A Practical Approach to Word Level Model Checking of Industrial Netlists

Per Bjesse

Advanced Technology Group
Synopsys Inc.
bjesse@synopsys.com

Abstract. In this paper we present a word-level model checking method that attempts to speed up safety property checking of industrial netlists. Our aim is to construct an algorithm that allows us to check both bounded and unbounded properties using standard bit-level model checking methods as back-end decision procedures, while incurring minimum runtime penalties for designs that are unsuited to our analysis. We do this by combining modifications of several previously known techniques into a static abstraction algorithm which is guaranteed to produce bit-level netlists that are as small or smaller than the original bitblasted designs. We evaluate our algorithm on several challenging hardware components.

1 Introduction

Word-level methods, which leverage design information captured at a higher level than that of individual wires and primitive gates, are the next frontier in hardware verification. At the word level, data-path elements and data packets are viewed as entities in their own right as opposed to a group of bit-level signals without any special semantics.

There has been a lot of activity lately around word-level formula decision procedures such as SMT solvers [10] and reduction-based procedures like `UCLID` [2] and `BAT` [7]. However, as promising as this direction of research is, the use of these procedures for model checking is inherently restricted in that they analyze formulas rather than sequential systems. This has two consequences: First of all, sequential properties can only be checked by these procedures by relying on methods such as induction and interpolation that employ bounded checks to infer unbounded correctness. Second, these procedures do not fit into a *transformation-based* approach to sequential system verification [1], where sequential verification problems are iteratively simplified and processed by any of a large set of back-end model checkers.

In this paper, we introduce a method for practical word-level model checking of both bounded and unbounded properties for hardware designs. Our aim is to (1) not require any additional input from the user, (2) never perform worse than a straight bit-level sequential analysis of a given netlist, and (3) to provide the possibility of speedups when there are significant parts of the design that can be treated on the word level. Our solution is engineered as a reduction method

A. Gupta and S. Malik (Eds.): CAV 2008, LNCS 5123, pp. 446–458, 2008.

where the word-level netlist is abstracted to an equivalent but smaller gate-level netlist. In the worst case, our analysis generates results that are no larger than the gate-level version of the original design. We tune the runtime of our solution so that it runs very quickly even on industrial size problems, and so that we incur little penalty in the case of netlists that are not amenable to our analysis.

In order to demonstrate the utility of our approach, we analyze an academic high-performance router and two industrial designs (a FIFO and a content addressable memory). We demonstrate significantly reduced netlists, while spending less than a second in the reduction part of our analysis.

2 Preliminaries

We assume a standard frontend flow that compiles problems into netlists by processing a hardware design with properties and constraints into combinational logic over a set of unconstrained inputs I, state variables S, and constants. The top of the resulting forest of combinational logic contain next-state variables S' and single bit outputs O. We assume that (1) the properties we are interested in are all safety properties whose failure is signaled by some output assuming the value `false`, and that (2) each state variable has a known initial state.

In Section 4 we will use an alternate way of describing netlists in terms of three formulas $\mathit{Init}(S)$, $\mathit{Next}(I, S, S')$, and $\mathit{Prop}(S)$. The correspondence between these formulas and the netlist is simple: The $\mathit{Init}(S)$ formula describes the initial states, $\mathit{Next}(I, S, S')$ describes the next-state functions, and $\mathit{Prop}(S)$ captures a particular safety property output function.

3 Netlist Reduction

We use a word-level frontend to compile a model of the given netlist problem. The resulting data is represented as a Directed Acyclic Graph (DAG) of operators over a set of inputs I, state variables S, and binary constant vectors. Every node ϕ in the graph has an associated signal width k; we sometimes annotate nodes with superscripts that denote the size of the bit vector they represent.

The top level nodes of the DAG are partitioned into circuit outputs O and next-state variables S'. Internal nodes in the graph have the following form:

- $\phi^k = \texttt{not}(\alpha^k)$.
- $\phi^k = \texttt{and}(\alpha^k, \beta^k)$.
- $\phi^k = \texttt{arithOp}(\alpha^k, \beta^k)$ for $\texttt{arithOp} \in \{+, -, \ldots\}$.
- $\phi^1 = \texttt{compOp}(\alpha^k, \beta^k)$ for $\texttt{compOp} \in \{<, \leq, =, \neq, \geq, >\}$.
- $\phi^k = \texttt{mux}(\alpha^1, \beta^k, \gamma^k)$.
- $\phi^k = \texttt{extract}(l, \alpha^m)$.
- $\phi^k = \texttt{concat}(\alpha_1^i, \alpha_2^j, \ldots)$.

The `not` and `and` operators are *bitwise* operators in the sense that bit i of the result is generated by applying the boolean operator to bit i of the input nodes.

The `mux` node returns β_k if α_1 is true, and γ_k otherwise. The `extract` node constructs a smaller bit vector by projecting out k bits from position l to $l+k-1$ of its operand. Finally, the `concat` node forms a larger signal by concatenating its operands to form a larger bit vector. Earlier operands in the argument list to `concat` become higher order bits, so $\texttt{concat}(01, 00) = 0100$.

The select signal of `mux` and the output of comparison operator nodes are restricted to have bit width one. Such signals are said to be *bit-level signals*. We refer to signals that are not bit level as *word-level signals*, and use the term *segment* to denote a group of contiguous bits.

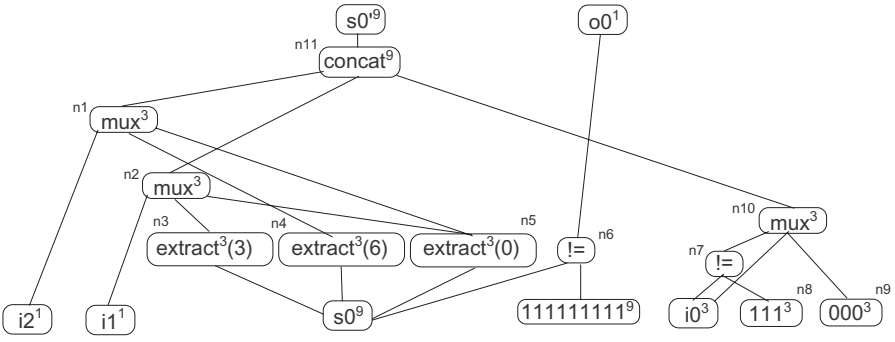

Fig. 1. An example word level netlist

Example 1. Consider the word-level netlist in Figure 1. The initial state for $\texttt{s0}^9$ is 000000000^9, and the property of interest is that $\texttt{o0}^1$ is always true. The circuit generates the next state for `s0` by concatenating three parts. The lowest part is a fresh input value, but only if it is not equal to 111 (otherwise we switch in 000). The other two segments are the result of either keeping the current segment of `s0`, or swapping in the low order segment 0..2 of `s0` depending on the value of two externally controlled inputs. Hence, the system is safe in that the output can never become false. □

Our analysis will be performed in three steps. First, we will rewrite the netlist into a design where the datapath is completely separated from all boolean control logic, and where word-level registers and inputs have been broken up into smaller parts that do not intermingle control and data. Second, we will analyze the datapath portions of the circuit, and find reduced safe sizes for all the word-level entities. Third, we generate a smaller final netlist which can be analyzed by standard gate-level reductions and model checking algorithms.

3.1 Selective Bitblasting

It is easy to see that every word-level netlist can be *bitblasted* into an equivalent bit-level netlist by splitting all variables into single bit segments, and implementing the internal nodes in terms of boolean logic. The result is a netlist where all

signals have width one, and the only internal nodes are the two boolean operators. In the first step of our analysis we will perform *selective bitblasting*, with the aim of ending up with a graph over constants, variable nodes, boolean operators of width one, comparison operators from the set $\{=, \neq\}$, and `mux` operators. All other operators will be removed by a translation into bit-level constructs.

In order to perform this analysis we will annotate each node in the graph with information on which of its segments are treated as word-level *packages*—units of data that are treated uniformly. To do this, we make use of a modification of a data flow algorithm introduced in [6]. We perform the analysis by maintaining a partition of each node in the network into bit segments. For each bit segment of every node, we maintain an equivalence class of other segments of nodes that either depends on it, or that it depends on.

We make use of some primitive operations on equivalence classes and intervals: $\mathit{registerNode}(n)$, $\mathit{split}(n, j)$, $\mathit{mkCompatible}(n_1, n_2, \ldots)$, $\mathit{bitblast}(n)$ and $\mathit{unionNodes}(n_1, n_2, \ldots)$:

- The creation operator, $\mathit{registerNode}(n)$ takes a node, and constructs a singleton equivalence class containing the segment $(0..k-1)$ if the node n has k bits.
- The refining operators *split*, *mkCompatible*, and *bitblast* perform the following functions:
  1. $\mathit{split}(n, j)$ finds the segment equivalence class for node n that contains the bit j. If the bit j falls internally to the segment interval $i \ldots k$ so that $i < j < k$, then the equivalence class is split into two new classes; the first containing the $j - i$ first bits of each segment, and the other containing the remaining bits of each segment.
  2. $\mathit{mkCompatible}(n_1, n_2, \ldots)$ applies the *split* operator to its operands until their segmentations match.
  3. $\mathit{bitblast}(n)$ applies *split* to a node until it is segmented up into one bit slices.
- The merge operator $\mathit{unionNodes}(n_1, n_2, \ldots)$ takes a number of nodes whose segmentation match. If the nodes all have k segments, then the merge operator generates k new equivalence classes by merging the equivalence classes for all first segments of its operands, then merging the equivalence classes for all second segments of its operands, and so on until all k new classes have been formed.

We will also use the following terminology: We say that the segmentation of a signal ϕ^k is *consistent* with the segmentation of another signal ψ^k if all cuts in ψ^k exists in ϕ^k. We *transfer* segments from ϕ^k to ψ^k by using *split* to introduce cuts in ψ^k at all positions where there are cuts in ϕ^k.

Our dataflow analysis is performed in a depth first recursive manor. Each node ϕ_k encountered is registered using *registerNode*, and then processed as follows:

- ϕ_k is a constant: Use *split* to partition the node into maximal segments of consecutive bits of the form $00\ldots0$ and $11\ldots1$. The constant 000100 generates the segments $(0, 1)$,$(2, 2)$, and $(3, 5)$.

- ϕ^k is a variable: Do nothing.
- $\phi^k = \texttt{not}(\alpha^k)$: Bitblast ϕ^k and α^k, and perform $unionNodes(\phi^k, \alpha^k)$.
- $\phi^k = \texttt{and}(\alpha^k, \beta^k)$: Bitblast ϕ^k, α^k, and β^k. Perform $unionNodes(\phi^k, \alpha^k, \beta^k)$.
- $\phi^k = \texttt{arithOp}(\alpha^k, \beta^k)$: Bitblast ϕ^k, α^k, and β^k. Perform $unionNodes(\phi^k, \alpha^k, \beta^k)$.
- $\phi^1 = \texttt{compOp}(\alpha^k, \beta^k)$ and
  - $\texttt{compOp} \in \{=, \neq\}$: Perform $mkCompatible(\alpha^k, \beta^k)$, and $unionNodes(\alpha^k, \beta^k)$.
  - $\texttt{compOp} \in \{<, \leq, >, \geq\}$: Bitblast α^k and β^k. $unionNodes(\alpha^k, \beta^k)$.
- $\phi^k = \texttt{mux}(\alpha^1, \beta^k, \gamma^k)$: Perform $mkCompatible(\phi^k, \beta^k, \gamma^k)$ and $unionNodes(\phi^k, \beta^k, \gamma^k)$.
- $\phi^k = \texttt{extract}(l, \alpha^m)$: Use *split* to introduce cuts at bit l and $l+k$ for α^m. Next, transfer all segment cuts in the region between bit l and $l+k-1$ from α^m to the corresponding positions in ϕ^k, and union the corresponding segments of ϕ^k and α^m.
- $\phi^k = \texttt{concat}(\alpha^i_1, \alpha^j_2, \ldots)$: Segment ϕ^k to match the operand borders. Then transfer the internal segment cuts from the operands to the corresponding points in ϕ^k, and union the corresponding segments.

When all nodes have been processed, we traverse all present-state and next-state pairs (ϕ^k, ϕ'^k) and perform $mkCompatible(\phi^k, \phi'^k)$ and $unionNodes(\phi^k, \phi'^k)$. We also use *split* to ensure that the segmentation of each current-state node is consistent with the segmentation of its initial state.

Example 2. Consider the verification problem from Example 1. Assume we traverse the netlist by first visiting `s0`. This creates the partition information `s0` : (0..8) in a singleton equivalence class. After visiting nodes `n3`, `n4`, `n5`, `i1`, `n2`, `i2` and `n1` we have the new segmentation `s0` : (0..2), (3..5), (6..8). The equivalence class of `s0` : (0..2), now contain the other elements `n1` : (0..2), `n2` : (0..2), `n3` : (0..2), `n4` : (0..2), `n5` : (0..2), `s0` : (3..5) and `s0` : (6..8). □

After we have performed the data flow analysis, we will have segment information for each node, and we are assured that (1) the segmentation of current and next-state variables is consistent, (2) the segmentation of current-state variables and initial-state variables is consistent, and (3) the segment sources of our netlist DAG of size greater than one will only be propagated through multiplexor networks or be compared using the operators $\{=, \neq\}$.

We can now create a modified word-level netlist from the bottom up, by converting each node in the original netlist into a list of new nodes (one per segment). We do this as follows. If a variable or constant node has n segments, we generate a list of n fresh nodes of appropriate type and size. Nodes of type `not`, `and`, `arithOp` or comparison operators from the set $\{<, \leq, >, \geq\}$ are guaranteed to have been bitblasted, so we just return the list of signals corresponding to the bit-level implementation of the operator in terms of its inputs. Any remaining node ϕ^k is handled as follows:

- $\phi^1 = \texttt{compOp}(\alpha^k, \beta^k)$ and $\texttt{compOp} \in \{=, \neq\}$: implement ϕ^1 as a boolean network of equalities over the respective segments.

- $\phi^k = \texttt{mux}(\alpha^1, \beta^k, \gamma^k)$: generate a list of multiplexors $\texttt{mux}(\alpha^1, x, y)$. Each multiplexor takes x and y to be the resulting segment nodes generated for the corresponding position in β^k and γ^k.
- $\phi^k = \texttt{extract}(l, \alpha^m)$: Project out the list of new nodes generated for the desired interval segments of α^m.
- $\phi^k = \texttt{concat}(\alpha_1^i, \alpha_2^j, \ldots)$: Return the concatenation of the lists of new nodes generated for the operands.

Example 3. Assume that a node $\texttt{mux}^{32}(\texttt{n1}^1, \texttt{n2}^{32}, \texttt{n3}^{32})$ has been segmented as $(0..7), (8..31)$, and that the result of reimplementing $\texttt{n1}^1$ was $[\texttt{m1}^1]$, and that the result of reimplementing $\texttt{n2}^{32}$ and $\texttt{n3}^{32}$ was $[\texttt{m2}^{24}, \texttt{m3}^8]$ and $[\texttt{m4}^{24}, \texttt{m5}^8]$, respectively. Then we return

$$[\texttt{mux}^{24}(\texttt{m1}^1, \texttt{m2}^{24}, \texttt{m4}^{24}), \texttt{mux}^8(\texttt{m1}^1, \texttt{m3}^8, \texttt{m5}^8)]$$

□

When we reach the top of the new DAG, we create one new next-state variable or output per segment from the list of implementations of the feeders.

The signals in the resulting netlist graph come in two different flavors. The first type of signal has bitwidth one, and are thus bit-level signals. These signals get processed using standard boolean logic. The second type has bitwidth greater than one. These word-level signals are moved around using multiplexor networks, and generate bit-level signals using comparison operators. Note that an original input or state variable in the design may very well be split up into several parts, some of which are bit level, and some which are word level.

Theorem 1. *The selective bitblasted netlist is equivalent to the original netlist.*

Proof. Our segment analysis is the formula analysis in [6], modified to bitblast more operators, and force consistent segmentation of current-state variables, next-state variables, and initial-state values. Correctness follows from the correctness of [6], together with an induction over time. Due to limited space we omit the detailed proof.

3.2 Abstraction of Word-Level Variables

The selectively bitblasted netlist now has two components: (1) A word-level component that reads packages from the inputs and word-level registers, moves them around using multiplexors, and performs package comparisons. (2) A bit-level component that reads bit-level signals from the inputs, controls the multiplexors (possibly based on the outputs from comparison operators), and computes bit-level outputs.

As the word-level variables are only compared for equality and inequality and moved around, it seems like we should be able to abstract them somehow. This is indeed true. In a 1995 paper, Hojati and Brayton introduce a reduction for designs they refer to as *Data Comparison Controllers* (DCCs) [4]. These designs

are partitioned into a boolean part and a datapath part that manipulates infinite packets modeled as integers by moving them around and comparing them, just like our selectively bitblasted designs. It is shown in Hojati and Brayton's paper that for every DCC, there always exists a finite smallest package size that preserve the status of the properties of the design. In fact, if the system has N infinite integer variables and M integer constant nodes, the integers can safely be modeled using length $S_{min} = \lceil \log_2(N + M) \rceil$ bit vectors.

Unfortunately we can not apply this result directly, for two different reasons:

1. Our packages do not have infinite initial size.
2. We have more than one package size.

Let us first deal with issue 1, and momentarily assume that we have a system where all word-level variables have a single bitwidth S.

Lemma 1. *The DCC sizing theorem from [4] can be restated as follows: Any package size $S \geq S_{min}$ bits gives the same status to all design properties.*

Proof. Reducing the size of one or more variable domains can only reduce the number of design behaviors. The number of provable properties will hence grow monotonically. As the DCC sizing theorem implies that exactly the same properties are provable in the infinite case as for package size S_{min}, finite package sizes larger than S_{min} proves exactly the same properties. □

As a result of Lemma 1, as long as the initial package size S is larger than or equal to S_{min}, size S_{min} packages will preserve the status of all properties. However, if the original package size is *smaller* than S_{min}, we have no guarantees about what will happen if we size them up. In order to circumvent this problem, we choose $S_{new} = \min(S, S_{min})$, which will guarantee safety regardless of initial package size. Also note that in our case, M (the number of constants of a particular length) always equals two, due to our partitioning of constants.

In order to solve the second issue, one potential solution could be to choose our S_{min} based on the size of the largest package in the system. This would clearly be conservative. However, this is not necessary: After our selective bitblasting, the resulting netlist has no facility for converting a size N word-level segment into some other size segment. Segments of a different width can hence not be compared, or registered in the same word-level register slices. Our converted designs are therefore *generalized DCCs*, with one bit-level component, and finite number of separate word-level components that only communicate with each other using bit-level signals. By iterating the argument in [4] it is easy to show that we can abstract each of these word-level components individually. By combining this fact with Lemma 1 we arrive at our master theorem:

Theorem 2. *For each S, assume that there exists N_S state variable and input segments of size S. The status of all properties of our selectively bitblasted design D are then preserved if we resize size S segments to have* $\min(S, \lceil \log_2(N_S + 2) \rceil)$ *bits.*

Example 4. Again consider the verification problem from Example 1. Our selective bit blasting generated the final segmentation `s0` : (0..2), (3..5), (6..9). Together with the input segment `i0` : (0..2), there are a total of four segments of size three. A safe reduced bit size for these segments is hence $\min(3, \lceil \log_2(4+2) \rceil) = 3$ bits. In this case we can not perform any reduction. However, note that any larger size system with the same structure could safely be scaled down to represent `s0` with nine bits. □

The next and final step of our analysis is to traverse all state variables of the selectively bitblasted netlist and compute a tally of segment size populations. For each segment size, we compute a new reduced size using Theorem 2. When we have sized all word-level state variables and constants, we compute our abstracted netlist by rewriting the word-level component of the selectively bitblasted design to use variables and constants of the new correct size, and adjusting the width of the internal operators.

4 Impossibility Results

The reduction we presented in Section 3 forces the word-level parts of the design to only contain multiplexors and negated and unnegated equality comparisons. Could this be extended to allow inequalities, bit-parallel boolean operators, or arithmetic? The answer, as we will see, is no.

Let us first investigate the extension of DCCs to allow inequality comparisons between signals.

Theorem 3. *If we allow inequality comparisons between word-level variables, there exists a system whose smallest reduction is the system itself for any bitwidth N. We can hence not find a static safe package size based only on the number of word-level state variables.*

Proof. Take the system to be constructed from the width N state variable v and input i, and the output o. Let $Init(v) \equiv 11\ldots1$, $Next(v) \equiv \texttt{mux}(i < v, i, 00\ldots0)$, and $Prop(v) \equiv v \neq 00\ldots0$. For a given bitwidth N this system has a trace where it takes $2^N - 1$ steps before the property output becomes zero. No smaller bitwidth will preserve this trace. □

However, it *is* safe to allow inequality comparisons where one operator is a constant. To see this, realize that the only constant segments that exist after our analysis have the form $00\ldots0$ and $11\ldots1$. The comparison can hence be implemented as a boolean network whose leaves only contain inequality comparisons between variable segments and $00\ldots0$ and $11\ldots1$. These leaves, in turn can be rewritten in terms of equality operators (for example $x_k < 00\ldots0_k$ is equivalent to `false` and $x_k < 11\ldots1_k$ is equivalent to $x_k \neq 11\ldots1_k$. The resulting transformed system is a DCC, so our main theorem applies.

It is also unsafe to allow the use of bit-parallel operators in the word level partition:

Theorem 4. *If we allow bitwise boolean operators in our word-level component, there exists a system whose smallest reduction is the system itself for any bitwidth N. We can hence not find a static safe package size based only on the number of word-level state variables.*

Proof. Take the system to be constructed from the width N state variable v and input i, and the output o. Let $Init(v) \equiv 11\ldots1$, $Next(v) \equiv \texttt{mux}(v \neq \texttt{and}(i,v), \texttt{and}(i,v), 00\ldots0)$, and $Prop(v) \equiv v \neq 00\ldots0$. For a given bitwidth N this system has a trace where it takes $N-1$ steps before the property output becomes zero. No smaller bitwidth will preserve this trace. □

Finally, we could imagine allowing the use of arithmetic nodes in our word-level machinery. Again, this would not be a good idea:

Theorem 5. *If we allow arithmetic operators in our word-level partition, there exists a system whose smallest reduction is the system itself for any bitwidth N. We can hence not find a static safe package size based only on the number of word-level state variables.*

Proof. Take the system to be the system containing a single state variable v, a single input i and the output o. Let $Init(v) \equiv 00\ldots0$, $Next(v) \equiv v+1$, and $Prop(v) \equiv v \neq 11\ldots1$. For a given bitwidth N this system has a trace where it takes 2^N-1 steps before the property output becomes zero. No smaller bitwidth will preserve this trace. □

It is easy to modify the arguments in our impossibility theorems to work for a system with some single other inequality between variables, a single `or` or `xor` operator, or a single other arithmetic operator. In essence, our analysis hence allows the richest word-level components possible, while still allowing a static analysis.

5 Implementing the Reduction

Our chief concern in implementing our formula reduction is to make the generation of the reduced system as fast as possible. The worst case for the reduction is that the design becomes completely bitblasted, which in our flow would be a necessary precondition of further processing anyway. As long as we tune our reduction to take a very small amount of time, we can hence always apply it safely.

In order to make the reduction as fast as possible, we must choose effective data structures for the signal segment information and the equivalence class information.

We maintain the segmentation information using *skip lists* [9], which is a probabilistic data structure that allow $\log(N)$ average case insertion and deletions of a set of ordered elements, with a better constant factor than many balanced tree implementations. Each segment in the skip list is indexed by the start of the segment. In each segment we store equivalence set information using standard union-find algorithms [3], which allows equivalence class operations in close to constant amortized time.

In our implementation, each splitting of a segment equivalence class takes a linear number of equivalence class and skip list operations in the size of the class. There are systems with N nodes with word-level variables of width bounded by M where every variable gets partitioned into one bit segments, and the equivalence classes contain every signal in the system, so that $O(N * M)$ equivalence set and skip list operations are necessary. However, our experimental experience is that with the efficient equivalence class and segment maintenance algorithms, processing of industrial netlists is in practice not noticeably slower than a linear node traversal.

There are several important practical implementation details that affect the reduction strength of our analysis. The most important detail in our implementation has to do with constant sharing: Any reasonable word-level DAG representation for the netlists use hashing to share nodes maximally. This is a good thing, but complicates things for our analysis in the case of constants: Constants with multiple fanout force the merging of some equivalence classes that otherwise would have been kept separate (if two logic cones only share a constant, segmentation propagation from one to the other is unnecessary). In our implementation, we work around this issue by introducing fresh variable nodes on the fly for each reference to a constant. At the end of the analysis, these variables gets transformed back into constant nodes.

Another improvement we have found empirically important is to preprocess the representation we get from the HDL frontend to provide an optimum starting point for our analysis. The reason for this is that in certain designs, some or all of the logic that moves packets is implemented by instantiating bit-level modules for each packet bit. This means that words coming in get broken up into individual bits, moved around in a uniform way, and then recombined into words. Such designs will fragment segments into bit-level signals and weaken the results of our reduction. In our experimental results, the high-performance router [8] is such a design. Without this preprocessing no netlist reduction is possible. We avoid this problem by sweeping the initial netlists and detecting subgraphs where words are split up, routed, and recombined. Each such subgraph is automatically reimplemented on the word level.

In order to cope with industrial designs, we have also implemented some extensions to the techniques presented in Section 3. Notably, we handle symbolic memories with abstract read and write nodes.

6 Related Work

As can be seen in Section 3, our reduction leverages a dataflow analysis for extracting a DCC from a general word-level netlist. The basic approach used in this analysis was introduced in [6], where it was used to perform a formula reduction. In order to apply similar techniques on the netlist level, we have had to change the analysis to include partitioning from initial states, and to make sure that the current and next-state variable segmentation corresponds. Moreover, due to the impossibility results presented in Section 4, we have to bitblast the bit-parallel boolean operators and inequality operators which are kept in [6].

Rather than automatically compute the segmentation of signals into word level and bit-level parts, we could imagine that we could let the user annotate the design with this information. As we can compute the necessary segmentation quickly and accurately, we believe that this is a bad proposition; in our experience, any piece of extra data that a user has to provide decrease the utility of a given algorithm in industrial tools.

Several different word-level formula solvers such as `UCLID` [2] and `BAT` [7] have been used for hardware verification. These procedures can in theory be used as a foundation for induction or interpolation-based unbounded property verification, but they do not provide a general way to leverage an arbitrary bit-level model checker as a back-end decision oracle. Moreover, performing the word-level reduction on netlists rather than on formulas has several benefits even if we are focused on bounded checks only: First of all, there is no need to re-abstract a newly unfolded formula every time the bound is increased. Second, by performing the reduction once and for all, we can apply standard performance optimizations in the back-end SAT solver such as incremental solving and the reuse of conflict clauses without having to resort to special tricks.

In terms of netlist reductions, the most closely related work we are aware of is the DCC reduction work [4] we make use of in our analysis. The chief problem with this work as it stands is that it requires the design to be partitioned up into a legal DCC to begin with, and this is unlikely to be true in the case of many systems where control data is integrated into the same packets as the word-level payload (the router we analyze in Section 7 is one such design). We solve this problem using our selective bitblasting approach. We also modify the DCC analysis to work with finite packets, and extend it so that it can deal with several different package sizes in the same design.

There are approaches to data type reduction that have some similarities to the DCC analysis, such as symmetry reduction based on the use of *scalarsets* [5]. For these type of reductions, it is typically up to the user to manually introduce reducible data types when modeling a given design, which would be very time consuming for industrial size netlists that intermingle control and data information. Moreover, these reductions are generally not straight netlist transformations, so they would be hard to use in a transformation-based verification environment.

7 Experimental Results

Our aim with this work is to provide a word-level reduction method for a transformation-based verification system. We are hence focusing on simplifying the netlist problem as much as possible, and providing output that can be processed with an interleaving of model checking methods and simplification procedures like retiming, rewriting and reparametrization. We thus strive for maximal reduction at minimal runtime cost.

We present results on three designs, which represent classes of problems where our customers have found that traditional model checking provides little or no

traction today. In all of our experiments, we use standard SAT-based bounded model checking and BDD-based reachability analysis as our back-end engines.

The first benchmark is an industrial FIFO memory. We check that if a slot has been written and not overwritten, the correct data is read out when the entry is popped. We check two version of this design. The first version has 16 slots, each 16 bits wide. Before this reduction, this design can not be proved correct in one hour of compute time using BDD-based model checking. After the reduction, this design is proved correct in less than one minute. The second version has 75 slots, each 32 bits wide. Before the reduction the design has 2594 registers, which is reduced to 646 registers after analysis. The reduction takes less than .5 seconds of compute time. Bounded model checking of the unreduced design for 18 cycles takes 45 minutes of CPU time, whereas it takes 180 seconds after the reduction.

The second benchmark is an industrial Content Addressable Memory (CAM) with three data ports and 48 slots each containing 20 bits of data. We are checking that if a piece of data has been written to some particular slot in the CAM, and has not been overwritten, the data is reported as existing in the CAM. The original design has 1111 registers, which is reduced to 383 registers post reduction. The transformation takes less than .5 seconds of compute time. A bounded check of depth eight takes approximately 60000 seconds before the reduction, compared to 1780 seconds after the reduction.

Our final benchmark is a pipelined high-performance router created at Stanford [8]. The router has four port connections to adjacent routers plus an inject node and an eject node. The router's main crossbar is implemented on the bit level. It has two virtual channels per port, and moves packets broken up into units called *flits* that it reads from the environment ports each cycle. Each flit contains destination data, type field control information, and data payload packed into 32 bits. We check the partial correctness property that a packet that is injected on a port when the router is in an neutral state appears on the correct output port within a predetermined time. The word-level netlist contains 7516 registers before the abstraction, and is reduced to 4816 registers after the abstraction. The total reduction analysis time is less than one second.

8 Conclusions

In this paper, we have introduced a word-level model checking approach aimed at unbounded property checking for industrial netlists. Our approach is based on a two-step method, where a quick analysis rewrites the netlist into a design where the word-level datapath that manipulates packages is completely separated from the boolean control logic. We then resize all packages using statically computed safe lower bounds that guarantee that we preserve the properties being checked. The resulting system can be analyzed using any standard bit-level model checking technique, or further processed using transformational verification simplifications.

Our contributions include the combination of a modified word-level extraction algorithm previously only used on formulas [6] with some modified lower bound

computation theory [4]. We have also showed that our analysis is tight in the sense that the obvious extensions of operators allowed in the word-level part of the circuit all preclude a static analysis. Finally, we provided key insights into how to implement the algorithms efficiently, and demonstrated the utility of our reduction on a number of industrial designs.

Acknowledgments. Many thanks to Tamir Heyman, who participated in discussions and helped with the work necessary to integrate our analysis into the frontend flow.

References

1. Baumgartner, J., Gloekler, T., Shanmugam, D., Seigler, R., Huben, G.V., Mony, H., Roessler, P., Ramanandray, B.: Enabling large-scale pervasive logic verification through multi-algorithmic formal reasoning. In: Proc. of the Formal Methods in CAD Conf. (2006)
2. Bryant, R.E., Lahiri, S.K., Seshia, S.A.: Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 78–92. Springer, Heidelberg (2002)
3. Galler, B., Fischer, M.: An improved equivalence algorithm. Communications of the ACM (May 1964)
4. Hojati, R., Brayton, R.: Automatic datapath abstraction in hardware systems. In: Wolper, P. (ed.) CAV 1995. LNCS, vol. 939, pp. 98–113. Springer, Heidelberg (1995)
5. Ip, C.N., Dill, D.L.: Better verification through symmetry. Formal Methods in System Design (August 1996)
6. Johannesen, P.: Speeding up hardware verification by automated data path scaling. PhD thesis, Christian-Albrechts-Universität zu Kiel (2002)
7. Manolios, P., Srinivasan, S.K., Vroon, D.: BAT: The Bit-Level Analysis Tool. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 303–306. Springer, Heidelberg (2007)
8. Peh, L.-S., Dally, W.: A delay model and speculative architecture for pipelined routers. In: Proc. Intl. Symposium on High-Performance Computer Architecture (2001)
9. Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. Communications of the ACM (June 1990)
10. Ranise, S., Tinelli, C.: Satisfiability modulo theories. Trends and Controversies - IEEE Intelligent Systems Magazine (December 2006)