Word-Level Sequential Memory Abstraction for Model Checking

Per Bjesse
Advanced Technology Group
Synopsys Inc.

Abstract—**Many designs intermingle large memories with wide data paths and nontrivial control. Verifying such systems is challenging, and users often get little traction when applying model checking to decide full or partial end-to-end correctness of such designs. Interestingly, a subclass of these systems can be proven correct by reasoning only about a small number of the memory entries at a limited number of time points. In this paper, we leverage this fact to abstract certain memories in a sound way, and we demonstrate how our memory abstraction coupled with an abstraction refinement algorithm can be used to prove correctness properties for three challenging designs from industry and academia. Key features of our approach are that we operate on standard safety property verification problems, that we proceed completely automatically without any need for abstraction hints, that we can use any bit-level model checker as a back-end decision procedure, and that our algorithms fit seamlessly into a standard transformational verification paradigm.**

I. Introduction

Bit-level model checking methods have for a long time been the standard approach for safety property verification of hardware systems. However, recently there has been a surge of interest in methods that model systems at the *word level*. Such methods differs from traditional bit-level methods by not forcing a fragmentation of a netlist into individual single bit inputs and registers, and by modeling data paths at the level of complex gates such as adders and multipliers. By retaining some of the high-level structure of the hardware systems that are analyzed, and by leveraging decision procedures for the richer word-level logic, the solving process can sometimes be sped up dramatically. Prime examples of such decision procedures include SMT-solvers [14], and approaches based on reduction to satisfiability checking like `UCLID` [5] and `BAT` [11]. These methods are all *formula-based* approaches in that they decide unquantified formulas in some logic, and as such require the verification problem for unbounded safety properties to be reduced to a number of bounded time formula checks.

One of the largest stumbling blocks for traditional model checking is the presence of large memories intermingled with complex control logic. This will typically result in very hard or intractable model checking problems. However, proving the correctness of these systems may require reasoning about much smaller number of memory entries.

In this paper, we present an analysis that leverages this fact together with word-level symbolic memory information to abstract model checking problems with large memories to problems that contain fewer memory slots. Our analysis is specifically geared to proving properties, but it could potentially be useful for finding bugs as well.

We have three prime design requirements for our analysis: (1) we do not want the user to have to provide any annotations or follow some particular design style for our analysis to be applicable, (2) we do not want to commit to the use of some particular model checking technology such as the use of SMT-solvers or the use of a specialized logic, and (3) we want our abstraction to be a computationally cheap netlist-to-netlist transformation so that we allow the use of the standard algorithms for iterated simplification and property checking in a *transformation-based verification environment* [3].

Our design requirements have several important implications. First of all, as we want our approach to be independent of the back-end model checking technology, our transformation will be orthogonal to the standard formula-based word-level approaches. Second, by leveraging word-level information to abstract the problem on the netlist level rather than changing the model checking procedure itself, we will be able to benefit from the differing characteristics of any of the multitude of model checking procedures available today, including both methods based on BDD-based fixpoint computations and SAT-based methods.

Our main idea is to abstract the design by remodeling memories to only represent a fixed, smaller number of memory slots. Writes to slots that are not represented are dropped by our abstracted memory, and reads to unrepresented slots return nondeterministic values. As the abstracted memories can produce all traces that the original memories can generate, our transformation is sound (but not necessarily complete).

In order to automatically come up with the slots we want to represent, we will be using an abstraction refinement loop, where counterexamples for the abstracted system are analyzed to determine what slots that need representing currently are missing.

II. Preliminaries

In the remainder of this presentation, we will be concerned with checking safety properties of synchronous hardware. We assume that a standard modeling of designs are used where all properties and constraints have been compiled into the netlist. All inputs to the netlist are thus unconstrained. The failure of some property is signaled by a dedicated single bit output *safe* assuming the value FALSE. A proof of safety thus needs to demonstrate that no input trace from an initial state of the netlist can reach a state where $safe$ fails.

The representation that we operate on is a word-level netlist Directed Acyclic Graph (DAG) representation. We obtain the DAG representation of a verification problem using a standard front-end compilation process that utilizes a Hardware Description Language (HDL) compiler that retains as much word-level information as possible. Our representation is rich enough to represent full synthesizable System Verilog.

All nodes in the DAG representation represents a bitvector of some fixed width k. We sometimes annotate nodes with superscripts denoting their width; $\texttt{sig}^4$ signifies a four bit signal called `sig`.

The sources in the DAG are input variables, current state register variables, and bitvector constants. The sinks of the graph are the one-bit output $safe$ together with the next-state registers nodes. If the current-state node for a register is named `r`, we follow the naming convention that the next-state node will be named $\texttt{r}'$. Each current state register has some particular ternary bitvector as its initial state; these vectors characterize the set of initial state bitvectors for the register as the set of vectors resulting from all possible instantiations of the X-values.

The computational semantics of our netlist problems is very simple. The netlist is implicitly clocked: one time instance of computation propagates values from the current state registers and inputs, and generates next state values and a value for the output $safe$. Our model checking problem consists of deciding whether there exists some binary initial state and sequence of input vectors that will take the design in zero or more steps to some state where $safe$ assumes the value FALSE.

The internal nodes in our graph representation for a design verification problem have the form

- $\texttt{nd}^k = \texttt{op}_l(\texttt{op1}^i, \texttt{op2}^j, \ldots)$
- $\texttt{nd}^k = \texttt{extract}(k, l, \texttt{op}^m)$
- $\texttt{nd}^k = \texttt{concat}(\texttt{op1}^i, \texttt{op2}^j, \ldots)$
- $\texttt{nd}^k = \texttt{mux}(\texttt{cond}^1, \texttt{optrue}^k, \texttt{opfalse}^k)$
- $\texttt{nd}^k = \texttt{read}(\texttt{op}^i, \texttt{addr}^j)$
- $\texttt{nd}^k = \texttt{write}(\texttt{op}^k, \texttt{addr}^i, \texttt{data}^j)$

The `op` nodes in the network are particular combinational functions of their inputs. The function computed by these nodes range from boolean operators and comparison operators to arithmetic functions such as $+$ or $*$. The `extract` nodes project out k bits from the bitvector `op` starting at bit l, and the `concat` node concatenates its operands into a larger signal. The `mux` node has a single bit node `cond` as input that selects whether to return `optrue` or `opfalse`.

The nodes `read` and `write` are used for modeling memories. Their semantics are simple. If a `read` node has width w, it returns the result of projecting out the bits $[\texttt{addr}*w \ldots (\texttt{addr}+1)*w-1]$ from the bitvector `op`; if a write node's `data` operand has width w, it returns the bitvector that would result from overwriting the region $[\texttt{addr} * w \ldots (\texttt{addr}+1) * w - 1]$ of the bitvector `op` with `data`. The address space of read and write nodes is not restricted in any particular fashion; out-of-bounds reads return nondeterministic values for nonexisting bits, whereas out-of-bounds writes do nothing. Also note that there are no dedicated special register nodes that represent large memories, or restrictions on what signals read and write nodes can be applied to: RT-level memories are compiled to bitvector state registers that are no different than the registers used for modeling other design artifacts. By appropriate use of control logic together with multiple read and write nodes, the DAG representation supports arbitrarily complex memory interfaces with large number of read and write ports. Moreover, by nesting reads, writes, and other nodes, we can implement complex policies on update and read orders resulting from same-cycle reads and writes.

III. A MOTIVATING EXAMPLE

As an example, consider the netlist problem in Figure 1. This example has three word-level inputs (`raddr`, `waddr` and

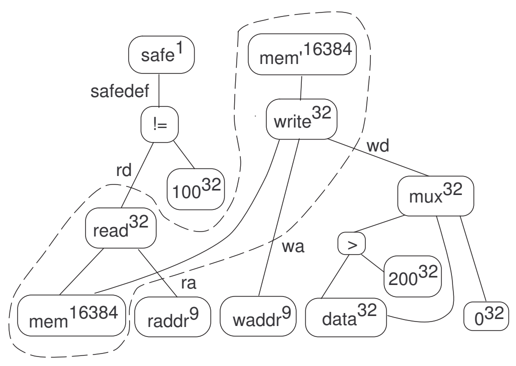

Fig. 1. A simple word-level netlist verification problem.

`data`), and one word level register `mem` which is initialized to the constant 0^{16384}. At each clock cycle the system reads the content of address `raddr` from `mem`. It also writes the input `data` to the address `waddr` if `data` is greater than 200; otherwise it writes 0. The property we check is that the value read from `mem` never is equal to 100. Clearly, this is true for any execution trace for the system. Moreover, as we will see, we can prove this statement by just reasoning about the contents over time of a single slot of the memory—the last slot read.

The circuit modeled in Figure 1 can conceptually be partitioned into two parts. The first part contains the large memory `mem`, and communicates with the rest of the design through two inputs and two outputs: A write address port `wa`, a write data port `wd`, a read address port `ra`, and a read data port `rd`. This part of the design resides inside the dashed line in Figure 1. The second part is the rest of the design.

We can abstract the memory part by removing the original implementation inside the dashed line, and instead instantiating a much simpler memory that contains two registers, $\texttt{sel}^9$ and $\texttt{cont}^{32}$ (see Figure 2). The `cont` register will represent the content of one memory slot selected by the `sel` register. The slot selected by `sel` is chosen during initialization of the circuit, and stays the same during the subsequent system

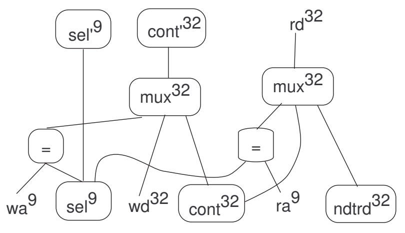

Fig. 2. Reimplementation of the memory part of the netlist from Figure 1.

execution. Specifically, in the new abstracted memory implementation in Figure 2, the register $\texttt{sel}^9$ has an unconstrained initial state, and a next state function that just propagates the current state value to the next state. The register `cont` is initialized to the value zero (remember: all slots in `mem` are zero initialized). In the new memory implementation, reads from address `raddr` return the value of `cont` if $\texttt{sel} = \texttt{raddr}$, otherwise they return a value read from the environment on an unconstrained input `ndtread`. Writes to the memory updates `cont` if $\texttt{sel} = \texttt{waddr}$, otherwise it retains its value.

The new memory implementation overapproximates the original memory as every I/O trace for the original partition can be generated by the modified memory. Every netlist resulting from substituting this simpler memory implementation for the original memory subsystem will hence overapproximate the original netlist.

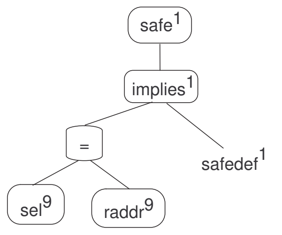

Fig. 3. The modified *safe* output .

Unfortunately, if we attempt to prove the modified system correct, we will immediately get a spurious counterexample where (1) the value of `raddr` in the final cycle is different from the value chosen as the initial value for `sel` and (2) `ndtread` has the value 100. The slot that was chosen for representation by the initialization of `sel` is hence not in sync with the address that is read in the counter example, and an erroneous value is read from the environment.

We can remove this spurious trace by changing the definition of correctness so that we only check the safety of the system when the address that is read the current clock cycle is the one that is represented. A simple way of doing this is to rewrite the combinational output signal *safe* from $\texttt{read}(\texttt{mem}, \texttt{raddr}) \neq 100$ to the signal $\texttt{sel} = \texttt{raddr} \rightarrow \texttt{read}(\texttt{mem}, \texttt{raddr}) \neq 100$, as shown in Figure 3. The resulting modified system N_{abs} is easily shown to be safe.

While it may not be immediately obvious, our remodeling of the memory together with the revised notion of correctness captured by our rewriting of the *safe* output cone is sound: To see this, we can view our rewritten netlist as having been produced in two stages.

In the first stage, we produce a netlist N' identical to the original netlist N with the exception that we have added the `sel` register to the netlist, and rewritten the *safe* output only. The resulting netlist is a sound abstraction of N: Assume that there exists a trace of the original system where at some execution time point $\texttt{read}(\texttt{mem}, \texttt{raddr})$ is 100 for some particular value v of `raddr`. Then the run of N' that is identical to the original trace, but initializes `sel` to v forces the modified safety output $\texttt{sel} = \texttt{raddr} \rightarrow \texttt{read}(\texttt{mem}, \texttt{raddr}) \neq 100$ to fail in the last step of the trace too (remember, we have not changed the memory implementation; we have only changed the property and added one register). As all counterexamples are preserved, this first stage of the transformation is hence sound.

In the second stage, we rewrite the memory in the netlist N' produced by the first transformation so that it only represents the slot chosen by `sel`. This system overapproximates N' as the reimplementation of the memory subsystem simulates the original memory. The second transformation hence also retains soundness, and any proof for the resulting system will guarantee safety of the original netlist N.

In order to systematize the technique used to solve the example in Figure 1, we need to solve three different problems: (1) we need to automatically extract abstractable memories from our netlist, (2) we need to substitute simpler, lossy, memory implementations for the original memories, and (3) we need to come up with an automatic way to extract information on what slots we need to represent in order to infer correctness. Next, we show how these problems can be solved.

IV. Identification of Remodellable Memories

The first part of our analysis will be to scan the DAG under analysis for what we term *remodellable memories*. These are memories that we can abstract. Informally, remodellable memories are addressed in a uniform way, only communicate with the rest of the design in such a way that another implementation easily can be substituted, and have a next state function of a particular simple structure. (Our implementation handle several simple generalizations of remodellable memories, as outlined in Section VI, but we will not delve into these generalizations here in order to keep the exposition simple.)

In order to formally define our notion of a remodellable memory, let us first introduce some notation. Given a register variable `mem`, the set of *pure memory nodes* for `mem` are recursively defined as follows:

- the node `mem` is a pure memory node for `mem`.
- $\texttt{write}(\texttt{op}^k, \texttt{addr}^i, \texttt{data}^j)$ is a pure memory node for `mem` if $\texttt{op}^k$ is a pure memory node for `mem`.
- $\texttt{mux}(\texttt{cond}^1, \texttt{optrue}^k, \texttt{opfalse}^k)$ is a pure memory node for `mem` if $\texttt{optrue}^k$ and $\texttt{opfalse}^k$ are pure memory nodes for `mem`.

The set of *pure read nodes* for a state variable mem is comprised of all netlist read nodes $\texttt{read}(\texttt{op}^i, \texttt{addr}^j)$ for which $\texttt{op}^i$ is a pure memory node for mem. Analogously, the set of *pure write nodes* for mem is comprised by all netlist nodes $\texttt{write}(\texttt{op}^k, \texttt{addr}^i, \texttt{data}^j)$ for which $\texttt{op}^k$ is a pure memory node for mem.

We can now define a remodellable memory to be a register state variable mem that fulfills the following properties:

1) all read and write nodes that have mem in their support read and write data of the same width w using address nodes of the same width a. Moreover, the bitwidth of mem is an integer multiple m of w, and $2^a \leq m * w$ so that all memory accesses occur inside the memory.
2) mem is either initialized to the boolean constant $000\ldots0$ or $111\ldots1$. (We lift this requirement in our implementation, as outlined in Section VI.)
3) the next state function for mem is a pure memory node for mem, and no other next state function is a pure memory node for mem.
4) every fanout path from mem is made up of a sequence of pure memory nodes terminating either in (1) the next state node for mem, or (2) a pure read node.

Requirement 1 ensures that the memory is treated as a bitvector of slots that are read and written in a uniform way. Requirement 2 ensures that all of these slots have the same initial state, which guarantees that the selected slots we will model all have the same initial state. The remaining requirements ensure that the memory register only occur in the fanin of other state registers and outputs through read nodes, and that the next-state function for the memory is a simple multiplexor tree that chooses between different write nodes updating the memory.

Given a netlist problem encoded as a word-level DAG, we use straight forward linear traversal algorithm to extract the set of remodellable memories and compute their associated sets of read and write nodes. Note that while some memories that could be used in a design are not remodellable as per our definition, our experience is that the majority of large memories used in datapath computations in practical hardware designs fall into this category. Our definition of an remodellable memory seem to provide a good balance between being able to cover most interesting memory implementations, while being simple enough to provide relatively straight forward memory abstraction algorithms.

V. Memory Abstraction

A. Remodellable Memory Abstraction

We now systematize the approach used in our motivating example.

In the case of the design from Section III, we abstracted the memory over the current value of $\texttt{raddr}^9$. While this worked well for this particular system, for many systems, memory accesses from a number of previous time instances have to be performed correctly to guarantee that the systems correctness can be checked in the current cycle. For example, if we are checking that a complete multi-part message is always forwarded correctly, then we may have to perform a sequence of reads correctly over time. We will therefore abstract a remodellable memory over a set containing *abstraction pairs* (v^l, t), where (1) each v^l is some signal from the netlist and (2) t is an integer time delay. We require that all abstraction nodes v^l have the same width l as the address nodes of read and writes operating on the memory. The design from Section III is abstracted over the singleton set $\{(\texttt{raddr}^9, 0)\}$.

In the remainder of this section, we will assume that the design that we want to memory abstract contains a single remodellable memory mem (the extension of the procedure to deal with several remodellable memories at a time is straightforward). We also assume that we have been given a set of abstraction pairs; we show how these pairs are generated in Section V-D

The given remodellable memory is abstracted over the provided abstraction pairs as follows:

1) Introduction of Represented Slots: Assume that the data read and written from the remodellable memory mem has width m, and that we are abstracting mem with n abstraction pairs. We introduce represented slots by performing the following three steps for each abstraction pair $p = (v_i^l, d_i)$.

First, we introduce a state variable $\texttt{sel}_i^l$ for (v_i^l, d_i), with an uninitialized initial state function, and a next state function that just propagates the previous value of $\texttt{sel}_i^l$. This *selection register* will contain the concrete slot number that is represented for this abstraction pair during system runs. We also create a register $\texttt{cont}_i^m$. This register is initialized in a way that corresponds to the initialization of mem: If mem has an all-zero initial value, then so does $\texttt{cont}_i^m$; otherwise mem has an all-one initial value, and we initialize $\texttt{cont}_i^m$ correspondingly. (Remember, mem is remodellable so all of its slots are initialized in a uniform way.)

Second, for each pure memory node for mem we create a (v_i^l, d_i)*-abstracted node* as follows:

- the node mem is (v_i^l, d_i)-abstracted to $\texttt{cont}_i^m$.
- the node $\texttt{write}(\texttt{op}^k, \texttt{addr}^l, \texttt{data}^m)$ is (v_i^l, d_i)-abstracted to $\texttt{mux}(\texttt{sel}_i^l = \texttt{addr}^l, \texttt{data}^m, s_0^m)$ if $\texttt{op}^k$ is (v_i^l, d_i)-abstracted to s_0^m.
- the node $\texttt{mux}(\texttt{cond}^1, \texttt{optrue}^k, \texttt{opfalse}^k)$ is (v_i^l, d_i)-abstracted to $\texttt{mux}(\texttt{cond}^1, s_0^m, s_1^m)$ if $\texttt{optrue}^k$ and $\texttt{opfalse}^k$ is (v_i^l, d_i)-abstracted to s_0^m and s_1^m, respectively.

Third, the next-state function for $\texttt{cont}_i^m$ is taken to be the (v_i^l, d_i)-abstracted node of the next-state function for mem; this is possible as the definition of a remodellable memory guarantees that the next state function for mem is a pure memory node for mem.

2) Reimplementation of Read Nodes: We can now reimplement all read nodes for mem in the following fashion. Assume that the ith read node has the form $\texttt{read}(\texttt{op}^k, \texttt{addr}^l)$. For each abstraction pair (v_j^l, t_j), assume that $\texttt{op}^k$ has been (v_j^l, t_j)-abstracted to s_j^m with the associated selection register $\texttt{sel}_j^l$. Introduce a fresh input variable $\texttt{ndtread}_i^m$. We reimplement the read node using a multiplexor tree that returns the contents of the first selected slot with a matching address. If the address does not match any selected slot, we return a nondeterministic value read from the environment on the input $\texttt{ndtread}_i^m$.

Example 1: The multiplexor tree generated for the case of two represented slots will have the form

$$\texttt{mux}(\texttt{addr}^l = \texttt{sel}_0^l, s_0^m, \texttt{mux}(\texttt{addr}^l = \texttt{sel}_1^l, s_1^m, \texttt{ndtread}_i^m))$$

3) Modifying the Verification Condition: The final step of the translation is to modify the output *safe* so that the property only is checked when the abstraction signals have had the selected values at the appropriate previous time instances. Let us define $prev^d(s)$ as a temporal formula that is true at time t in the execution of a system precisely if $t \geq d$ and the combinational signal s evaluates to true at time $t-d$. Assume there are n abstraction pairs (v_i^l, d_i) for s. Our new *safe* output can now be generated by synthesizing a checker for the temporal formula

$$(\bigwedge_{i=0}^{n-1} prev^{d_i}(\texttt{sel}_i^l = v_i^l)) \rightarrow safedef$$

where *safedef* is taken to be the combinational node feeding the old *safe* output. This checker can be implemented in a very simple manner using a number of register chains that delays previous values of some netlist node comparisons.

As the original memory mem is remodellable, it can only occur in the fanin of other state variables through read nodes. We have reimplemented all the read nodes, so the netlist can now be reduced by removing the original memory mem and all logic that depends on it.

B. Correctness

Our notion of correctness is the following: If we have n abstraction pairs (v_i^l, d_i) for a remodellable memory in a design that we are abstracting, and the largest time delay in any abstraction pair is d, then *safe* will always hold in the original system if (1) the transformed signal *safe* in the memory-abstracted system always holds, and (2) *safe* holds in the original system for d cycles from the initial states.

To see that this always holds, we can follow the reasoning used to show the soundness of the transformation of our example from Section III. Specifically, we again remark that the composite system that we produce can be seen as constructed by a two-step process.

In the first step we (1) introduce the selector registers $\texttt{sel}_i^l$ that will be used to control what slots are kept by the abstracted memory (but we do not abstract the memory), and (2) generate a new *safe* signal by synthesizing a checker for the temporal formula

$$(\bigwedge_{i=0}^{j-1} prev^{d_i}(\texttt{sel}_i^l = v_i^l)) \rightarrow safedef$$

where *safedef* is the fanin node for the old *safe* output. To see that this transformation is sound, assume that there is a counterexample trace π in the original system of length l and that $d = max(d_0 \ldots d_{n-1})$. If $l < d$, then we will detect the bug using the bounded check of the original system. If $l \geq d$, some particular initial values for the selectors $\texttt{sel}_i^l$ will make the antecedent of the implication hold in the last time step of π (remember, the initial states for the selector variables are unconstrained, and these variables hold their initial state forever). Furthermore, π will make the consequent of the implication fail in the final time instance. The trace π with some particular initialization for the new selector variables will hence be a valid counterexample for the modified system, and the first transformation is hence sound.

In the second step, the memory in the system resulting from the first step is abstracted to only represent the slots selected by the $\texttt{sel}_i^l$ registers. Due to our construction of the new memory implementation, all reads and writes to slots chosen for representation by the selector variables in our modified memory will happen correctly during all system execution traces. The set of traces generated by this abstraction of the remodellable memory structure is hence a superset of the set of traces of the original memory, so any property that holds on the modified composite system holds on the original system. The second step is hence also sound, as is the overall flow.

C. Abstraction Size

Let us analyze the size of the generated abstracted netlist. Assume that the data read and written from given memory mem has width w, all address nodes have width a, there are i abstraction pairs, j read nodes, and that the maximum delay value in any abstraction pair is d. Each abstraction entry generates a selector entry sel and a memory storage slot cont of width a and w respectively. Furthermore, each read node generates a new nondeterministic input of width w, and the synthesized monitor for the output condition can be implemented with d total single bit delay elements. As a result, the remodeled system will have shrunk the number of necessary registers for the memory encoding from $w * 2^a$ bit-level registers to $i * (a + w) + j * w + d$ bit-level state variables. In the case of the netlist in Section III the memory in the original system has more than 16000 registers and our abstracted implementation will use 75 state variables—a decrease by two orders of magnitude.

D. Generation of Abstraction Pairs

Section V-A shows how a given problem can be abstracted with respect to a particular set of abstraction pairs. However, we have not discussed how to generate abstraction pairs for a particular design problem. One alternative would be to rely on the user for this information. However, this would be error prone, effort intensive, and decrease the utility of our approach significantly. To get around these problems, we instead apply a *counter-example guided abstraction refinement framework* [7] (see Figure 4).

For every remodellable memory in the design, we maintain a current set of abstraction pairs that is monotonically growing. Initially this set is empty for each memory (this initial abstraction generates a system where no slots are represented and each read from a memory returns a nondeterministic result).

Each iteration of our verification flow proceeds as follows:

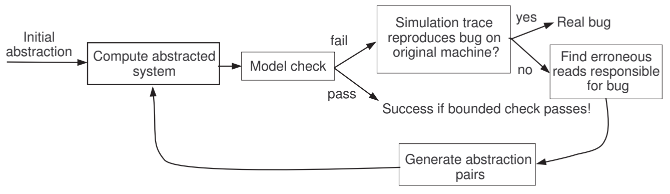
Initial abstraction
Compute abstracted system
Model check
fail
pass
Simulation trace reproduces bug on original machine?
yes
Real bug
no
Find erroneous reads responsible for bug
Success if bounded check passes!
Generate abstraction pairs

Fig. 4. Our abstraction refinement loop

Given the current set of abstraction pairs, we generate a new abstracted model of the system using the techniques from Section V-A and use bit-level model checking to check correctness. If the property holds on the abstracted system, we check bounded correctness of the original system for the necessary number of cycles using standard SAT-based bounded model checking, and declare the system correct if this check passes. If the bounded check of the original system fails, the original system is faulty.

If the model check detects a counter example on the abstracted system, we try to refine our abstraction. The inputs of the abstracted system are a superset of the inputs of the original system, so we can replay the counter example from the abstraction on the original system. If the bug replays on the original system, we are done and report an error to the user. However, if the bug does not replay, we have to refine the abstraction to remove the error trace.

The only difference between the original and abstracted system that could introduce a spurious counterexample is the memory encoding. Some abstracted read node at some time instance must hence return the contents of an unrepresented slot. By inspecting the simulation run on the original system and comparing the values of the pre- and post-abstraction read nodes, we can identify erroneous reads over time in the execution of the abstracted system.

Not all of the erroneous reads will have an impact on the checked property. We find a minimal set of reads and associated time points by initially forcing correct values for all erroneous reads in the abstract system simulation, and iteratively shrinking this corrected set in a greedy way until we find a local minima of forced reads.

Now that we know what erroneous reads need to be corrected and at what time distances from the error cycle, we need to choose the abstraction signal for each time point. We use the following heuristic: If the fraction of read nodes for a memory relative to the number of memory slots for is smaller than 20%, then we abstract on the address signals of the failing reads at each time point. However, if it is not, then we search for a register that 1) is of the same size as the address width of the memory, 2) is in the cone-of-influence of the memory, and 3) contains the address value being read by the erroneous read node at the time instance that the read is performed. We search for the first such register we can find, and if we succeed we use this register node as the abstraction signal at the correct time point rather than the forced read address signal.

The rationale for our abstraction signal choice in the presence of a large number of memory read nodes is that if we are to succeed, we have to hope that there is a register entry that contains the important slot to focus on somewhere else in the design. This is crucial for being able to deal with certain designs such as the content addressable memory in Section VII, where every entry in the memory is read in each cycle, but only a small number of reads at a given time matter.

Example 2: Assume we have a counter example of length 15 for an abstracted version of a design with a single remodellable memory mem with 32 slots and two read nodes. If we detect that the read node $\mathtt{read}(\mathtt{mem}^{1024}, \mathtt{raddr_1}^5)$ needs to have a correct value at cycle 13 (one time step before the failure cycle) to remove the bug trace, then we add the abstraction pair $(\mathtt{raddr_1}^5, 1)$ to the current abstraction set for mem. However, if mem had 28 read nodes, then we would search for a register $\mathtt{reg}^5$ that at cycle 13 contained the concrete address that the read failed for and abstract on $(\mathtt{reg}^5, 1)$ instead. If no such register exists, we revert to the an unabstracted modeling of mem.

VI. Practical Considerations

We use several improvements to the basic algorithms presented in Section V to improve the accuracy and practicality of our framework.

Real designs often contain more than one remodellable memory. We have therefore extended our algorithms to be able to deal with several remodellable memory simultaneously. Moreover, for simplicity of exposition, our presentation in Section V is centered around memories that either are initialized to the constant value $00\ldots0$ or $11\ldots1$ value. In order to deal with real designs it is also necessary to deal with uninitialized memories, and memories with nonuniform initialization. We have also generalized our algorithms to deal with memories with potential out-of-bound reads and writes. These extensions are performed by generating more complex logic for the new implementation of read nodes and selected memory locations.

If a remodellable memory is small but still needs many abstraction pairs, it may be more expensive to abstract it than to bit blast it into a register bank implementation. One example of when this may happen is when some design artifact is represented as an array of single bits. To deal with this problem, we remove memories from our set of remodellable memories when the size of the abstraction of a given memory approaches 75% of the size of the original memory implementation.

In our presentation of the remodellable memory abstraction we introduce one fresh word-level input for each read node to model reads from unrepresented slots. An alternative encoding, that is more efficient for designs with a large number of reads, is to selectively employ a dual rail encoding [15] where appropriate, and return the value X for unrepresented reads. This encoding incurs a one bit overhead for each word-level register in the recursive fanout of the read nodes, as opposed to a fresh width k input per read node. We dynamically evaluate which encoding will work best, and choose the one the generates the smallest amount of registers.

VII. Experimental Results

We have applied our memory abstraction algorithm to two industrial designs, and one state-of-the-art academic design. These designs are representatives from classes of problems that users see very little return-on-investment from traditional model checking technology on today, due to the presence of wide datapaths and the intermingling of data with control. Unfortunately, these designs also represent fundamental building blocks of many larger hardware systems in important domains like telecommunications, networking and graphics processing. Procedures that allow proofs of full or partial correctness of such systems are hence of very high importance.

In our experiments, we combine the memory abstraction with our property preserving word-level bitwidth reduction [4] before generating the bit-level model for analysis. This reduction runs in less than one second for the examples we study, often provides dramatic further reductions, and is guaranteed to never increase the netlist size on designs that are not amenable to it.

Our first example is an industrial FIFO with 75 slots, each containing 32 bits of data. We check the end-to-end correctness safety property that data that gets written into the FIFO gets read out correctly if it has not been overwritten. The original design has about 2500 registers, which are all necessary for a direct full proof, and the standard black-box checker contains a simple reference array which stores correct values to check outputs against. We have not managed to prove this property using any available bit-level method, even when run for several days. The memory abstraction immediately reduces the design to contain one slot after one ten second iteration of abstraction refinement, which shrinks the design to 276 registers. Our bitwidth reduction pass then shrinks the design further to a 58 register equivalent implementation, which BDD-based model checking proves correct in about 20 minutes. Interestingly, although the final reduced design is very small, it is quite complex—over 19000 forward image computations are necessary to traverse its full state space. We also applied our method to a version of the design with a more space efficient white-box checker, which nondeterministically chooses a slot in the FIFO to observe and then monitors correctness of transactions to that particular slot. Verification runtimes do not change when we use this modified checker.

The second example is an industrial Content Addressable Memory (CAM) with three data ports. The CAM has 48 slots, each storing a 20 bit data packet. We check that data that have been written into the CAM are reported as present when the memory is queried, as long as it has not been evicted. The pre-abstraction netlist has 1111 registers, all of which is necessary for a direct proof. No method in our arsenal manages to provide an unbounded proof for this property. One five second round of abstraction refinement reduces the CAM to one slot, which reduces the netlist size to 156 registers. The bitwidth reduction in turn decrease the size of the model to 26 registers, which can be proven using BDD-based model checking instantaneously.

Our third example is an academic high performance router [13]. The router has four ports that connect to adjacent routers, plus inject and eject ports. Messages passed into the router are broken up into *flits*—packets that include both control data and message payload. The design uses two virtual channels per port, each with an associated flit buffer and an array containing information on whether buffer entries are valid. We check the simple partial correctness property that if the router is in a neutral state, then it will forward a message broken up into a header packet, some payload, and a footer packet correctly from the inject port to the north port in a certain number of cycles. This property is provable on the unabstracted model using SAT-based induction [16], but it takes more than 6900 seconds to find the correct induction depth and prove the goal. We detect twenty remodellable memories in the router, which has 7516 registers before the abstraction. Two rounds of abstraction refinement, which takes about 200 seconds, detects that all the validity memories have to be modeled explicitly, whereas we only need to represent two reads at two timepoints from one of the buffer memories. After bitwidth reduction, we end up with a design containing 2196 registers. Induction proves this reduced system in 133 seconds.

VIII. Related Work

Our circuit transformation hinges on the fact that we can compile general RT-level circuit descriptions into a representation that encodes memory manipulations using dedicated read and write nodes. This word-level approach to describing memory manipulating circuitry was pioneered by Burch and Dill in their work on microprocessor verification using quantifier-free logic [6].

The most closely related work to our memory abstraction is the efficient memory modeling that relies on the boundedness of unquantified formula queries used by Ganai and coauthors [9], [10], and by BAT [11]. Our technique is fundamentally different from these approaches, as we *abstract* a sequential *netlist* rather than *simplify* a bounded *formula*. As a case in point, we observe that the standard efficient memory modeling would have provided no reduction for the FIFO and the router from Section VII: From BDD-based model checking of the abstracted design, we know that the FIFO has a backward depth that exceeds 100 time steps (and would hence need more than depth 100 induction). We also know that the router needs depth 20 induction for a proof. These unfolding depths exceed the number of entries in the memories in these designs, so efficient memory modeling can provide no benefits as it must model at least as many slots as the induction depth necessary for a proof. In contrast, we can abstract the FIFO to one slot, and remove several memories completely from the router.

In the domain of software model checking, Armando and coauthors have devised a method for abstracting linear programs operating on arrays [2]. Their idea is to use an abstraction refinement framework to iteratively identify a subset of array elements that needs to be modeled in order to prove a given property. As is the case in our procedure, array entries that are not modeled return nondeterministic values when read. The main difference between Armando and coauthors work and ours, is that our approach can deal with problems where the array indexes that need to be represented vary in each execution sequence of the problem. In fact, none of the examples in Section VII can be verified by abstracting on some particular nontrivial subset of fixed slots.

In previous work, Symbolic Trajectory Evaluation (STE) [15] has been used to verify correctness of designs such as CAMs that contain large memories through the use of a technique called *symbolic indexing* coupled with a specialized abstraction refinement framework [1]. STE verifies bounded length temporal properties for hardware, and has been successful in internal verification tools at a number of semiconductor companies; however, in commercially available tools, unbounded model checking of safety properties has become the dominant technology. Our work allows the benefits of memory abstraction in a framework that the vast majority of users of formal verification technology is familiar with, without requiring them to learn a new logic, rewrite properties, or change their verification methodology.

Our approach for the memory abstraction can be seen as a combination of (1) the selective abstraction of certain memory slots, and (2) an abstraction refinement loop [7]. In previous work, McMillan has used abstract interpretation together with a proof technique he calls *temporal case splitting* to decompose the refinement checking of complex designs like implementations of Tomasulo's algorithm [12] into refinement checking of several abstract models that each only models a small number of memory slots. McMillan performs his abstraction by manually annotating designs with information on how to decompose the checks. In contrast, we focus on solving safety property verification problems, and have structured our approach as a completely automatic algorithm that requires no user input.

There are other attempts to leverage word-level information during symbolic model checking of safety properties. In particular, research has been done on improving the effectiveness of model checking for systems with wide data paths involving arithmetic [8]. These techniques do not deal with symbolic memories in any special way, and their use is thus orthogonal to our memory abstraction. As a result, we can use such word-level model checking tools as our back-end decision procedure and leverage whatever improvements they offer.

In previous work, we have investigated the use of static analysis to improve bounded and unbounded safety property verification of industrial designs [4]. While this work (1) allows us to reduce the size of the verification problems in Section VII significantly and (2) consistently produce significant speed-ups for bounded checks after the reduction, the bitwidth reduction alone does not always manage to shrink designs to the point where we can decide the properties in the unbounded case. In contrast, the combination of memory abstraction and bitwidth reduction allows us to get to full unbounded proofs.

IX. Conclusion

In this paper we have introduced a method that uses word-level netlist information to identify a particular kind of memories that we refer to as remodellable. Such memories interact with their environment using dedicated read and write nodes only, are initialized in a uniform way, and are accessed uniformly. We have presented an abstraction for netlists containing such memories that allows proofs for certain types of properties for which the proof can be done by reasoning about a significantly smaller number of memory slots and time instances than what would be needed in a standard bit-level model check. Our abstraction is sound but not complete. In order to avoid having to rely on abstraction information from the users of our algorithm, we have coupled the abstraction with a counter-example driven abstraction refinement framework that analyze spurious counter examples to incrementally refine the abstraction.

Key features of our approach is that (1) we seamlessly fit into a standard transformation-based verification system for safety property verification, (2) our algorithms are completely automatic, (3) we require no input on abstraction from users, (4) we can use any bit-level model checker as the decision procedure in our abstraction refinement framework.

As demonstrated by our experimental results in Section VII, we can prove correctness properties for industrial and academic high-performance designs that are out of range for standard bit-level model checkers. The designs we analyze have wide datapaths, manipulate large swaths of memory, and have nontrivial control structure. Approaches that rely on explicit modeling of all memories, and the flow of information between each and every memory cell hence end up with decision problems that can be extraordinarily more expensive to decide than our reduced models.

We believe that the work presented here, together with our previous work on utilizing word-level information to perform bitwidth reduction, will provide a way to leverage model checking on some classes of designs that today are intractable for users of industrial tools. Hopefully, the present research represents just the tip of the iceberg in terms of ways to utilize higher-level information to speed up safety property verification.

Acknowledgments

Many thanks to Tamir Heyman, who participated in discussions and helped with the work necessary to integrate our analysis with the front-end flow. The authors would also like to thank Jason Baumgartner, Aarti Gupta, John Matthews, and the anonymous reviewers for their helpful feedback on previous versions of this paper.

References

[1] S. Adams, M. Björk, T. Melham, and C.-J. Seger. Automatic abstraction in symbolic trajectory evaluation. In *Proc. of the Formal Methods in CAD Conf.*, 2007.

[2] A. Armando, M. Benerecetti, and J. Mantovani. Abstraction refinement of linear programs with arrays. In *Proc. Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2007.

[3] J. Baumgartner, T. Gloekler, D. Shanmugam, R. Seigler, G. V. Huben, H. Mony, P. Roessler, and B. Ramanandray. Enabling large-scale pervasive logic verification through multi-algorithmic formal reasoning. In *Proc. of the Formal Methods in CAD Conf.*, 2006.

[4] P. Bjesse. A practical approach to word level model checking of industrial netlists. In *Proc. of the Computer Aided Verification Conf.*, 2008.

[5] R. Bryant, S. Lahiri, and S. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Proc. of the Computer Aided Verification Conf.*, 2002.

[6] J. L. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Proc. of the Computer Aided Verification Conf.*, 1994.

[7] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. of the Computer Aided Verification Conf.*, 2000.

[8] E. Clarke, M. Khaira, and X. Zhao. Word-level symbolic model checking: avoiding the Pentium FDIV error. In *Proc. of the Design Automation Conf.*, 1996.

[9] M. K. Ganai, A. Gupta, and P. Ashar. Efficient modelling of embedded memories in bounded model checking. In *Proc. of the Computer Aided Verification Conf.*, 2004.

[10] M. K. Ganai, A. Gupta, and P. Ashar. Verification of embedded memory systems using efficient memory modeling. In *Proc. of Design, Automation, and Test in Europe.*, 2005.

[11] P. Manolios, S. Srinivasan, and D. Vroon. BAT: The bit-level analysis tool. In *Proc. of the Computer Aided Verification Conf.*, 2007.

[12] K. L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *Proc. of the Computer Aided Verification Conf.*, 1998.

[13] L.-S. Peh and W. Dally. A delay model and speculative architecture for pipelined routers. In *Proc. Intl. Symposium on High-Performance Computer Architecture*, 2001.

[14] S. Ranise and C. Tinelli. Satisfiability modulo theories. *Trends and Controversies - IEEE Intelligent Systems Magazine*, December 2006.

[15] C.-J. H. Seger and R. E. Bryant. Formal verification by symbolic evaluation of partially ordered trajectories. *Formal Methods in System Design*, 6(2):147–190, March 1995.

[16] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *Proc. of the Formal Methods in CAD Conf.*, 2000.