Model Checking the IBM Gigahertz Processor: An Abstraction Algorithm for High-Performance Netlists

Jason Baumgartner[1], Tamir Heyman[2], Vigyan Singhal[3], and Adnan Aziz[4]

[1] IBM Corporation, Austin, Texas 78758, USA,
jasonb@austin.ibm.com
[2] IBM Haifa Research Laboratory, Haifa, Israel,
tamirh@vnet.ibm.com
[3] Cadence Berkeley Labs, Berkeley, California 94704, USA,
vigyan@cadence.com
[4] The University of Texas at Austin, Austin, Texas 78712, USA,
adnan@ece.utexas.edu

Abstract. A common technique in high-performance hardware design is to intersperse combinatorial logic freely between level-sensitive latch layers (wherein one layer is transparent during the "high" clock phase, and the next during the "low"). Such logic poses a challenge to verification – unless the two-phase netlist N may be abstracted to a full-cycle model N' (wherein each memory element may sample every cycle), model checking of N requires at least twice as many state variables as would be necessary to obtain equivalent coverage for N'. We present an algorithm to automatically obtain such an abstraction by selectively eliminating latches from both layers. The abstraction is valid for model checking CTL* formulae which reason solely about latches of a single phase. This algorithm has been implemented in IBM's model checker, RuleBase, and has been used to enable model checking of IBM's Gigahertz Processor, which may not have been feasible otherwise. This abstraction has furthermore allowed verification engineers to write properties and environments more efficiently.

1 Introduction

A *latch* is a hardware memory element with two Boolean inputs – data and clock – and one Boolean output. A behavioral definition for latches is provided in [1]. High performance netlists often must use level-sensitive latches [2]. For such a latch, when its clock input is a certain value (e.g., a logical "1"), the value at its data input will be propagated to its data output (i.e., transparent mode); otherwise, its last propagated value is held at its output.

The clock is modeled as a signal which alternates between 0 and 1 at every time-step. A latch which samples when the clock is a 1 will be denoted as an L1 latch; one which samples when the clock is a 0 will be denoted as an L2 latch. Hardware design rules, arising from timing constraints, require any logic

path between two L1 latches to pass through an L2 latch, and vice-versa. An elementary design style requires each L1 latch to feed directly to an L2 latch (called a master-slave latch pair), and allow only L2 to drive combinatorial logic. However, a common high-performance hardware development technique involves utilizing combinatorial logic freely between L1 and L2 latches to better utilize each half-cycle. It should be noted that such designs are typically explicitly implemented in this manner; this topology is not the byproduct of a synthesis tool.

There are two major problems with the verification of such netlists. First, because of the larger number of latches, the verification tool requires much more time and memory. Additionally, the manual modeling of environments and properties is more complicated in that they must be written in terms of the less abstract half-cycle model, and an oscillating clock must be explicitly introduced.

Most hardware compilers will allow automatic translations of a master-slave latch pair into a single flip-flop; retiming algorithms [3] may be used to retime the netlist such that L1-L2 layers become adjacent and one-to-one. However, retiming adds complexity in that the specification, the environment, and any witnesses / counterexamples (all of which may "observe" the netlist), may need to be retimed as well to match the retimed, full-cycle model.

We develop an efficient algorithm for abstracting a half-cycle netlist N to a full-cycle model N', which may be utilized for enhanced verification in any FSM-based verification framework (e.g., simulation and model checking). We will achieve this by selectively eliminating some latches. We will use a notion of dual-phase-bisimulation equivalence between the abstracted and unabstracted models. This equivalence ensures that specification and environment written in terms of L2 latch outputs need not be modified other than a conversion to full-cycle format (as will be discussed in Sect. 3). Our algorithm performs maximum such reductions, and thus provides an important model reduction step which may greatly augment existing techniques (such as retiming, cone-of-influence, etc.). As we show, this reduction alone reduces the number of state variables by at least one-half, and has greatly enhanced the model checking of IBM's Gigahertz Processor, which may not have been feasible otherwise (as demonstrated by our experimental evidence). This abstraction is now part of the model checker RuleBase [4]. Additionally, designers and verification engineers prefer to reason about the full-cycle models.

The optimality of the algorithm results from the identification of *minimal dependent layers (MDL)* of latches, and removing all L1s or all L2s per MDL.

Definition 1. *A* dependent layer *is a set of L1 and L2 latches $L1'$ and $L2'$, such that $L2'$ is a superset of all latches in the transitive fanout of $L1'$, and $L1'$ is a superset of all latches in the transitive fanin of $L2'$.*

Definition 2. *A dependent layer is termed* minimal *if and only if there does not exist a nonempty set of L1 and L2 latches L' which may be removed from that layer and still result in a nonempty dependent layer.*

Consider the netlist in Fig. 1 (the triangles denote combinatorial logic, and the rectangles denote latches). The L1 latches are shaded. The two unique MDLs are marked with dotted boxes. Merely removing all L1s or all L2s will not yield an optimum reduction in this case; the L1s of layer A, and the L2s of layer B should be removed to yield an optimum solution for this netlist, which removes four of the six latches.

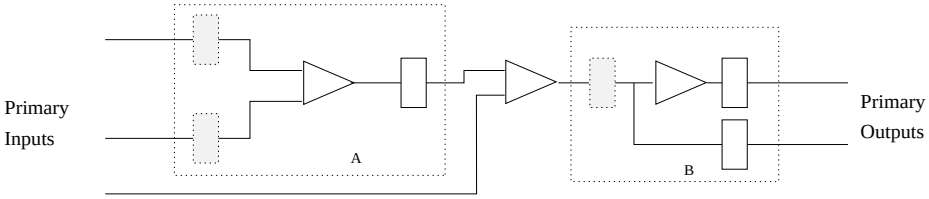

Fig. 1. Sample Netlist with Two Minimal Dependent Layers

In Sect. 2 we introduce a half-cycle netlist, and two different abstracted full-cycle models of this netlist. In Sect. 3 we study the state space of the netlist and its two abstracted models to demonstrate the validity of the abstraction for CTL* formulae which reason solely about latches of a single type (L1 or L2). In Sect. 4 we introduce the algorithm used to perform the netlist reduction, and demonstrate its optimality. In Sect. 5 we give some experimental results of the use of this algorithm as implemented in RuleBase [4] for application to IBM's Gigahertz Processor.

2 Half-Cycle versus Full-Cycle Models

Consider the half-cycle netlist, denoting an MDL, shown in Fig. 2. All nets and primitives may be vectors.

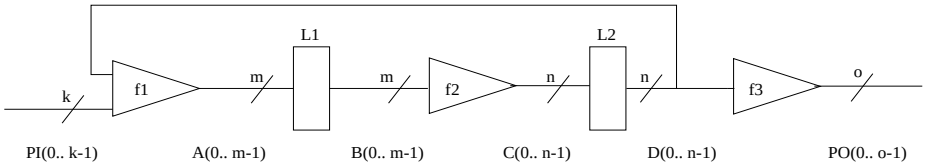

Fig. 2. Half-Cycle Netlist N

Definition 3. *A netlist is* dual-phase (DP) *if and only if:*

1. *All latches in the transitive fanouts of L1 latches are L2 latches, and*

2. *All latches in the transitive fanin of L2 latches are L1 latches, and*
3. *No primary inputs exist in the transitive fanin of any L2 latch, and*
4. *No primary outputs exist in the transitive fanout of any L1 latch.*

The first two rules are enforced by hardware timing constraints. Note that, at the periphery of a design, there may be some inputs which have L2 latches in their transitive fanout, and outputs which are in the transitive fanout of L1 latches (thus violating rules 3 and 4). While the analysis presented in this paper disallows such connectivity for simplicity, these cases are supported by our implementation; for ease of reasoning, we have found it beneficial to preserve all L2 latches which violate rule 3, and to remove all L1 latches which violate rule 4.

The notion of MDLs (Defn. 2) allows us to partition the design under test into a maximum number of partitions such that each is DP. Next, we propose two abstractions for DP netlists. For each DP partition of the original design, one of these two abstractions may be applied independently of the other partitions, thus yielding an overall abstraction which has a globally minimum number of latches (refer to Theorem 5). This minimum would, in general, be less than removing either all of the L1 or all of the L2 latches.

In this paper, we assume that properties may only refer to the L2 nets (which we term $L2-visible$ properties). In our actual implementation, we also handle the case where the properties refer only to L1 nets. Furthermore, by forcing our tool to remove only L1 or only L2 latches (i.e., restricting its freedom to choose which type to remove), each property may refer to both types of nets. However, we skip these generalizations in this paper.

2.1 The Abstracted Models

The values of the nets in Fig. 2 are specified for time-steps $i \geq 0$. The pre-specified initial values of the latches are $B_0(0..m-1)$ and $D_0(0..n-1)$. Let c denote the clock input, which initializes to 1, and alternates between 1 and 0 at every time step, indicating whether the L1 or L2 latches (respectively) are presently transparent. The subscript i means "at time i".

For $i > 0$, if $(c_i = 1)$, $B_i(0..m-1) = A_{i-1}(0..m-1)$, else $B_i(0..m-1) = B_{i-1}(0..m-1)$. Similarly, for $i > 0$, if $(c_i = 1)$, $D_i(0..n-1) = D_{i-1}(0..n-1)$, else $D_i(0..n-1) = C_{i-1}(0..n-1)$. For the combinatorial nets, $A_i(0..m-1) = f1(PI_i(0..k-1), D_i(0..n-1))$; $C_i(0..n-1) = f2(B_i(0..m-1))$; and $PO_i(0..o-1) = f3(D_i(0..n-1))$.

Either layer of latches may be removed (and the remaining layer transformed to flip-flops which may be clocked every cycle – not by an alternating clock), and the resulting abstracted model will be shown to be bisimilar to the original netlist. Fig. 3 shows the first abstraction with layer L2 removed. We need a new variable, labeled f, whose initial value is 1, and thereafter is 0. This latch ensures that the initial value D_0 from the original netlist N (which need not be deterministic) is applied to the combinatorial nets D in N'. $B_0(0..m-1)$ is still the initial value of the remaining latches. For $i > 0$, $B_i(0..m-1) = A_{i-1}(0..m-1)$.

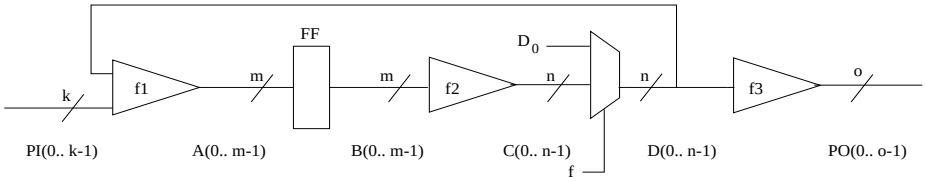

Fig. 3. Abstracted Model N'

If $(f_i = 1)$, $D_i(0..n-1) = D_0(0..n-1)$, else $D_i(0..n-1) = C_i(0..n-1)$. For the other combinatorial nets, $A_i(0..m-1) = f1(PI_i(0..k-1), D_i(0..n-1))$; $C_i(0..n-1) = f2(B_i(0..m-1))$; and $PO_i(0..o-1) = f3(D_i(0..n-1))$.

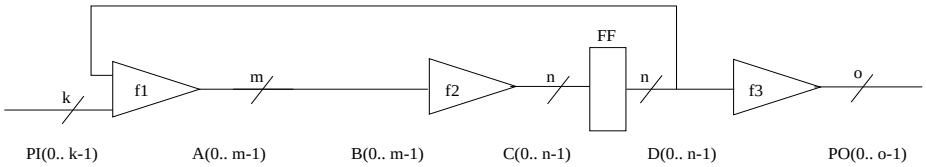

Fig. 4. Alternate Abstracted Model N''

Fig. 4 illustrates the second abstraction, which removes the L1s. $D_0(0..n-1)$ is the initial value of the remaining latches; $D_i(0..n-1) = C_{i-1}(0..n-1)$. For the combinatorial nets, $A_i(0..m-1) = f1(PI_i(0..k-1), D_i(0..n-1))$; $B_i(0..m-1) = A_i(0..m-1)$; $C_i(0..n-1) = f2(B_i(0..m-1))$; and $PO_i(0..o-1) = f3(D_i(0..n-1))$. Note that the f variable is unnecessary for this abstraction; the initial value of the removed latch does not propagate.

It is noteworthy that either one of the two abstractions may be chosen; since the layers may be of differing width ($m \neq n$), the removal of one layer may result in a smaller state space than the other. We term both of the above reductions as dual-phase (DP) reductions.

3 Validity of Abstraction

We define a notion of dual-phase-bisimulation relation (inspired by Milner's bisimulation relations [5]); this notion is preserved for composition of Moore machines. Further, if two structures are related by a dual-phase-bisimulation relation, we show that $L2-visible$ CTL* properties are preserved (modulo a simple transformation). We show the existence of a dual-phase-bisimulation relation for both abstractions presented in the previous section.

We will relate our designs to Kripke structures, which are defined as follows.

Definition 4. *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,* $\mathcal{L} : S \to 2^{\mathcal{A}}$ *is the labeling function, and* $R : S \times S$ *is the transition relation.*

Our designs are described as Moore machines (using Moore machines, instead of the more general Mealy machines [6], simplifies the exposition for this paper, though our implementation is able to handle Mealy machines). We use the following definitions for a Moore machine and its associated structure (similar to Grumberg and Long [7]).

Definition 5. *A* Moore machine $\mathcal{M} = \langle L, S, S_0, I, O, V, \delta, \gamma \rangle$, *where* L *is the set of state variables (latches),* $S : 2^L$ *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 nets,* $\delta : S \times 2^I \times S$ *is the transition relation, and* $\gamma : S \rightarrow 2^O$ *is the output function.*

Definition 6. *The* structure of 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* $S^K = 2^L \times 2^I$, $S_0^K = S_0 \times 2^I$, $\mathcal{A} = V$, $\mathcal{L} = S^K \rightarrow 2^V$, *and* $R((s,x),(t,y))$ *iff* $\delta(s,x,t)$.

In the sequel we will use M to denote the Moore machine as well as the structure for the machine. We now define our notion of dual-phase-bisimilarity, which characterizes our proposed abstraction.

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

1. $\mathcal{L}(s) = \mathcal{L}'(s')$.
2. *for every* $t, v \in S$, *such that* $R(s,v)$ *and* $R(v,t)$, *we have* $\mathcal{L}(s) = \mathcal{L}(v)$, *and 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 exist* $t, v \in S$ *such that* $\mathcal{L}(v) = \mathcal{L}'(s')$, $R(s,v)$, $R(v,t)$, *and* $G(t,t')$.

We say that a dual-phase-bisimulation exists from M *to* M' *(denoted by* $M \prec M'$*) iff there exists a dual-phase-bisimulation relation* G *such that for all* $s \in S_0$ *and* $t' \in S_0'$, *there exist* $t \in S_0$ *and* $s' \in S_0'$ *such that* $G(s,s')$ *and* $G(t,t')$.

Notice that, in the above definition, such a dual-phase-bisimulation relation may exist only if M has a dual-phase nature – i.e., for all i, the visible labels of states s_{2i} and s_{2i+1} are equivalent.

An infinite path $\pi = (s_0, s_1, s_2, \ldots)$ is a sequence of states ($s_0 \in S_0$) 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 the dual-phase-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 every i, $G(s_{2i}, s_i')$.

The composition of Moore machines ($M_1 \parallel M_2$) is defined in the standard way [7], by allowing the outputs of one design to become inputs of the other. The following result is shown similarly as the proof that simulation precedence is preserved under composition [7].

Theorem 1. *If* $M_1 \prec M_1'$ *and* $M_2 \prec M_2'$, *then a dual-phase-bisimulation exists from the Moore composition* $M_1 \parallel M_2$ *to the Moore composition* $M_1' \parallel M_2'$ *(i.e.,* $M_1 \parallel M_2 \prec M_1' \parallel M_2'$*).*

The set of dual-phase-reducible CTL* formulae is a subset of CTL* formulae [8], and is a set of state and path formulae given by the following inductive definition. We also define the dual-phase reduction for such formulae:

Definition 8. *A* dual-phase-reducible (DPR) *CTL* formula ϕ, and its* dual-phase reduction*, denoted by $\Omega(\phi)$, are defined inductively as:*

- *every atomic proposition p is a DPR state formula $\phi = p$; $\Omega(\phi) = p$.*
- *if p is a DPR state formula, so is $\phi = \neg p$; $\Omega(\phi) = \neg\Omega(p)$.*
- *if p, q are DPR state formulae, so is $\phi = p \wedge q$; $\Omega(\phi) = \Omega(p) \wedge \Omega(q)$.*
- *if p is a DPR path formula, then $\phi = \mathsf{E}p$ is a DPR state formula; $\Omega(\phi) = \mathsf{E}\Omega(p)$.*
- *each DPR state formula ϕ is also a DPR path formula ϕ.*
- *if p is a DPR path formula, so is $\phi = \neg p$; $\Omega(\phi) = \neg\Omega(p)$.*
- *if p, q are DPR path formulae, so is $\phi = p \wedge q$; $\Omega(\phi) = \Omega(p) \wedge \Omega(q)$.*
- *if p is a DPR path formula, so is $\phi = \mathsf{XX}p$; $\Omega(\phi) = \mathsf{X}\Omega(p)$.*
- *if p, q are DPR path formulae, so is $\phi = p\mathsf{U}q$; $\Omega(\phi) = \Omega(p)\mathsf{U}\Omega(q)$.*

Note that XX is transformed to X through the reduction; intuitively, this is due to the "doubling of the clock frequency", or the replacement of the oscillating clock with an "always active" clock, enabled by the abstraction. As an example, if $\phi = \mathsf{AG}(rdy \rightarrow (\mathsf{AXAX}(req \rightarrow \mathsf{AF}(ack))))$, then $\Omega(\phi) = \mathsf{AG}(rdy \rightarrow (\mathsf{AX}(req \rightarrow \mathsf{AF}(ack))))$ (note that AXAXp is equivalent to AXXp). $L2 - visible$ properties may be readily expressed utilizing DPR CTL*, since latches of any given type may only toggle every second time-step; there is no need to express such a property with a single X, which is the only restriction we impose upon full CTL* expressibility.

Theorem 2. *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 dual-phase-bisimulation relation such that $G(s, s')$ and $G(\pi, \pi')$, then*

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

We describe the Moore specifications (Defn. 5) for the abstractions presented in Sect. 2. Refer to Figs. 2-4. Let c be the clock variable which alternates between 1 and 0, indicating whether the L1 or L2 latches are presently transparent, respectively. The original netlist $N = \langle L^N, S^N, S_0^N, PI, PO, V^N, \delta^N, \lambda^N \rangle$ has $L^N = B \cup D \cup \{c\}$, $V^N = D$, and the transition and output functions, δ^N and λ^N, are given by the formulae in Sect. 2.1. The state space of N is denoted by (b, d, v, x), comprising of the values of latches B, D, c, and the input PI, respectively.

As presented here, the properties cannot refer to inputs – V^N does not contain inputs. This restriction is due to the requirement that visible labels of states s_{2i} and s_{2i+1} are identical (Defn. 7), and is not necessary if the inputs to the design do not change values between s_{2i} and s_{2i+1}. This assumption is typically sound; except for clock inputs (which no longer need to be modeled), synthesis

timing constraints enforce this requirement (since the partition will ultimately be composed with other partitions, or occur at chip boundaries). After our abstraction, the environment is no longer constrained from toggling only once every two time-steps, but may toggle every time-step – this reflects a conversion of the environment from half-cycle to full-cycle, and (along with the synthesis requirements reflected in rules 1 and 2 of Defn. 3, and the synthesis requirement that the design be free from combinatorial loops) allows applicability of this abstraction to Mealy machines.

The first abstraction $N' = \langle L^{N'}, S^{N'}, S_0^{N'}, PI, PO, V^{N'}, \delta^{N'}, \lambda^{N'} \rangle$, which we denote the "remove-L2" abstraction, has $L^{N'} = B \cup \{f\}$, $S_0^{N'} = (B_0, 1)$, $V^{N'} = D$. Again, the transition and output functions, $\delta^{N'}$ and $\lambda^{N'}$, are given by the formulae in Sect. 2.1. The state space of N' is denoted by (b, w, x), comprising of the values of latches B, f, and the input PI, respectively. The second abstraction N'', which we denote the "remove-L1" abstraction, has $L^{N''} = D$, $S_0^{N''} = D_0$, $V^{N''} = D$. The state space of N'' is denoted by (d, x), comprising of the values of latches D and the input PI, respectively. Note that we define V in all cases as D, the L2 latch outputs, as necessary for arbitrary $L2 - visible$ properties, and to enable the dual-phase-bisimulation.

Theorem 3. *If N' is a "remove-L2" abstraction of N, then $N \prec N'$.*

Proof. The following relation G between states of N and N' is a dual-phase-bisimulation relation. G is defined so that it is 1 only for the following two cases:

- for any x, $G((B_0, D_0, 1, x), (B_0, 1, x))$ is 1
- for any $(b, d, 1, x)$ reachable from the initial state of N after at least one transition, $G((b, d, 1, x), (b, 0, x))$ is 1

Theorem 4. *If N'' is a "remove-L1" abstraction of N, then $N \prec N''$.*

Proof. The following relation G between states of N and N'' is a dual-phase-bisimulation relation. G is defined so that it is 1 only for the following two cases:

- for any x, $G((B_0, D_0, 1, x), (D_0, x))$ is 1
- for any $(b, d, 1, x)$ reachable from the initial state of N after at least one transition, $G((b, d, 1, x), (d, x))$ is 1

Theorems 1, 2, 3 and 4 allow us to apply the two abstractions independently on each dependent layer, and still show the validity of model checking $L2-visible$ CTL* formulae on the composition of the abstractions:

Corollary 1. *If D' is obtained from D by applying either the "remove-L2" abstraction or the "remove-L1" abstraction, independently, on each of its minimal dependent layers, $D \prec D'$.*

4 Algorithm for the Abstraction of DP Netlists

The algorithm picks a primary input at random – a *while* loop ensures that every primary input is chosen. It then finds the latches in the transitive fanout of this input – this set is called $L1''$, and must consist solely of L1 latches (except for inputs connected to L2s, which are treated specially). It places these elements of $L1''$ one-at-a-time into the set $L1'$. For each latch in $L1'$ not previously considered, it finds all L2 latches in the transitive fanout of $L1'$ – this set is denoted $L2'$. It then looks for any latches in the transitive fanin of $L2'$ – these must be L1s – and adds them to $L1'$. It then iteratively ping-pongs between the L1s and L2s for this MDL until no new latches are found. These latches are now labeled with their type and layer identifier (which is then incremented), and a record kept as to the number of L1 and L2 latches in that layer. It then continues iteratively with the next element of the set $L1''$.

The algorithm then looks for L1 latches in the transitive fanout of the L2s encountered in the previous layers. If it finds any, these new MDLs are explored iteratively as above until no new latches are encountered. The outer *while* loop then begins traversing from the next primary input.

If the algorithm encounters a previously-marked L1 latch while looking for an L2 latch (or vice-versa), it flags this violation. If no violation has been reported during the analysis, the netlist is DP, and reduction may proceed.

After the above analysis, either the L1 or the L2 set may be removed per layer; these layers are minimal by construction. A simple iteration over every MDL will yield optimum reductions; if the given layer has more L2s than L1s, the L2s of that layer should be replaced with multiplexors as discussed in Sect. 2.1. If not, the L1s of that layer should be replaced with wires.

If the type of all latches is provided (L1 versus L2), an alternate algorithm may simply iterate over each latch within the netlist, and calculate its MDL given its type. When this abstraction was initially deployed, no such type data was automatically available; the inputs provided a convenient point of reference.

Theorem 5. *This algorithm performs optimum DP reductions.*

Proof. By construction, each latch will be a member of exactly one MDL. Furthermore, the MDLs are of minimum size, resulting in a maximum number of dependent layers in the netlist. Since each MDL is independent of the others, the locally optimal solutions yield a globally optimum result.

Note that along any input - output path within a single MDL, exactly one flip-flop must exist after abstraction – if zero or two exist, the bisimulation is clearly broken. Take any latch from any MDL which which was removed by the abstraction – assume that it is an L1 latch $L1'$. All L2s $L2'$ in the fanout of $L1'$ must remain. All L1s in the fanin of $L2'$ must have been removed, and so on until we are left with the case that (within this MDL) all of the L1s are removed, and all of the L2s remain, if this is a correct abstraction (similar reasoning applies to consideration of a removed L2 latch). This demonstrates optimality of reduction of each MDL.

The algorithm may be optimized to ensure that each combinatorial gate (or net) is only considered once in fanout traversal, and once in fanin traversal, to ensure that its complexity is $O(netlist\ size^2)$. However, in practice, we have found that the complexity of this algorithm grows roughly linearly with model size and takes a matter of seconds for even the largest designs we have considered for model checking (more than 8,000 latches). This near-linearity is not surprising for synthesizable high-performance netlists, since the depth of combinatorial logic between latches and the number of sinks of a net are restricted to ensure that the netlist meets timing constraints.

5 Experimental Results

The above algorithm was implemented into the model checker RuleBase [4], developed in IBM Haifa Research Lab as an extension to SMV [9]. It is utilized as a first-pass netlist reduction technique; the reduced full-cycle model is saved and used as the basis for further optimizations before being passed to SMV for model checking.

This algorithm was deployed for use on many components of IBM's Gigahertz Processor. The reduction results obtained by this step are given in Table 1 below. These numbers do not reflect the results of any other reduction techniques. We recommend, due to the speed of this algorithm ($O(n^2)$ in theory, but roughly $O(n)$ in practice) and its global preservation of $L2 - visible$ properties, that it be used as a first-pass reduction technique upon design compilation. The resulting abstracted design may then be analyzed for formula-specific reductions (e.g., cone-of-influence, constant propagations, retiming), which are likely to proceed faster upon the abstracted design due to the fewer number of latches and simpler transition relation (the clock is no longer in the support of the transition relation).

Logic Function	State Bits Before Reduction	State Bits After Reduction
Load Serialization Logic	8096	2586
L1 Reload Logic	3102	1418
Instruction Flushing Logic	138	69
Instruction-Fetch Address Generation Logic	4891	2196
Branch Logic	6918	3290
Instruction Issue Logic	6578	3249
Tag Management Logic	578	289
Instruction Decode Logic	1980	978
Load / Store Control	821	409

Table 1. DP Reduction Results

During the initial stages of model checking, this abstraction was not available. Once the abstraction became available, properties which previously took many

hours to complete would finish in several minutes. More encompassing properties became feasible on the abstracted model which would not otherwise complete.

As a small example, a property run on the Load Serialization Logic which took 25.6 seconds, 36 MB of memory on the abstracted model (with 81 variables) took 450.2 seconds, 92 MB of memory for the unabstracted netlist (with 116 variables) on the same machine (an IBM RS/6000 Workstation Model 590 with 2 GB main memory), with no initial BDD order. This time includes that necessary to perform the netlist analysis and reduction. As a larger example, a property run on the Instruction Flushing Logic took 852 seconds of user time, 48 MB on the abstracted model (with 96 variables). This same property did not complete on the unabstracted netlist (with 162 variables) within 72 hours.

While it may seem surprising that the number of variables after abstraction is more than half that before abstraction, this is due to two phenomena. First, some of these variables are used for environment and specification; these are modeled directly as flip-flops (rather than L1-L2 latches). Second, in some cases, RuleBase was able to exploit some redundancy among these variables through other model reduction techniques (e.g., constant simulation).

The benefits obtained by this algorithm extend beyond a mere reduction in state depth, which reduces the time and memory consumed by reachability calculations. BDD variable reordering time is often greatly reduced (since the BDDs tend to be smaller, and since with less variables a "good ordering" tends to be faster to compute). The reduction to full-cycle models also reduces the number of image calculations necessary to reach a fixed-point or on-the-fly failure – the diameter of the model is halved. Further, since fewer state variables require evaluation, it is possible that the above reduction may be exploited to "collapse" adjacent functions to a single function, which may be represented on the same BDD. However, this risks blowing up the BDD size; the functions may thus remain distinct and implicitly conjoined [10] to ensure proper evaluation.

With this abstraction available, as demonstrated above, model checking was enabled to verify much "larger" and more meaningful properties in less time. Users of our tool have found that writing specifications and environments for the full-cycle abstracted models is much less complex than for the corresponding half-cycle netlists (as is viewing traces). All RuleBase users quickly converted to running exclusively with this abstraction. There have been many hundreds of formulae written and model checked to date on this project, which collectively have exposed on the order of 200 bugs at various design stages. We have not encountered any properties we wished to specify which became impossible on the abstracted model. This algorithm thus provided an efficient and necessary means by which to free ourselves from the verification burdens imposed by the low level of the implementation.

It is noteworthy that roughly 70 HDL bugs were isolated due to violations of L1-L2 connectivity during this work. While algorithms for detecting such problems are simple (and other tools implementing such checks became available later in the design cycle), the many benefits resulting from this reduction provided strong motivation for quickly correcting these errors. Due to the nature of logic

interpretation in simulation and model checking frameworks, the logic flawed in such a manner typically behaved "properly" for verification – these platforms assume zero combinatorial delay, but no combinatorial "flow-through" for two adjacent level-sensitive latches even if both are simultaneously in the transparent phase.

6 Conclusions

We have developed an efficient algorithm for identifying and abstracting dual-phase L1-L2 netlists. The algorithm performs netlist graph traversal, rather than FSM analysis, hence is CPU-efficient – $O(n^2)$ in theory, but roughly $O(n)$ in practice due to timing constraints imposed upon synthesizable netlists. The benefits obtained by the abstraction include much smaller verification time and memory requirements (through "shallower" state depth – often less than one-half that necessary without the abstraction – which reduces complexity of the transition relation and simplifies BDD reordering, and a halving of the diameter of the model), as well as more abstract specification and environment definitions. A bisimulation relation is established between the unreduced and reduced models. This reduction is optimum, and is valid for model checking CTL* formulae which reason solely about latches of a given phase. Experimental results from the deployment of this algorithm (as implemented in the model checker RuleBase) upon IBM's Gigahertz Processor are provided, and illustrate its extreme practical benefit.

References

1. S. Mador-Haim and L. Fix. Input Elimination and Abstraction in Model Checking. In G. Gopalakrishnan and P. Windley, editor, *Proc. Conf. on Formal Methods in Computer-Aided Design*, volume 1522, pages 304–320. Springer, November 1998.
2. K. Nowka and T. Galambos. Circuit Design Techniques for a Gigahertz Integer Microprocessor. In *Proc. Intl. Conf. on Computer Design*, October 1998.
3. C. E. Leiserson and J. B. Saxe. Optimizing Synchronous Systems. *Journal of VLSI and Computer Systems*, 1(1):41–67, Spring 1983.
4. I. Beer, S. Ben-David, C. Eisner, and A. Landver. RuleBase: an Industry-Oriented Formal Verification Tool. In *Proc. Design Automation Conf.*, June 1996.
5. R. Milner. *Communication and Concurrency*. Prentice Hall, New York, 1989.
6. Z. Kohavi. *Switching and Finite Automata Theory*. Computer Science Series. McGraw-Hill Book Company, 1970.
7. O. Grumberg and D. E. Long. Module Checking and Modular Verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.
8. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.
9. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
10. A. J. Hu, G. York, and D. L. Dill. New Techniques for Efficient Verification with Implicitly Conjoined BDDs. In *Proc. Design Automation Conf.*, June 1994.