Coping with Moore's Law *(and More)*: Supporting Arrays in State-of-the-Art Model Checkers

Jason Baumgartner Michael Case Hari Mony
IBM Systems & Technology Group

***Abstract*—State-of-the-art hardware model checkers and equivalence checkers rely upon a diversity of synergistic algorithms to achieve adequate scalability and automation. While higher-level decision procedures have enhanced capacity for problems of amenable syntax, little prior work has addressed (1) the generalization of many critical synergistic algorithms beyond bit-blasted representations, nor (2) the issue of bridging higher-level techniques to problems of complex circuit-accurate syntax. In this paper, we extend a variety of bit-level algorithms to designs with memory arrays, and introduce techniques to rewrite arrays from circuit-accurate to verification-amenable behavioral syntax. These extensions have numerous motivations, from scaling formal methods to verify ever-growing design components, to enabling hardware model checkers to reason about software-like systems, to allowing state-of-the-art model checkers to support temporally-consistent function- and predicate-abstraction.**

I. Introduction

Contemporary hardware designs are often of substantial complexity, comprising a diversity of bit-level control logic, datapaths, and performance-related artifacts including pipelining, multi-threading, out-of-order execution, and power-saving techniques. While reference models expressing the correctness of such designs may be specified at a higher abstraction level, it is often necessary to directly reason about the circuit-accurate implementation. For example, equivalence checkers *must* reason about the circuit-accurate implementation. If the designer-specified implementation closely matches the circuit, combinational equivalence checking (CEC) may scalably solve the equivalence-checking problem – leaving a formidable correctness check of the circuit-accurate implementation vs. the reference model. If in contrast the implementation more closely matches the higher-level specification, functional verification becomes simpler, leaving a formidable sequential equivalence check between the implementation vs. the circuit.

Numerous automated transformations have been developed to alleviate the challenges of verifying contemporary hardware designs. For example, *phase abstraction* eliminates verification complexities of designs with intricate clocking and latching schemes [1]. *Retiming* reduces the verification overhead associated with pipelined designs [2]. *Redundancy removal* and *rewriting* eliminate numerous design artifacts which may dramatically hurt verification scalability [3], [4], [5]. Such techniques have become key components of state-of-the-art model checkers and equivalence checkers [6], [1], [7], without which such solvers often fail to yield a conclusive result on industrial designs. However, these techniques have hitherto largely been developed assuming a *bit-blasted* representation.

Substantial recent research has focused upon enhanced reasoning scalability for designs expressed at a higher-level of abstraction. For example, numerous techniques have been established to enhance the verification scalability of designs containing *arrays*: storage devices arranged as a set of addressable rows of a specific width, accessed through atomic *write* and *read* operations. Example techniques include the *efficient memory model* which preserves data consistency within temporally-bounded reasoning using a modeling whose complexity grows sub-linearly with respect to array size [8], [9], and the abstraction-refinement technique of [10] which reduces an array to a small number of consistently-modeled rows. Additionally, a *large number* of dedicated decision procedures have been developed around theories of arrays [11].

While extremely powerful for amenable problems, such techniques have not yet delivered their full impact in industrial hardware verification for several reasons. First, such techniques are often applicable only to designs with behavioral syntax, not to designs of intricate circuit-accurate syntax. Manual creation of behavioral models may alleviate this concern for property checking – though at an often-prohibitive expense to the overall design flow. Furthermore, these behavioral models must be equivalence-checked to the circuit-accurate implementation to ensure the soundness of such an approach; while property checking may become simpler, the equivalence check may be intractable. Second, techniques which are incompatible with bit-level transformations are of limited utility on industrial designs, given capacity limitations in reasoning about the logic *adjacent to* the arrays. In our experience, the logic around the dataflow often contains the most subtle flaws; the dataflow itself poses a bottleneck to verification algorithms which often necessitates manual guidance to expose these flaws and ultimately establish correctness.

In this paper, we address the issue of efficient formal reasoning about industrial hardware designs which include arrays. Our contributions include: **(1)** algorithmic extensions to a variety of traditionally bit-level transformation algorithms to support designs with arrays, including redundancy removal (Section III), phase abstraction (Section IV), temporal decomposition and retiming (Section V); **(2)** techniques to simplify array syntax, enabling efficient array reasoning upon designs which may otherwise lack a suitable behavioral representation (Section III-C); **(3)** enhancements to the robustness and scalability of known array abstraction techniques (Section VI). Experiments are provided in Section VII to confirm the profound verification benefits enabled by these techniques.

There are numerous motivations for this work.

- As per Moore's Law, increasing array size (caches, main memory, lookup tables, ...) is one prevalent way in which growing transistor capacity is used to increase design performance [12]. While bit-blasted analysis suffers substantial overhead with doubled array size, native reasoning techniques often entail sub-linear complexity growth – e.g., merely requiring an additional address-comparison bit.
- There are numerous problem domains which are practically infeasible for bit-blasted techniques without manual abstraction, such as formally verifying logic that interacts with main memory or large caches. Large arrays already constitute a substantial scalability differential between formal and informal industrial verification efforts, as most hardware simulators and accelerators represent arrays without bit-blasting.
- Increasing the scalability of automated solutions *mandates* enabling the applicability of as large a set of algorithms as possible, to leverage algorithmic synergies to eliminate implementation characteristics which otherwise may pose a bottleneck to, if not outright inapplicability of, otherwise well-suited algorithms. This is particularly true for *satisfiability modulo theories* solvers, which tend to be highly sensitive to the type of logic which may be efficiently handled by a given combination of theories (e.g., [13]).
- Randomly-initialized read-only arrays may be used to abstract complex combinational functions in a temporally-consistent manner. In particular, the data output of such an array, addressed by the arguments to the function being abstracted, may be used to replace the logic associated with that function. This uninterpreted modeling may simulate the original function, hence is sound for verification – and maintains the necessary invariant for arbitrary model checking algorithms that applying identical arguments to the abstracted function at different points in time yields identical results [14]. Our techniques thus constitute a method to utilize uninterpreted functions in a state-of-the-art model checker.

II. Preliminaries

We represent the design under verification as a *netlist*.

Definition 1: A *netlist* comprises a directed graph with vertices representing gates, and edges representing interconnections between gates. Gates have associated functions, such as constants, primary inputs (termed *RANDOM* gates), combinational logic of various functionality, and single-bit sequential elements termed *registers*. Registers have associated *initial values* defining their time-0 or *reset* behavior, and next-state functions defining their time-$(i+1)$ behavior.

The *And / Inverter Graph (AIG)* is a commonly-used netlist representation where the only combinational primitives are single-bit inverters and two-input AND gates [3], [4]. This implies a bit-blasting of all higher-level constructs. Our netlist format is an AIG which also includes *array* primitives.

Definition 2: An *array* is a gate representing a two-dimensional grid of registers (referred to as *cells*), arranged as rows vs. columns. Cells are accessed via *read* and *write ports*.

```
reg [COLS-1 : 0] ram[ROWS - 1 : 0]; // array declaration
always @(posedge clk) begin
                     // write port:
  if (wr_en)         // enable is  "(posedge clk AND wr_en)"
    ram[wr_addr] <= // address is "wr_addr"
      wr_data;       // data is    "wr_data"
end
                     // read port:
assign rd_data =     // data is    "rd_data"
     rd_en ?         // enable is  "rd_en"
     ram[rd_addr] : // address is "rd_addr"
     {(COLS){1'bX}};
```

Fig. 1: Verilog array example

Ports have three types of *pins*: an enable, an address vector, and a data vector: refer to Figure 1. The *enable* indicates whether the given port is actively accessing the array cells. The *address* indicates which row is being accessed. The *data* represents the values to be stored to (read from) the given row for a write (read) port. A *column* refers to a one-dimensional vector: the ith cell of each row. All pins are inputs of the array gate, aside from read data pins which are outputs.

Arrays have a defined number of r rows, q columns, and p address pins per port; a default *initial value* (in case an unwritten row is read); and an indication of *read-before-write* vs *write-before-read* behavior. The latter is relevant in case of a concurrent read and write to the same address: write-before-read will return the concurrent write data, whereas read-before-write will not. Read data is conservatively randomized when the read enable is de-asserted, or when the read is "out-of-bounds" – i.e., its address exceeds the number of array rows. Write ports have a specified precedence (e.g., reflecting the order of *if, else if* statements in Verilog), defining which will persist in case of concurrent stores to the same address.

We refer to the read ports as $R_1, \ldots, R_m$, and the write ports in order of increasing precedence as $W_1, \ldots, W_n$. For a given port P_i, let $P_i.e$ represent its enable pin, $P_i.a(0 \ldots, p-1)$ its address pins, and $P_i.d(0, \ldots, q-1)$ its data pins.

Definition 3: A *merge* is a reduction technique which effectively eliminates a gate from a netlist by replacing its fanout references with references to a semantically-equivalent gate.

It is highly desirable to be able to merge array outputs if it can be determined that the referenced array cells exhibit redundancy. However, the nondeterminism exhibited at an array output when its read port is disabled or out-of-bounds often precludes a direct merge from being a semantically-consistent transformation.

Definition 4: An *array output merge* is a specialized merge to achieve the desired netlist reduction while preserving necessary nondeterminism. This operation consists of replacing the array output to be merged by a multiplexor which selects the merged-onto gate when the corresponding read port is enabled and in-bounds, else selects a unique RANDOM gate.

A. Temporal Unfolding and the Efficient Memory Model

Many algorithms reason about netlist behavior over a specific number of timesteps. *Unfolding* is commonly used for this purpose, replicating the netlist for the desired number of timesteps to allow valuations to propagate through next-state functions. Depending upon the application for which

unfolding is performed, the time-0 unfolding of the sequential elements will differ. For Bounded Model Checking (denoted as unfold$_b$), the time-0 value will be the initial value of the array or register [15]. For induction, the time-0 value will be a RANDOM gate [16]. For a sequential transformation such as phase abstraction (denoted as unfold$_p$), the time-0 value will be a reference to an existing array or register in the netlist [1].

The efficient memory model (EMM) represents data consistency for arrays within unfoldings using sub-linear modeling size vs. the number of array cells [9]: the data at an array output at time i for an enabled, in-bound read must be the highest-priority, most-recently-written data for the corresponding address. This may be modeled in unfolding using a sequence of *if-then-else* constructs, one per write port and timestep, selected by the corresponding write being enabled and address-matching the read being synthesized [9]. Because each read must be compared to each write, the size of *each* synthesized read for time t is $O(t \cdot |W|)$, resulting in overall quadratic unfolding size with respect to depth as a multiple of the number of write ports $|W|$ and read ports $|R|$.

A technique to further reduce array unfolding size is proposed in [17], re-encoding array references given upper-bounds on the number of distinct referenced addresses. Rewriting rules are used to minimize the number of memory references, e.g., synthesizing *if-then-else* constructs for reads as with EMM. While highly effective for suitable problems, we have not yet found a method to advantageously leverage this technique in a model checking framework: these rewriting rules shadow the complexity of an EMM unfolding, and since arrays are generally interconnected by arbitrary bit-level logic it is challenging to improve upon the effectiveness of standard logic optimization techniques upon such unfoldings, or to *a priori* meaningfully upper-bound a desired unfolding depth.

B. Symbolic Row Abstraction

While EMM is highly effective to boost the efficiency of temporally-bounded reasoning, many alternate algorithms are critical to a robust model checker. For example, BDD-based reachability analysis is often necessary to prove properties of extremely temporally deep netlists. For such temporally-unbounded algorithms, EMM is not directly applicable.

A related technique has been proposed in [10] as a *sequential netlist* abstraction that is applicable for arbitrary model checking algorithms. This abstraction bit-blasts an array into a small set of symbolic rows. This set begins empty and rows are added during refinement in response to spurious counterexamples. In addition to modeling *data* contents for represented rows, the *address* correlating to each modeled row is represented using nondeterministically-initialized registers; reads and writes to modeled rows are performed precisely, whereas writes to unmodeled rows are ignored and reads of unmodeled rows are randomized. To prevent trivial failures merely due to reading unmodeled rows, *antecedent conditioning* of properties is performed: given a spurious counterexample caused by a read from port R_i which occurred k timesteps prior to the property failure, resulting in a new row being modeled with symbolic address r_i^k, property $always(p)$ is replaced by $always\big(prev^k(R_i.a \equiv r_i^k) \rightarrow p\big)$. This abstraction is sound because the abstract netlist may simulate the original, and the antecedent conditioning forms a complete temporal case-split. While very effective for certain types of problems, the abstraction risks exceeding the size of a bit-blasted netlist due to the need to represent modeled addresses, and due to a potentially large number of temporal read dependencies.

III. Logic Optimization Techniques

A vast collection of logic optimization techniques have been developed over the past decades, which reduce netlist size while preserving the behavior of sequential elements. Examples include redundancy removal [3], [5] as well as extensions under observability don't cares [18], and syntactic combinational rewriting [4]. Many of these techniques operate on local logic windows treating sequential elements as unconstrained *cutpoints*, hence are directly applicable to netlists with arrays. Some require bounded / inductive reasoning, possibly to derive *invariants* with which to constrain local analysis, for which the *efficient memory model* provides a suitable extension to netlists with arrays. There are however several optimization techniques which have required substantial customization to achieve an adequate level of scalability and optimality, which we detail in this section.

A. Ternary-Simulation Based Analysis and Reduction

Ternary simulation-based reduction is a method to identify and eliminate a subset of semantically-equivalent gates. Initially, the registers are assigned their initial values and the inputs are assigned an *unknown* ternary *X* value. Next-state functions are then simulated, overapproximating an image computation. These next-states values are propagated through the registers, and another overapproximate image is computed. This iteration continues until a repeated ternary state is detected, indicating that an overapproximation of the reachable states have been explored. Pairs of gates which always evaluate to the same deterministic values in these states may be *merged* to reduce netlist size [1]. This technique is highly overapproximate and able to identify a relatively small subset of truly redundant gates, though is remarkably scalable and often able to yield substantial reductions on industrial netlists [1]. This analysis may also be used to detect oscillating clocks for phase abstraction (Section IV), and transient behavior for temporal decomposition (Section V). Ternary simulation has thus found a role in many state-of-the-art model checkers.

Unlike with Boolean simulation, each three-valued address may resolve to *multiple* existing simulated array value entries. Numerous commercial simulators support multi-valued reasoning, though to avoid the computational overhead entailed by multiple-entry address resolution they take shortcuts such as mapping *X* values on enables or addresses to Boolean constants, or *X*'ing array contents in such cases, as also was noted in [8]. Such shortcuts render unacceptable suboptimalities and even unsoundness in model checking applications.

Building upon the work of [8], which uses three-valued write lists for precise read resolution in *symbolic trajectory*

Algorithm 1 Ternary simulation *write* function

1: **function** write(*enable, addr, data*)
2: **if** (*enable* $\equiv$ 0) **then** return
3: *nodesToWrite* = deepest nodes whose address intersects *addr*
4: **for all** *node* in *nodesToWrite* **do**
5: *subAddr* = intersection of *addr* and *node*.address
6: **if** (*subAddr* $\equiv$ *node*.address) **then**
7: *node*.data = (enable $\equiv$ 1) ? *data* : `resolve`(*node*.data, *data*)
8: **else** add child to *node* with address *subAddr* and data *data*
9: **end for**
10: **for all** (address cube *subAddr* in *addr* not written above) **do**
11: add new child to tree root with address *subAddr* and data *data*
12: **end for**
13: subsume children with data equal to parent
14: **end function**

evaluation, we have developed the following algorithm for precise and efficient three-valued array simulation. Our framework uses a tree structure, where each node represents an ⟨address, data⟩ tuple and edges satisfy the following relationships, maintained during *writes* to enable efficient *reads*:

- A child's address cube is *contained* in its parent's address.
- Child addresses are *exceptions* to parent addresses. E.g., given parent $\langle XX1, D_0\rangle$ with child $\langle X11, D_1\rangle$, addresses $\{XX1 \setminus X11\}$ have data D_0 and $\{X11\}$ has data D_1.
- For any parent, the addresses of all children are disjoint.

Read operations traverse the tree to identify nodes with addresses intersecting the referenced address. A *resolve* function is used to compute the tightest cube that contains all associated three-valued data, similar to resolution across list entries in [8]. Accordingly, *X*-saturated data may be returned without traversing all relevant nodes. Write operations, detailed in Algorithm 1, similarly traverse nodes with intersecting addresses. These nodes are updated if the write address covers their address, else a new child node is created. We employ a more efficient data structure with more aggressive subsumption rules than used in [8], since simulation applications may entail 1000s of timesteps of analysis. The need to continually retraverse lists often degrades to quadratic runtime over simulation depth, whereas the use of a tree enables analysis to be limited to the subset of nodes relevant to a given operation.

Figure 1 illustrates the tree resulting from an array initialized to 000, after a write of ⟨*1*XX, 1XX⟩, then ⟨*X*X0, XX0⟩, then ⟨*X*1X, 01X⟩.

B. Sequential Redundancy Identification and Removal

Arrays are composed of columns comprising one cell per row. It is possible for two array columns (within the same or across different arrays) to evaluate identically in all reachable states. This is particularly common when equivalence checking netlists with arrays; the arrays themselves may be unaltered (merely the logic *adjacent to* the arrays may be altered), or they may reflect a column-equivalence-preserving transformation such as partitioning. The overall equivalence check nonetheless often requires reasoning about array contents, if e.g. the logic adjacent to the arrays was optimized using *don't care* conditions inherent in the array data, precluding their elimination via *black-boxing* [19]. Solving the equivalence checking problem requires efficient methods to identify and

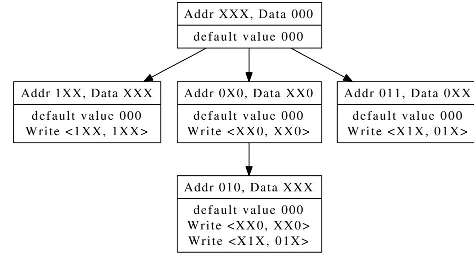

Fig. 1: Three-valued array simulation example

eliminate such column redundancy. More generally, column equivalence is a form of netlist redundancy whose removal significantly benefits the scalability of all types of verification.

Induction is a scalable technique which may be used to identify sequential redundancy [5]. An inductive unfolding instantiates a distinct RANDOM gate for each sequential element to represent an arbitrary state. If it is desired to prove equivalences among sequential elements, the corresponding induction hypotheses constrain the values of these RANDOM gates and thereby often enable inductive redundancy identification. With arrays, however, it is desirable to *not* require explicit correlation of individual cells or even rows, as their cardinality may render such reasoning intractable – basically degrading to the overhead of redundancy identification on a bit-blasted netlist. Directly attempting to establish array output or column equivalence without cell correlation is a highly-noninductive problem, since each unfolding timestep may reference a distinct row, hence induction hypotheses over earlier timesteps do not meaningfully constrain later timesteps.

One approach that we have found useful to enable inductive redundant column identification is to move the proof obligation from array outputs to inputs: two columns are equivalent if they have the same number of rows, they initialize equivalently and any value written to one column is concurrently written to the other column. This proof obligation may be decomposed into a bidirectional check that each enabled, inbound irredundant write to one column has an equivalent write to the other column. This check may be formalized as follows, where it is suspected that columns i and j of arrays A_i and A_j, respectively, are equivalent. Predicate $\mathtt{oob}(W_i.a)$ indicates that W_i has an out-of-bounds address. Predicate $\mathtt{rdt}(W_i)$ indicates that W_i is superseded by a higher-precedence write to the same address, and may be strengthened to check that $W_i.d(i)$ differs from the current value of the addressed cell.[1]

$$\forall\,\text{ports}\, W_i \text{ of } \mathrm{A_i} : W_i.e \,\wedge\, \neg\mathtt{oob}(W_i.a) \,\wedge\, \neg\mathtt{rdt}(W_i).$$
$$\exists\,\text{port}\, W_j \text{ of } \mathrm{A_j} : W_j.e \wedge (W_i.a \equiv W_j.a) \wedge \neg\mathtt{rdt}(W_j)$$
$$\wedge\,\big(W_i.d(i) \equiv W_j.d(j)\big)$$

Speculative reduction is a technique to enable the benefit of a merge even before the corresponding suspected redundancy has been proven, yielding orders of magnitude speedup to redundancy identification [5]. This technique simplifies the netlist while retaining a proof obligation to identify whether the postulated redundancy is accurate. Speculative reduction

[1]This *irredundant* write-data condition is often necessary in practice, to enable column equivalence detection despite *don't care* optimizations used to minimize the redundant writing of values already present in the array.

may be extended for column equivalences by modifying each read port that references a potentially-redundant column. If it is suspected that columns i and j of arrays A_i and A_j, respectively, are equivalent, each read port R_i referencing column i may be modified to derive values from column j. This is accomplished by synthesizing a new read port R^*_{ij} of array A_j with $R^*_{ij}.e = R_i.e$ and $R^*_{ij}.a = R_i.a$, and replacing references to the redundant column of R_i by references to the representative column of R^*_{ij}. Note that speculative reduction of array outputs reduces the number of RANDOM gates in the inductive unfolding, which is *essential* to overall inductivity.

Column equivalence conditions may be verified directly on the speculatively-reduced netlist. Any identified column equivalences may be eliminated from the netlist, replacing R_i by the corresponding R^*_{ij} as in the speculatively-reduced netlist. The array representation may then be simplified using the techniques introduced in the following section.

C. Array Simplification Techniques

In addition to simplifying logic *around* the arrays, it is advantageous to simplify the arrays themselves: the number of columns, rows, ports, and even the number of distinct arrays. All simplifications tend to enhance algorithmic scalability, and these particular simplifications are often practically *necessary* to enable the efficient use of array reasoning techniques. For example, "content-addressable memories" often have one read port per row, using downstream logic to select which reads are actually relevant. Additionally, industrial arrays often entail circuit-oriented characteristics which may entail fragmenting wide arrays into numerous narrow arrays, implementing one write port per row with orthogonal address-related enables, or intertwining test- or initialization-logic with the array.

Such circuit-accurate arrays pose numerous challenges to verification, which often render them substantially *less* efficient to verify in their native vs. bit-blasted form.

- The efficient memory model entails large unfoldings for netlists containing many arrays with many read and write port (refer to Section II-A).
- As will be discussed in Section VI, the abstraction approach of [10] may run into suboptimalities or even inapplicability given such circuit-accurate syntax.
- Logic simulators are significantly burdened by such representations, and accelerators may be *unable* to model such arrays without bit-blasting – motivating manual creation of behavioral representations for enhanced validation, and using equivalence checking to establish their correctness.

We have found the following transformations essential to *automatically* convert circuit-accurate array representations to behavioral representations for enhanced property checking and equivalence checking. These techniques also are useful to simplify ports created through other transformations such as phase abstraction, and generally to simplify arrays to as efficient of representations as possible.

1. If a given data pin is disconnected from every read port, the corresponding column may be eliminated from the array.
2. Read ports with no connected data pins may be eliminated.
3. Arrays with no read ports may be eliminated.
4. If the enable pin of a given port is semantically equivalent to 0, that port may be eliminated. If that port is a *read*, its outputs may be replaced by RANDOM gates.
5. If a given address pin is an identical constant across every read port, some rows are un-readable hence the array's address space may be reduced. Each write port may conjunct its enable with the condition that its corresponding address pin evaluates to this constant value, then the number of rows and address pins may be reduced accordingly.
6. If a pair of ports P_i, P_j for $i < j$ have identical addresses, and these ports are *compatible*,[2] then these ports may be coalesced to eliminate P_i. Coalescing of write ports consists of multiplexing data: *if* $P_j.e$ *then* $P_j.d$ *else* $P_i.d$. Read data may be directly merged as per Definition 4. The enable pin of P_j is finally replaced by $(P_i.e \vee P_j.e)$.
7. Similar to item 6, if compatible ports P_i, P_j for $i < j$ have orthogonal enables, then these ports may be coalesced to eliminate P_i. Data and address pins on P_j are multiplexed by enables, then P_j's enable is disjuncted with that of P_i.
8. If every data pin of read port R_i has the same *observability don't care* condition O_i, then $R_i.e$ may be optimized using O_i as a don't care – e.g. conjuncting $R_i.e$ with O_i. This often enables the orthogonal-enable port coalescing of item 7.
9. For a write-before-read array, if read port R_i and write port W_j have semantically-equivalent addresses, $W_j.e$ implies $R_i.e$, and no higher-precedence write port may address-match R_i, then $R_i.d$ may be merged onto $W_j.d$ as per Definition 4.
10. If each write port has a semantically-equivalent data pin for two different columns m and n, array outputs for columns m and n may be merged.
11. If arrays A_i and A_j have an identical number of rows and read-before-write vs. write-before-read type, and they have an identical number of ports of each type with semantically-equivalent enable and address pins, the columns of A_j may be concatenated onto A_i, eliminating A_j.
12. If arrays A_i and A_j have identical size and type, identical deterministic initial values, and an identical number of write ports with semantically-equivalent enable, address, and data pins, the read ports of A_i may be migrated to A_j.
13. A write-before-read array may be converted to a read-before-write array, by creating a multiplexor for each read port which selects the highest-precedence concurrent write port data, else the array output itself if no such write exists. This may enable array elimination as per item 11 or 12.

These simplifications are synergistic in that one reduction may enable the applicability of another, and we have found it useful to iterate the above transformations until no further reduction is achieved. It is also useful to iterate these reductions with other logic optimization and abstraction techniques because simplifying the logic *around* the arrays may greatly enhance the applicability of these reductions and vice-versa.

[2]All read ports are compatible. Write ports are compatible if no port P_k for $i < k < j$ may concurrently write to the same address.

IV. Phase Abstraction

Phase abstraction is a temporal abstraction which unfolds next-state functions for a specific number of timesteps c. The resulting netlist represents a c-accelerated variant of the original netlist, such that each state transition of the abstracted netlist correlates to c consecutive transitions of the original netlist. This unfolding results in c copies of every combinational gate in the original netlist, correlating to different modulo-c timesteps. Safety property checking is preserved by disjuncting over each copy of the property gate [1].

Phase abstraction has been demonstrated to yield dramatic speedups to the verification of *clocked* netlists where most registers toggle at most once every c consecutive timesteps. This transformation eliminates the need to model an oscillating clock in the netlist, and often eliminates many registers from the cone of influence as their values become irrelevant to the unfolded next-state functions. Additionally, phase abstraction greatly enhances the reduction capability of techniques such as retiming and redundancy removal [1] and enhances a variety of verification algorithms such as reachability analysis, interpolation [20], and induction [16]. This technique has thus become an essential component of many industrial-strength hardware model checkers [6], [1], [7]. In this section, we extend phase abstraction to netlists with arrays.

Phase abstracted arrays intuitively must have the following characteristics: **(1)** Abstracted write ports must be replicated to reflect all updates that may occur during the c consecutive unfolded timesteps. **(2)** Abstracted read ports must be replicated to support all data fetches which may occur during the c consecutive unfolded timesteps. It is nonetheless essential to ensure that data consistency is maintained during this transformation: read ports for "older" unfolded timesteps cannot be allowed to return write data from "newer" unfolded timesteps. Algorithm 2 yields the necessary semantics-preserving transformation through creation of new array ports.

Algorithm 2 Array-compatible phase abstraction algorithm

```
 1: function phaseAbstract(netlist, unfoldDegree)
 2:   for all array in netlist do
 3:     writePorts = set of write ports in original array
 4:     readPorts = set of read ports in original array
 5:     for all time in 0 to unfoldDegree-1 do
 6:       for all R in readPorts do
 7:         // port syntax: ⟨enable, address, data⟩
 8:         create shell read port ⟨∅, ∅, R_time⟩
 9:       end for
10:     end for
11:     for all time in 0 to unfoldDegree-1 do
12:       for all R in readPorts do
13:         fill in ⟨unfold_p(R.e, time), unfold_p(R.a, time), R_time⟩ for R
14:       end for
15:       for all W in writePorts via increasing precedence do
16:         append ⟨unfold_p(W.e, time), unfold_p(W.a, time), unfold_p(W.d,
            time)⟩ as highest-precedence write port
17:       end for
18:     end for
19:   end for
20:   perform traditional phase abstraction over non-array gates [1]
21:   convert all arrays to type read-before-write
22: end function

23: function unfoldReadPort_p(port, time)
24:   readData = R_time
25:   time′ = (port's array is write-before-read) ? time : time-1
26:   for all time″ in 0 to time′ do
27:     for all W in writePorts via increasing precedence do
28:       readData = if (unfold_p(W.e, time″) ∧ (unfold_p(W.a, time″) ≡
          unfold_p(R.a, time))) then unfold_p(W.d, time″) else readData
29:     end for
30:   end for
31:   readData = if (¬unfold_p(R.e, time) ∨ (unfold_p(R.a, time) is out-of-
      bounds)) then R_time else readData
32:   return readData
33: end function
```

To ensure data consistency, function unfoldReadPort$_p$ synthesizes data-forwarding paths for read ports unfolded within unfold$_p$, to capture the most-recent applicable unfolded write data. This data may be concurrent for a write-before-read array, else must be strictly earlier. If no such write occurs (the *if-the-else* returns line 24), or if the read enable is deasserted or its address is out-of-bounds (line 31), the read is satisfied by a reference to the newly-created read port from line 8. Note also that the type of the array is converted to read-before-write to ensure that unfoldings for "newer" write ports will not satisfy "older" reads.

V. Temporal Decomposition and Retiming

Transient simplification is a technique to reduce a netlist with respect to *transient signals* which behave arbitrarily for a fixed number of timesteps after reset, and thereafter settle to a reducible behavior. The prefix timesteps, before the transient signals settle to their reducible behavior, may be verified with Bounded Model Checking. The netlist may then be *time-shifted* to represent its post-prefix behavior, decomposing the verification task such that unbounded analysis may focus only upon timesteps after which the transient signals have settled and hence may be eliminated [21]. Such decomposition may reduce the overhead associated with *initialization logic* in a verification testbench. A subset of transients may be efficiently detected using ternary simulation. Given efficient techniques for ternary simulation and Bounded Model Checking, the extension necessary to support temporal decomposition for netlists with arrays is that of time-shifting the arrays.

Time shifting replaces initial values by the set of states reachable in a specific number of timesteps. For registers, a temporal unfolding of their values may be used as their new initial values [21]. Like registers, arrays have initial values that must be modified to reflect writes that occur within the time-shifted prefix. Algorithm 3 illustrates the overall time-shifting transformation. To ensure data consistency, this algorithm places the unfolded prefix write ports lower in precedence than the existing ports, which are used to reflect post-transient writes. These prefix ports are prioritized in order of increasing unfolding time, following the precedence order of the original write ports within each timestep.

Retiming is a technique which moves registers across other types of gates in a netlist, reducing their cardinality while preserving overall netlist behavior. Each retiming step moves one register from each input of a gate to each of its outputs, or vice-versa. The number of registers moved fanin-wise across a gate is referred to as its *lag*, representing the number

Algorithm 3 Array-compatible time-shifting algorithm

```
1: function timeShift(netlist, timeSteps)
2:   for all register in netlist do
3:     initialValue[register] = unfold_b(register, timeSteps)
4:   end for
5:   init = new register with initial value 1, next-state function 0
6:   for all time in 0 to timeSteps-1 do
7:     for all array in netlist do
8:       newPorts = ∅
9:       for all writePort of array via increasing precedence do
10:        append newPorts with ⟨(init ∧ unfold_b(writePort.e, time)),
           unfold_b(writePort.a, time), unfold_b(writePort.d, time)⟩
11:      end for
12:      inject newPorts in appended precedence order as lowest-priority
         write ports for array
13:    end for
14:  end for
15: end function
```

of timesteps its behavior has been delayed. Coupled with *peripheral retiming*, in which registers may be borrowed or discarded across RANDOM gates or properties, retiming has been demonstrated to enable orders of magnitude speedup to numerous verification algorithms [2], [6]. *Normalized retiming*, in which all lags are negative, is often used in verification to ensure that retimed initial values may be consistently computed through unfolding. Computing of retimed initial values is analogous to that for time-shifting, aside from the distinction that the lag of each gate may differ hence unfolding is performed at a finer level of granularity.

The following customizations enable the retiming of arrays.

1. All pins associated with a given port must have an identical lag to ensure that each port may be evaluated atomically.
2. No write port may be lagged to a more-negative degree than any read port for a given array. This is to ensure that a read cannot return data from a *later* write, similar in justification to the need to convert write-before-read to read-before-write arrays for phase abstraction in Algorithm 2.
3. For every array with a lagged write port, we use a mechanism similar to Algorithm 3 to reflect its prefix writes. For each array, we iterate from 0 to the maximum negative lag of any write port. For each write port, if its lag is more-negative than the current time iteration, we enqueue a port reflecting the time-iteration unfolding of that port, conjuncting the corresponding enable with an *init* register. We finally inject this queue as the lowest priority write ports.
4. For every read port R_i, a bypass path is constructed to capture data consistency constraints, similar to lines 24-31 of Algorithm 2. Specifically, for any write port lagged to a less-negative degree than a given read port, we build a multiplexor chain that selects the appropriate unfolded write which is more-recent than what is reflected by the array representation, fetching the array contents only if there is no such more-recent write or if the read was not enabled or was out-of-bounds.

VI. Symbolic Row Abstraction

The array abstraction technique described in Section II-B is capable of substantially reducing verification complexity for certain classes of properties [10], though faces several limitations which we have found extensions to ameliorate.

First, in *content-addressable memory* style arrays, all rows are read every timestep, using logic downstream of the array to select which reads are actually relevant. Antecedent-conditioning properties with respect to a particular read port address-matching a modeled address is thus basically meaningless. In [10] it is instead proposed to search for a vector of registers of the width of the address, which evaluates to the address appearing at the read port referenced in the counterexample trace being refined. If found, the equality of that vector of registers (vs. the address of the read port) to the modeled address is used to antecedent-condition the properties.

This approach tends to be fragile in practice. For example, some arrays use arbitrary signals, not only registers, in their read-selection logic. Additionally, given arbitrary design styles, it may not be the case that a dedicated vector exists representing the address of relevance. We have found our array simplification techniques from Section III-C able of eliminate this concern, in reducing the number of read ports in content-addressable memory arrays and thus obviating the need for heuristics to identify useful antecedent addresses.

Second, it is often suboptimal to model a distinct address per refinement step, as doing so fails to explicitly reflect address correlation in the abstract netlist. Consider the equivalence checking of two netlists, each containing an array to abstract. The testbench itself may ensure that equivalent addresses are presented to these arrays, even if design optimizations such as retiming are used to change the timing with which relevant reads occur across these arrays. Additionally, for arrays which are fragmented to reflect circuit characteristics, *many* arrays may have correlated addresses.

A correlated-address optimization may be implemented as follows. Instead of immediately modeling a fresh address upon refinement, we first attempt to assess a relationship between the address to be refined and a previously-refined address. If a correlation is found, the newly modeled row will have its address defined as the postulated correspondence with respect to the previously-modeled address, and no antecedent-conditioning is performed for this refinement step – else this optimization would not be sound. Only if this modeling fails to block the spurious counterexample is a fresh address modeled.

Regarding postulated equivalences: often *identity* between the address of a current refinement and that of a previously-modeled row is an adequate relation. Alternatively, we have encountered equivalence checking problems where an array with a large number of rows in one netlist is replaced with multiple arrays of a smaller number of rows in another. In such cases, postulating a correspondence between an address of the larger array to an address *identical modulo the number of rows in the smaller array* is often effective.

Failure to directly model address correlation in the abstract netlist poses several verification suboptimalities. First, the abstract netlist is larger, requiring more logic to represent more modeled addresses. Second, because distinct addresses are being modeled, this lack of *address* correlation entails a loss of any *data* correlation which holds in the original netlist. For example, in equivalence checking, array data may be identical

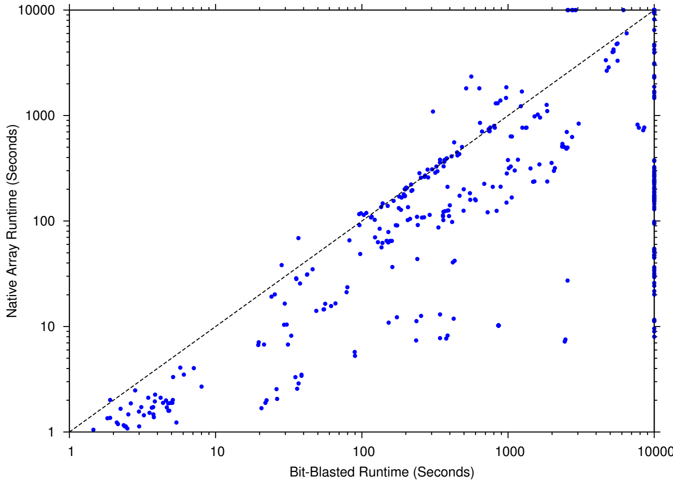

Fig. 2: Cumulative verification runtime experiments

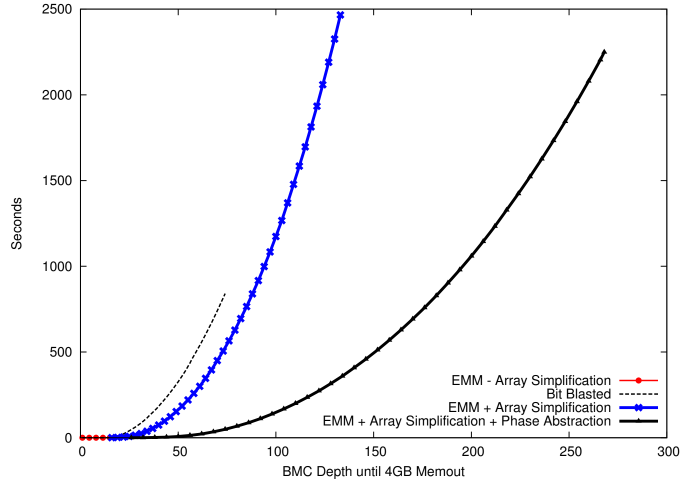

Fig. 3: Bounded Model Checking experiments

on a per-address basis across two arrays, and hence logic optimization techniques may be able to merge those arrays as being redundant. However, if a distinct address is used to abstract each array, the modeled data will differ in states for which modeled addresses differ, precluding such reductions. While such states may be irrelevant due to antecedent conditioning, it is computationally expensive to need to identify such irrelevance through the sequential observability don't care condition of the antecedent vs. being able to identify such redundancy natively using arbitrary logic optimizations.

Furthermore, this lack of modeled address correlation tends to inherently limit the subsequent effectiveness of localization abstraction [22], which eliminates irrelevant gates through replacing them with RANDOMs. E.g., the localized netlist must include enough logic in the fanin of the array addresses to establish the correlation conditions that otherwise would be natively reflected in the correlated-address abstraction.

VII. Experimental Results

In this section we experimentally demonstrate the utility of our techniques to reduce verification resources. All experiments were run on a 1.9 GHz POWER5 Processor, using the IBM internal verification toolset *SixthSense* [6].

Cumulative Impact: Given the numerous techniques presented in this paper, and their ability to synergistically enable solutions to complex problems for which standalone or bit-blasted techniques would fail, our first set of experiments in Figure 2 demonstrates their *cumulative* impact across a large set of complex non-falsifiable industrial property checking and sequential equivalence checking problems. We used a set of often-effective algorithm sequences including the simplification and abstraction techniques presented in this paper, followed by either interpolation or inductive redundancy removal, assessing their effectiveness on bit-blasted netlists vs. ones with arrays within a 10000 second timeout.

Most runtimes become significantly faster without bit-blasting, and many (108 of 810) complete that otherwise timeout. Only a small percentage witness significant slowdown with arrays; almost all of these may be turned to an advantage by fine-tuning algorithm parameters. While these experiments illustrate the profound cumulative benefit of our techniques in enhancing the capacity of state-of-the-art verification solutions, this high-level overview offers little insight into the merit of particular techniques, which we focus on below.

Array Simplification: Figure 3 shows Bounded Model Checking (BMC) performance for several runs: efficient memory model (EMM) with and without our array simplification techniques, vs. a bit-blasted representation, each run until memout. This netlist has 430373 AND gates and 21429 registers, in addition to 444 1-column, 128-row arrays, each with 128 read and write ports: a *content-addressable memory*. Our array simplification techniques from Section III-C reduce these to three 148-column 128-row arrays with one read and one write port each, using 1.3 seconds of runtime. The bit-blasted netlist has 599381 AND gates and 78209 registers.

EMM run *without* array simplification quickly completes 15 timesteps of BMC, after which a formidable resource spike is encountered due to the large number of arrays and ports – ultimately resulting in memout. The bit-blasted approach fares considerably better, completing 74 timesteps before memout. Array simplification enables EMM to yield substantially better results, completing 133 timesteps before memout. These results clearly illustrate the utility of automated techniques to convert circuit-accurate arrays to behavioral representations, without which bit-blasting may be a superior solution.

Phase Abstraction: Recall from Section IV that phase abstraction multiplies the number of read and write ports by its unfolding depth. However, for every netlist we have encountered for which phase abstraction reduced clocking complexity, array simplifications eliminate these duplicated ports as irrelevant (e.g., enables being conjuncted with a clock signal) or redundant (e.g., identical reads / writes occur across consecutive clock phases). Phase abstraction plus array simplification may thus *quarter* (or better) the size of EMM modelings through halving (or better) the number of read ports and write ports compared across a specific unfolding depth. This benefit is illustrated in Figure 3, where modulo-2 phase abstraction enabled the completion of 268 BMC timesteps before memout, requiring only 0.5 seconds of reduction time.

Correlated Row Abstraction: This netlist also illustrates the value of the correlated-address abstraction techniques discussed in Section VI. We focused on a single parity-style property. If applying the technique directly from [10], each of the 444 1-column arrays requires the modeling of a single row. This yields a substantial reduction; seven registers to represent each modeled address, and one for the modeled data, per array – vs. 128 registers for a precise bit-blasting. However, the large number of arrays entails a large collective abstraction size. Furthermore, the failure to model address correlation hampered subsequent verification: localization could not reduce the resulting netlist below 3241 registers, which we could not verify within an eight hour timeout.

In contrast, using our address-correlation optimization, only three abstract addresses need to be modeled across *all* 444 arrays. Localization and logic optimizations were able to reduce this address-correlated abstraction to only 32 registers, which interpolation solved within one second of runtime.

Localization: We have noted numerous additional benefits of applying localization without bit-blasting: **(1)** BMC tends to be much more efficient; **(2)** far fewer refinements need to be performed given fewer gates in the netlist; and **(3)** fewer necessary refinements entails fewer inevitable *mistakes* which unnecessary bloat the abstract netlist.

Sequential Redundancy Identification: To illustrate the benefit of identifying redundancies without operating on a bit-blasted netlist, we detail a sequential equivalence checking (SEC) problem involving a DRAM. This DRAM implementation and its redundancy scheme (used for fault-tolerance) was altered, yet in a way that preserved input-to-output behavior. One netlist has sixty-four 9-column, 128-row arrays; the other has four 144-column, 128-row arrays. The overall SEC problem additionally has 95786 AND gates and 5286 registers surrounding these arrays. Our redundancy identification framework from Section III-B is able to automatically identify 572 column equivalences and 1238 register equivalences inductively in 851 seconds. However, given changes in the fault-tolerance scheme, four columns and 2810 registers did not correspond hence the SEC problem remained unsolved; a combination of localization and interpolation on the redundancy-eliminated netlist was necessary to complete the overall SEC problem with a total runtime of 34 minutes. The bit-blasted variant has 770215 AND gates and 152710 registers, for which we were unable to even prove the equivalent sequential elements (without tedious *manual* correlation of array cells [19], [23]) given 48 hours of runtime.

Overall, redundancy identification substantially benefits without bit-blasting due to **(1)** speedups to BMC and simulation used to filter invalid candidate equivalences, and to induction used in proofs, and **(2)** requiring far fewer computations at the granularity of columns vs. cells.

VIII. Conclusion

Arrays are ubiquitous in industrial hardware designs, along with many control- and performance-related artifacts which practically mandate the availability of a large set of synergistic algorithms to enable automated verification. In this paper, we extend numerous traditionally bit-level state-of-the-art model checking and equivalence checking algorithms to support designs with arrays, and introduce automated techniques to transform arrays of circuit-accurate to behavioral syntax, enabling the use of higher-level reasoning techniques on problems of otherwise-unsuitable syntax. Nearly all algorithms used in a state-of-the-art model checker (simulators, logic optimization and abstraction techniques, isomorphism detection, . . .) tend to significantly benefit from operating on the smaller non-bit-blasted netlist, in addition to the even more profound benefits that dedicated array reasoning techniques may offer. These techniques have collectively enabled dramatic scalability enhancements to our model checking and equivalence checking solutions, enabling automation for verification tasks that otherwise would have required significant manual guidance.

Acknowledgments: The authors wish to thank Per Bjesse for helpful feedback on this work.

References

[1] P. Bjesse and J. Kukula, "Automatic generalized phase abstraction for formal verification," in *ICCAD*, Nov. 2005.
[2] A. Kuehlmann and J. Baumgartner, "Transformation-based verification using generalized retiming," in *CAV*, July 2001.
[3] A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification," *TCAD*, vol. 21, Dec. 2002.
[4] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis," in *DAC*, 2006.
[5] H. Mony, J. Baumgartner, A. Mishchenko, and R. Brayton, "Speculative-reduction based scalable redundancy identification," in *DATE*, Apr. 2009.
[6] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, "Scalable automated verification via expert-system guided transformations," in *FMCAD*, Nov. 2004.
[7] Berkeley Logic and Synthesis Group, *ABC: A System for Sequential Synthesis and Verification*. http://www.eecs.berkeley.edu/alanmi/abc.
[8] M. N. Velev and R. E. Bryant, "Efficient modeling of memory arrays in symbolic ternary simulation," in *TACAS*, March 1998.
[9] M. Ganai, A. Gupta, and P. Ashar, "Verification of embedded memory systems using efficient memory modeling," in *DATE*, March 2005.
[10] P. Bjesse, "Word-level sequential memory abstraction for model checking," in *FMCAD*, Nov. 2008.
[11] J. McCarthy, "Towards a mathematical theory of computation," in *IFIP Congress*, 1962.
[12] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, Apr. 1965.
[13] L. de Moura and N. Bjorner, "Generalized and efficient array decision procedures," in *FMCAD*, Nov. 2009.
[14] K. McMillan, "A methodology for hardware verification using compositional model checking," *Cadence Technical Report*, April 1999.
[15] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *TACAS*, March 1999.
[16] M. Sheeran, S. Singh, and G. Stalmarck, "Checking safety properties using induction and a SAT-solver," in *FMCAD*, Nov. 2000.
[17] P. Manolios, S. Srinivasan, and D. Vroon, "Automatic memory reductions for RTL model verification," in *ICCAD*, Nov. 2006.
[18] Q. Zhu, N. Kitchen, A. Kuehlmann, and A. Sangiovanni-Vincentelli, "SAT sweeping with local observability don't-cares," in *DAC*, July 2006.
[19] A. Koelbl, J. Burch, and C. Pixley, "Memory modeling in ESL-RTL equivalence checking," in *DAC*, June 2007.
[20] K. McMillan, "Interpolation and SAT-based model checking," in *CAV*, July 2003.
[21] M. Case, H. Mony, J. Baumgartner, and R. Kanzelman, "Enhanced verification through temporal decomposition," in *FMCAD*, Nov. 2009.
[22] P. Chauhan et al., "Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis," in *FMCAD*, 2002.
[23] Z. Khasidashvili, M. Kinanah, and A. Voronkov, "Verifying equivalence of memories using a first order logic theorem prover," in *FMCAD*, 2009.