints, but follows immediately from the example in the inductive step, by omitting the color 0 and $c - 1$ FFs (and their feedback path).

5 Algorithms

Our algorithm for determining the maximum c is similar to that presented in [2]. We iterate over each FF in the netlist; if unlabeled, it is labeled with an

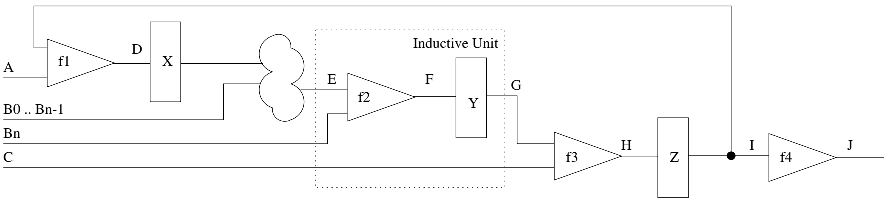

Fig. 4. Unabstracted Netlist N, Inductive Case

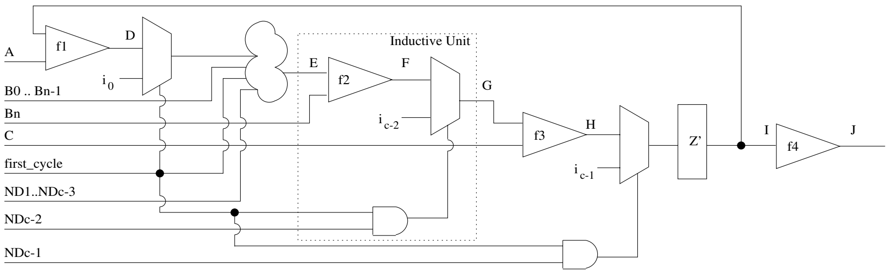

Fig. 5. Abstracted Netlist N', Inductive Case

arbitrary index 0. We next perform a depth-first fanout search from this FF; each time we encounter a FF, we label it with the current index and increment that index for the recursive fanout search from that FF. Once the fanout search is finished, we perform a depth-first fanin search from the original FF; each time we encounter an unlabeled FF, we label it with the current index and decrement that index for the recursive fanin search from that FF. When an already-labeled FF is encountered during a fanin or fanout search, we find the greatest-common divisor of the previous "maximum c" and the difference between the previous index of this FF and the current index. Once completed, this algorithm (which runs in linear time) yields the maximum value of c. If c is equal to 1, no c-slow abstraction may be performed. If c is never updated during coloring, the netlist is a feed-forward pipeline. To obtain the FF colorings, we transform the initial indices modulo c such that the property-visible FFs have color $c-1$. For feed-forward pipelines, we merely shift the initial indices such that the lowest-numbered ones are equivalent to 0.

Abstraction of the netlist occurs as in Section 3. Rather than introducing $c-1$ nondeterministic variables (used only for the calculation of initial values), we need only $\lceil log_2(c) \rceil$ variables. This is due to the observation that, if the nondeterministic value conjuncted for the selector of a color i mux is equivalent to 1, then the nondeterministic values conjuncted with the selectors of all color j muxes (where $j < i$) are don't cares. We may utilize this log_2 nondeterministic value to determine the color of the FFs whose initial values will propagate to the remaining FFs. Thus, an abstract initial state may be bound to a unique timeframe of the concrete design. Note that, regardless of the implementation, the same random value must be utilized for all color i muxes to ensure atomicity of the data.

Other than the $\lceil log_2(c) \rceil$ nondeterministic values introduced for initial value generation, the only other variables introduced are due to splitting of input cones. While this may increase the number of inputs of the design by a factor of c, we have found in practice that this increase is quite small – a small fraction of the total number of inputs – and negligible compared to the number of state variables removed. Despite this increase, these split inputs should not cause a serious BDD blowup since they tend to form independent (on a per-color basis) input ports, which a sophisticated BDD ordering may exploit.

5.1 Trace Lifting

We will perform our model checking upon the abstracted netlist N'. However, we must translate the traces obtained to the original netlist N. We have found that the simplest way to perform the translation is to generate a testcase by projecting the trace from N' down to the inputs, and manipulating this testcase for application to N. We perform this generation in two steps: a prefix generation, and a suffix generation.

For suffix generation, if no input splitting occurs, we may merely stutter each input $c-1$ times to convert the testcase. If input splitting does occur, we apply the stuttering technique to all non-split inputs. We define the color of an input as the color of the FFs to which it fans out. For example, if input A drives FFs of color 0 and 2, it will be split into two inputs – A_0 and A_2, of color 0 and 2, respectively. We use the value upon input A_i at cycle j in the abstract trace for the value at time $c \cdot j + i$ of input A in the testcase for N. We fill in any remaining gaps by an arbitrary selection of any legal value; this value does not influence the behavior of interest. The suffix generation accounts for the fact that every transition of the abstract machine correlates to a c-transition of the original machine.

The prefix generation is used to prepend to the suffix trace a path suitable to transition N from an initial state to the first state in the suffix trace (which is an element of S_{init}). Letting *nd_init* be the value encoded in the log_2 initial value variables in the initial abstract state, the length of the prefix p is equal to $c - 1 - nd\_init$. Generating the prefix backwards (and beginning with $i = 0$), we iteratively prepend the input values (from the *initial value* copy of the netlist) which fan out to color $c - 1 - i$ FFs, then increment i and repeat until $i \equiv p$. For feed-forward pipelines, all trace lifting is performed via prefix generation.

6 Experimental Results

We utilized IBM's model checker, RuleBase [10], to obtain our experimental results. We arbitrarily selected ten components of IBM's Gigahertz Processor which had previously been model checked. Our algorithm identified two of these as being c-slow. The first is a feed-forward pipeline; the second is the five-slow pipeline with feedback mentioned in Section 1. Both were explicitly entered in HDL as c-slow designs – this topology is not the by-product of a synthesis tool. Both had multi-colored inputs, which our generalized topology was able to exploit. Both of these components had been undergoing verification and regression for more than 12 months prior to the development of this abstraction technique. Consequently, the unabstracted variants had very good BDD orderings available. All results were obtained on an IBM RS/6000 Workstation Model 595 with 2 GB main memory. RuleBase was run with the most aggressive automated model reduction techniques it has to offer (including dual-phase abstraction [5]), and with dynamic BDD reordering (Rudell) enabled.

Prior to running the c-slow abstraction algorithm on these netlists, we ran automated scripts which removed scan chain connections between the latches (which unnecessarily limited c), and which cut self-feedback on "operational mode" FFs (into which values are scanned prior to functional use of the netlist, and held during functional use via the self-feedback loop).

We first deployed this abstraction technique on the feed-forward pipeline. The most interesting case was the most complex property against which we verified this design. The unabstracted version had 148 variables, and with our best initial ordering took 409.6 seconds with a maximum of 1410244 allocated BDD nodes. The first run on the abstracted variant (with a random initial ordering) had 53 variables, and took 44.9 seconds with a maximum of 201224 BDD nodes. While this speedup is significant, this comparison is skewed since the unabstracted run benefited from the extensive prior BDD reordering. Re-running the unabstracted experiment with a random initial ordering took 3657.9 seconds, with 2113255 BDD nodes. Re-running the abstracted experiment using the ordering obtained during the first run as the initial ordering took 4.6 seconds with 98396 nodes. Computing the c-slow abstraction took 0.3 seconds.

The next example is the five-slow design. With a good initial ordering, model checking the unabstracted design against one arbitrarily selected formula took 5526.4 seconds, with 251 variables and 3662500 nodes. The first run of the abstracted design (with a random initial ordering) took 381.5 seconds, with 134 variables and 339424 nodes. Re-running the rule twice more (and re-utilizing the calculated BDD orders) yielded a run of 181.1 seconds, 293545 nodes. Model checking the unabstracted design with an random initial ordering took 23692.5 seconds, 7461703 nodes. Computing the c-slow abstraction took 3.2 seconds.

Note that, due to the potential increase in depth of combinational cones entailed by this abstraction, there is a risk of a serious blowup of the transition relation or function. Splitting or conjoining may be utilized to combat such blowup [11]. A reasonable ordering seems fairly important when utilizing this abstraction. One set of experiments were run with reordering off, and a random initial ordering. The results for the feed-forward pipeline were akin to those reported above. However, the five-slow abstracted transition relation was significantly larger than the unabstracted variant given the random ordering, thereby resulting in a much slower execution than on the unabstracted run. With reordering enabled, the results (as reported above) were consistently superior.

7 Conclusions and Future Work

We have developed an efficient algorithm for identifying and abstracting generalized flip-flop based c-slow netlists. Our approach generalizes the definition provided in [2]; this generality allows us to apply our abstraction to a substantial percentage of design components. This abstraction is fully automated, and runs in $O(c \cdot nets)$ time. Our abstraction decreases the number of state variables, and the diameter of the model by c. Our experimental results indicate the substantial benefit of this abstraction in reducing verification time and memory (one to two magnitudes of order improvement), when applicable. We discuss expressibility constraints on CTL* model checking using this abstraction.

Future work items involve efficient techniques for identification of netlist substructures which violate c-slowness, yet may be safely replaced by others which do not. Other work involves extending the class of netlists to which this technique may be applied through the splitting of sequential cones.

References

1. C. E. Leiserson and J. B. Saxe. Optimizing Synchronous Systems. In *Journal of VLSI and Computer Systems*, 1(1):41–67, Spring 1983.
2. C. E. Leiserson and J. B. Saxe. Retiming Synchronous Circuitry. In *Algorithmica*, 6(1):5–35, 1991.
3. S. Mador-Haim and L. Fix. Input Elimination and Abstraction in Model Checking. In *Proc. Conf. on Formal Methods in Computer-Aided Design*, November 1998.
4. G. Hasteer, A. Mathur, and P. Banerjee. Efficient Equivalence Checking of Multi-Phase Designs Using Phase Abstraction and Retiming. In *ACM Transactions on Design Automation of Electronic Systems*, October 1998.
5. J. Baumgartner, T. Heyman, V. Singhal, and A. Aziz. Model Checking the IBM Gigahertz Processor: An Abstraction Algorithm for High-Performance Netlists. In *Proceedings of the Conference on Computer-Aided Verification*, July 1999.
6. R. Milner. *Communication and Concurrency.* Prentice Hall, New York, 1989.
7. Z. Kohavi. *Switching and Finite Automata Theory.* Computer Science Series. McGraw-Hill Book Company, 1970.
8. O. Grumberg and D. E. Long. Module Checking and Modular Verification. In *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.
9. E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching Time Logic. In *Proc. Workshop on Logic of Programs*, 1981.
10. I. Beer, S. Ben-David, C. Eisner, and A. Landver. RuleBase: an Industry-Oriented Formal Verification Tool. In *Proc. Design Automation Conference*, June 1996.
11. A. J. Hu, G. York, and D. L. Dill. New Techniques for Efficient Verification with Implicitly Conjoined BDDs. In *Proceedings of the Design Automation Conference*, June 1994.