EPiC Series in Computing

Volume 100, 2024, Pages 136–146

Proceedings of 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning

EPiC Computing

Efficient Simulation for Hardware Model Checking

Joseph Tafese and Arie Gurfinkel

University of Waterloo, Waterloo, Ontario, Canada
{jetafese, agurfink}@uwaterloo.ca

Abstract

Simulation is an important aspect of model checking, serving as an invaluable preprocessing step that can quickly generate a set of reachable states. This is evident in model checking tools at the Hardware Model Checking Competitions, where BTOR2 is used to represent verification problems. Recently, BTOR2MLIR was introduced as a novel format for representing safety and correctness constraints for hardware circuits. It provides an executable semantics for circuits represented in BTOR2 by producing an equivalent program in LLVM-IR. One challenge in simulating BTOR2 circuits is the use of persistent (i.e., immutable) arrays to represent memory. Persistent arrays work well for symbolic reasoning in SMT but they require copy-on-write semantics when being simulated natively. We provide an algorithm for converting persistent arrays to transient (i.e., mutable) arrays with efficient native execution. This approach is implemented in BTOR2MLIR, which opens the door for rapid prototyping, dynamic verification techniques and random testing using established tool chains such as LibFuzzer and KLEE. Our evaluation shows that our approach, when compared with BTORSIM, has a speedup of three orders of magnitude when safety properties are trivial, and at least one order of magnitude when constraints are disabled.

1 Introduction

Model Checking [7] has been an important part of the hardware verification pipeline. Given a circuit and its specifications, model checking exhaustively searches through the circuit state space to determine if any property is violated. This is known to be expensive for circuits with a large number of states. Simulation as a preprocessing step can provide a fast, yet incomplete, method of exploring the state space defined by a circuit. In a well integrated pipeline [18], the states that are found to be reachable by the simulation effort can be used to guide the model checking component. These semi-formal [11] verification methods have been used to verify microprocessors [4] by striking a balance between the speed of simulation and the rigor of model checking. We are interested in exploring semi-formal verification through the simulation of formal designs with native code. This is not only useful as a preprocessing step for model checking, but it is also a valuable addition to a verification pipeline.

Formal designs are tailored to capture specifications in a format that benefits downstream solvers. Adjacent fields have seen this with the adaption of SMT-LIB [3] for SMT solving and Conjunctive Normal Form for SAT solving. In the domain of hardware verification, BTOR2 [12] has risen to be a popular format for word-level verification as seen in the Hardware Model Checking competitions [5]. Benchmarks in these competitions span two categories: bit-vectors and arrays. We focus our attention on the simulation of arrays. A robust memory representation is a prerequisite for model checking and simulation alike. For instance, Btor2 has persistent arrays (i.e., immutable) that are designed to correspond to Smt-Lib [3] arrays. This is a convenient logical representation of memory for the underlying solvers. Tools like BtorSim [2] simulate circuits with arrays using an interpreter that maintains a map from array indices to values. For native simulation, however, enforcing copy-on-write semantics can be an expensive ordeal. It is therefore interesting for us to convert as many persistent arrays to transient (i.e., mutable) arrays as possible. This allows us to reap the benefits of compiling to native code while maximizing the speed at which simulations are executed.

To this end, we extend Btor2 with new operations that are applied to transient arrays. We provide the semantics for existing Btor2 operations as well as our extensions. Using the new operations, we enable an efficient simulation of Btor2 circuits by contributing: a sound but incomplete algorithm for converting persistent arrays to transient arrays and translation passes that generate executable native programs in LLVM-IR. The implementation has been incorporated into Btor2MLIR [16] to build a verification pipeline that produces native code for the purpose of simulation.

The rest of the paper is organized as follows: we provide the necessary background in Section 2, present semantics for Btor2 and new operations for transient arrays in Section 3, our transformation algorithm in Section 4, an evaluation of our techniques in Section 5 and our conclusions in Section 6.

2 Background

The dominant format in the Hardware Model Checking competition has been Btor2. It is used to represent sequential circuits over Smt-Lib theories of BitVec [14] and Arrays [17]. The formal syntax is provided in [12]. A circuit consists of sort definitions, state definitions, inputs, gates, safety and liveness properties, and initial state and next state functions. We illustrate Btor2 using a four bit counter, C (see Fig. 1a), that uses an array to store its current value. The example does not show the use of liveness properties or inputs. Observe that each line in C is referred to by a line identifier (*lid*) that represents a sort (*sid*) or a node (*nid*). For a given line, a *sid* gives the sort of the operation and arguments are given by their *nid*. For a *nid* to be used as an argument, the syntax requires that it refers to a value producing gate i.e., it cannot refer to a property or function.

Sort definitions are shown in line 1 for a bit-vector of bit width 4 ($bv4$) and line 2 for an array of index and element sort $bv4$. Constants $b0001$ (resp. $b1000$) are defined in line 3 (resp. line 4). A state definition for an array is shown in line 5. Btor2 has an implicit clock that simulates the execution of a circuit. Therefore, the initial (resp. next) state function is used to initialize (resp. update) a state given a value. The initial state function (line 6) sets all the indices of our state array (*nid* 5) to $b0001$. This is run once at the beginning of circuit execution. C uses the $8th$ (*nid* 4) index of the state array as the counter by reading the current value (*nid* 7), incrementing it (*nid* 9) and storing it (*nid* 10). Then, depending on whether the safety property has been violated (*nid* 13), either the old state array (*nid* 5) or the new array (*nid* 10) is chosen and stored at *nid* 14. The next state function (line 15) updates the state with *nid* 14 at the end of each cycle. The safety property (line 17) asserts that the counter is not 15 ($b1111$). It is checked using the value of the counter at the beginning of the cycle (*nid* 7).

```
; BTOR counter
; using arrays
1 sort bitvec 4
2 sort array 1 1
3 one 1
4 constd 1 8
5 state 2
6 init 2 5 3
7 read 1 5 4
8 one 1
9 add 1 7 8
10 write 2 5 4 9
11 ones 1
12 sort bitvec 1
13 neq 12 7 11
14 ite 2 13 10 5
15 next 2 5 14
16 eq 12 7 11
17 bad 16
```

(a) Counter in Btor2.

```
module {
  func @main() {
    %0 = constant 1 : bv4
    %1 = array %0 : a<4,4>
    br ^bb1(%1 : a<4,4>)
  ^bb1(%2):
    %3 = constant 1 : bv4
    %4 = constant 8 : bv4
    %5 = read %2[%4] : bv4
    %6 = add %5, %3 : bv4
    %7 = write %6,%2[%4]: a<4,4>
    %8 = constant 15 : bv4
    %9 = cmp ne, %5, %8 : bv1
    %10 = ite %9,%7,%2 : a<4,4>
    %11 = cmp eq, %5, %8 : bv1
    assert_not(%11)
    br ^bb1(%10 : a<4,4>)
  }
}
```

(b) Counter in Btor Dialect.

```
...
define void @main() {
  br label %1
1:       ; preds = %10, %0
  %2 = phi <16xi4>[%7, %10],
         [<1,..>, %0]
  %3 = extractelement %2, 8
  %4 = add i4 %3, 1
  %5 = icmp ne i4 %3, 15
  %6 = insertelement %2, %4, 8
  %7 = select %5, %6, ... %2
  %8 = icmp eq i4 %3, 15
  %9 = xor i1 %8, true
  br i1 %9, label %10, label %11
10:      ; preds = %1
  br label %1
11:      ; preds = %1
  unreachable
}
```

(c) Counter in LLVM-IR.

Figure 1: Running Problem: Btor2 to Btor Dialect

Btor2MLIR: A format and toolchain for hardware verification that is build on the Multi-Level Intermediate Representation (MLIR) [10] framework. It is a publicly available project [1] that makes use of MLIR parsers, generators and optimizations to support efficient compilation of Btor2 circuits to executable LLVM-IR programs. This enables the use of software model checkers [8,13], dynamic verification [6,15] and static analysis [9] techniques. It also facilitates analysis and optimization at different levels of abstraction. For example, given the running problem (Fig. 1a), we generate Btor Dialect and LLVM-IR. We show the simplified versions of these dialects in Fig. 1b and Fig. 1c respectively. A detailed presentation of the dialects and their conversions can be found in [16].

3 Semantics

In this section, we introduce formal semantics for Btor2 extended with transient write operations. To this end, we capture the behaviour of a Btor2 circuit in a transition system, $T = \langle St,\ I,\ O,\ \mathit{INIT},\ \mathit{TR}\rangle$. St is a set of states, I represents the set of possible inputs and O represents the set possible outputs. INIT is a subset of St that represents the set of initial states for the system. TR represents the relationship between states such that a directed edge (s, i, s', o) relates state s to state s' given input $i \in I$ and producing output $o \in O$. Let $r = (s_1, i_1, o_1), (s_2, i_2, o_2), \ldots$ be an infinite sequence of tuples of states, inputs and outputs that represents the run of a transition system. It is a feasible run if s_1 satisfies INIT, and, for all i, (s_i, i_i, s_{i+1}, o_i) is in TR. The semantics of a circuit A is the set $L(A)$ of all its feasible runs, called the language of A. Two circuits, A and B, are *observationally equivalent* iff they have the same languages, i.e., $L(A) = L(B)$.

Let $\pi : \mathit{Nid} \to \mathit{Val}$ be an evaluation context – a map from node identifiers Nid to values Val. The domain of values is $\mathit{Val} = \mathit{Bv}(\mathit{BitVec}) \mid \mathit{Array}(\mathit{Arr}) \mid \mathit{ArrayRef}(\mathit{Nid})$, where BitVec, Arr, and Nid are, respectively, the sorts for bit-vectors, arrays, and node identifiers. Let $\texttt{dec} : (\mathit{Nid} \to \mathit{Val}) \to St$ to decode an evaluation context into a state in St. The opposite direction is performed by function $\texttt{enc} : St \to (\mathit{Nid} \to \mathit{Val})$. Note that dec is bijective, and enc its inverse, since there is a unique encoding of the state of an evaluation context i.e., $\pi = \texttt{enc}(\texttt{dec}(\pi))$.

Inputs are encoded into the evaluation context with $\texttt{enci} : I \to (\mathit{Nid} \to \mathit{Val})$. Outputs are extracted from the evaluation context with $\texttt{deco} : (\mathit{Nid} \to \mathit{Val}) \to O$. An evaluation context for a state, i.e. $\texttt{enc}(s)$, can be combined with a non-intersecting evaluation context for inputs, i.e. $\texttt{enci}(i)$, with $(+)$ where the values for inputs are added to the encoding of a state. For example, in $\texttt{enc}(s) + \texttt{enci}(i)$, the resulting evaluation context is $\texttt{enc}(s)$ with the values from $\texttt{enci}(i)$ added to it.

We describe the semantics of Btor2 operations relative to Smt-Lib theories of BitVec and Array, as shown in Fig. 2. The definition of the semantics relies on the helper functions $\mathit{root} : \mathit{Nid} \times (\mathit{Nid} \to \mathit{Val}) \to \mathit{Nid}$ and $\mathit{IsArrayRef} : \mathit{Val} \times \mathit{Nid} \to \mathit{Bool}$ defined in Fig. 2b. We use the notation $[\![B]\!]_{mode}(\pi)$ in Fig. 2a to represent the evaluation of circuit B under a specific mode. For example, we use the rules in Fig. 2c to determine the initial states of circuit B. More specifically, we get the initial states of T by evaluating every line in B using the general rules in Fig. 2 and the mode specific rules in Fig. 2c. Similarly, we use $[\![B]\!]_{next}(\pi)$ to represent the evaluation of circuit B using the mode specific rules and an Offset in Fig. 2d. Offset is the size of circuit B and it allows us to hold new values for an Nid in π. We assume that a given Nid is assigned at most one new value per cycle.

Btor2 operations can represent circuit gates or circuit level functions. There are two important functions in Btor2: `init` and `next`. These define the initial state function and the next state function respectively. To simplify the presentation of our semantics, we use the notation $\xrightarrow{B}_{init}$ and $\xrightarrow{B}_{next}$ to represent the rules for evaluating a circuit B in the respective mode. The sets in T are defined as:

$$\mathit{INIT} = \{s \in \mathit{St} \mid \quad \xrightarrow{B}_{init} t \wedge s = \texttt{dec}(t)\}$$

$$\mathit{TR} = \{s, s' \in \mathit{St}, i \in I, o \in O \mid \texttt{enc}(s) + \texttt{enci}(i) \xrightarrow{B}_{next} t \wedge s' = \texttt{dec}(t) \wedge o = \texttt{deco}(t)\}$$

Btor2 has three instructions that work with array values: `read, write` and `ite`. The formal semantics are presented in Fig. 2e. For example, consider $a' = \texttt{write}(a, x, v)$, where a and a' are indexes of arrays in the execution context π. After `write` executes, $\pi(a)$ and $\pi(a')$ refer to the original and updated array respectively. This is consistent with the behaviour of persistent arrays, therefore, we map `write` to Smt-Lib store in our semantics. `read` and `ite` are mapped to Smt-Lib select and ite with the results stored at the operation *nid*.

Our key idea is to extend the semantics of Btor2 with transient array operations: `write_mut` and `write_mutz` to represent, respectively, unconditional and conditional mutable writes. Unlike their persistent counterparts, these operations update an array in place. We describe the intuition behind the semantics with the `write_mut` operation since `write_mutz` is similar. Consider an instruction $a' = \texttt{write\_mut}(a, x, v)$, where a and a' are indexes of arrays in the execution context π. After `write_mut` executes, $\pi(a)$ and $\pi(a')$ both refer to the same array. Moreover, that array is the same as the array $\pi(a)$ before the execution, but with value v stored at index x. To this end, we add the notion of *pointing* in the execution context π, by extending allowed values with *ArrayRef*. Intuitively, the value $\mathit{ArrayRef}(i)$ represents a *pointer* to an array at location i in the context π. This allows us to represent references to an array while preserving support for existing array operations. In our example above, at the end of execution, $\pi(a') = \mathit{ArrayRef}(a)$, $\pi(a) = \mathit{Array}(u)$, and u is the new array value. In our formal semantics, we resolve (or dereference) pointers using a helper *root*, that returns the index of an *Array* in π pointed to by the corresponding reference. Note that in our semantics, every *ArrayRef* is one hop away from an array value. That is, $\mathit{root}(d, \pi)$ returns d if $\pi(d)$ is an *Array*, or v if $\pi(d) = \mathit{ArrayRef}(v)$ and, therefore, $\pi(v)$ must be an *Array*.

Conditional writes in Btor2 are a result of combining the `write` and `ite` instructions. To support in place conditional writes, we extended Btor2 with `write_mutz`. The formal

$$\frac{t = \pi \quad [\![B]\!]_{mode}(\pi) \rightsquigarrow \pi' \quad t' = \pi'}{t \xrightarrow{B}_{mode} t'} \qquad \frac{\exists \pi_1, \ldots, \pi_{|B|}\ \forall i. [\![B_i]\!](\pi_i) \rightsquigarrow_{mode} \pi_{i+1} \ \ \pi' = \pi_{|B|}}{[\![B]\!]_{mode}(\pi) \rightsquigarrow \pi'}$$

(a) Semantics of whole-circuit evaluation

$$root(d, \pi) = \begin{cases} d & \text{if } \mathit{IsArray}(\pi(d)) \\ v & \text{if } \mathit{IsArrayRef}(\pi(d), v) \\ d & \text{otherwise} \end{cases}$$

$$\mathit{IsArrayRef}(u, v) = \begin{cases} \mathit{True} & \text{if } u = \mathit{ArrayRef}(v) \\ \mathit{False} & \text{otherwise} \end{cases}$$

(b) Helper functions.

$$[\![\texttt{next}(s, v)]\!](\pi) \rightsquigarrow_{init} \pi$$

$$\frac{\mathit{IsBv}(\pi(s))}{[\![\texttt{init}(s, v)]\!](\pi) \rightsquigarrow_{init} \pi[s \coloneqq \pi(v)]}$$

$$\frac{a = \pi(root(s, \pi)) \quad \forall i \in \mathbb{N}_{<|a|}, a[i] = \pi(v)}{[\![\texttt{init}(s, v)]\!](\pi) \rightsquigarrow_{init} \pi[s \coloneqq a]}$$

(c) Semantics of *init* mode.

$$\frac{}{[\![\texttt{init}(s, v)]\!](\pi) \rightsquigarrow_{next} \pi} \qquad \frac{a = \pi(root(v, \pi)) \quad s' \coloneqq s + \textsc{Offset}}{[\![\texttt{next}(s, v)]\!](\pi) \rightsquigarrow_{next} \pi[s' \coloneqq a]}$$

(d) Semantics of *next* mode.

$$\frac{m = \pi(root(a, \pi)) \qquad v = \textsc{select}(m, \pi(x))}{[\![n = \texttt{read}(a, x)]\!](\pi) \rightsquigarrow_{mode} \pi[n \coloneqq v]}$$

$$\frac{u = \pi(root(a, \pi)) \qquad v = \pi(root(b, \pi))}{[\![n = \texttt{ite}(c, a, b)]\!](\pi) \rightsquigarrow_{mode} \pi[n \coloneqq \textsc{ite}(\pi(c), u, v)]}$$

$$\frac{m = \pi(root(a, \pi))}{[\![n = \texttt{write}(a, x, v)]\!](\pi) \rightsquigarrow_{mode} \pi[n \coloneqq \textsc{store}(m, \pi(x), \pi(v))]}$$

$$\frac{p = root(a, \pi) \qquad m = \textsc{store}(\pi(p), \pi(x), \pi(v)) \qquad \pi' = \pi[p \coloneqq m, a \coloneqq m]}{[\![n = \texttt{write\_mut}(a, x, v)]\!](\pi) \rightsquigarrow_{mode} \pi'[n \coloneqq p]}$$

$$\frac{p = root(a, \pi) \qquad m' = \textsc{store}(\pi(p), \pi(x), \pi(v)) \qquad \pi' = \textsc{ite}(\pi(c) = 0, \pi[p \coloneqq m', a \coloneqq m'], \pi)}{[\![n = \texttt{write\_mutz}(c, a, x, v)]\!](\pi) \rightsquigarrow_{mode} \pi'[n \coloneqq p]}$$

(e) Semantics of array operations

$$\frac{\exists v.\ \mathit{IsBv}(v)}{[\![n = \texttt{input}]\!](\pi) \rightsquigarrow \pi[n \coloneqq v]} \qquad \frac{v = \texttt{bv\_op}(\pi(a))}{[\![n = \texttt{op}(a)]\!](\pi) \rightsquigarrow_{mode} \pi[n \coloneqq v]}$$

$$\frac{v = \texttt{bv\_op}(\pi(a), \pi(b))}{[\![n = \texttt{op}(a, b)]\!](\pi) \rightsquigarrow_{mode} \pi[n \coloneqq v]} \qquad \frac{v = \texttt{bv\_op}(\pi(a), \pi(b), \pi(c))}{[\![n = \texttt{op}(a, b, c)]\!](\pi) \rightsquigarrow_{mode} \pi[n \coloneqq v]}$$

(f) Semantics of bit-vector operations

Figure 2: Semantics of Btor2 operations.

semantics for `write_mut` and `write_mutz` are shown in Fig. 2e. Observe that the instruction $a' = \texttt{write\_mutz}(c, a, x, v)$ ensures that $\pi(a)$ and $\pi(a')$ refer to the same array. If the condition c is zero, $\pi(a')$ at index x gets value v. Otherwise, the instruction does nothing. This pattern should be familiar to those acquainted with the `jmpz` instruction. The rest of the operations in Btor2 are presented in Fig. 2f, separated by the number of arguments they take. None of these operate on arrays. Unary operations (*op*) are mapped to their corresponding operation in Smt-Lib (`bv_op`), and their results stored in π at the operation *nid*. Binary and Ternary operations follow the same pattern. At the end of a cycle, the next state function computes new values for each state using π. For array states that are modified with transient writes, the *ArrayRef* is resolved using *root* before being assigned to the next state.

To illustrate the semantics, consider the two Btor2 circuits shown in Fig. 3. The circuit on the left (C_1) uses `write` while the one on the right (C_2) uses the new conditional write `write_mutz`. Both circuits create and initialize an array state, write to the array, read from the array state and output the read value. Let π_1 (resp. π_2) be the evaluation context for C_1 (resp. C_2). Assume that the evaluation contexts have the array state initialized according to the initial state function. In C_1, *root* returns 5 when `write` is evaluated under our semantics and the result is referred to as $\pi(7)$. Note that $root(7, \pi) \neq root(5, \pi)$. Therefore, when C_1 is evaluated under our semantics, *root* never returns an id that resolved to an *ArrayRef* in π. Hence, the values that are written are not visible at $\pi(5)$. Now consider C_2, where *root* returns 5 when `write_mutz` is evaluated and the result is referred to as $\pi(7)$. Unlike the evaluation of C_1, $root(7, \pi) = root(5, \pi)$. Therefore, unlike the evaluation of C_1, the value written at $\pi(7)$ are also visible at $\pi(5)$ in the evaluation of C_2.

4 Persistent to Transient Arrays

Btor2 `write` operations use persistent arrays which, as shown in the previous section, have copy-on-write semantics. This is expensive to simulate when large arrays are involved, especially when the copied array is not used by future operations. For example, if the copied array is never used, it does not need to be preserved, and we can replace `write` with `write_mut`. In fact, we have found two common patterns where significant gains can be made: unconditional writes and conditional writes. We offer the intuition behind our approach by breaking down these cases. In the first case, we observe that the copied array is not used after a `write` operation. If this pattern is detected, we can perform the replacement discussed above. In the case of conditional writes, we observe that an array is copied, a value is written into it and an `ite` operation is used to determine which array will be used going forward. If this pattern is detected, then we can perform a replacement of the `write` and `ite` operations with a single `write_mutz` operation. In both cases, under some conditions, we can omit the copy altogether by using transient arrays and their corresponding operations.

To present the conditions under which our transformation takes place, we setup some useful terminology. Let *op* represent an operation in Btor2, and $id(op)$ its unique identifier (line id). Then, for two operations op_1 and op_2, let $uses(op_2, op_1)$ be a function that returns true iff op_2 uses the result of op_1. In other words, op_2 has $id(op_1)$ as an argument. Using this, we construct a def-use graph $G = (V, E)$, where V is the set of nodes corresponding to an array operation and E the set of use relationships between nodes. Thus, for $\{u, v\} \in V, (u, v) \in E \iff uses(v, u)$. There are four array operations: `state, write, ite` and `read`. The first three operations are used to define arrays and the last three operations use arrays. Let us call `write` and `ite` hybrid operations since they do both. Let $u \in V$ be a node corresponding to an array valued state, and G_u the largest connected component containing u. We assume that the operations of a

```
1 sort bitvec 4
2 sort array 1 1
3 one 1
4 constd 1 8
5 state 2
6 init 2 5 3
7 read 1 5 4
8 one 1
9 add 1 7 8
10 write 2 5 4 9
11 ones 1
12 sort bitvec 1
13 neq 12 7 11
14 ite 2 13 5 10 ; replaced -->
15 next 2 5 14
16 eq 12 7 11
17 bad 16
```

```
1 sort bitvec 4
2 sort array 1 1
3 one 1
4 constd 1 8
5 state 2
6 init 2 5 3
7 read 1 5 4
8 one 1
9 add 1 7 8
10 write 2 5 4 9
11 ones 1
12 sort bitvec 1
13 neq 12 7 11
14 write_mutz 2 13 5 4 9
15 next 2 5 14
16 eq 12 7 11
17 bad 16
```

Figure 3: Comparing running example using `write` vs `write_mutz`.

Figure 4: Graph representation for running example.

circuit are sorted in topological order relative to the def-use graph. Under these conditions, we say that G_u represents an array group.

Our goal is to replace all hybrid operations in G_u with `write_mut` and `write_mutz`, as appropriate. We illustrate this using our running example (left of Fig. 3) where u represents the state operation in line 5. The array group, G_u, is shown on the left of Fig. 4. We can see that G_u has four nodes corresponding to the array operations in the circuit, and each use is represented with an edge, as expected. We would like to replace each hybrid operation in G_u such that `write` and `ite` are replaced with `write_mutz`. To do this safely, we use a common definition of liveness, i.e., the result of an array operation with identifier v is live at location w if it is defined before w and used after w.

We now present the conditions under which it is legal to transform an array group (e.g., G_u) by replacing its hybrid operations. Condition 4.1 relies on the fact that an operation result cannot be resurrected. Once it is not live at a location, it is not live in all future locations. Therefore, storage can be transferred from one member of a component to its successor. These are the conditions under which we can replace `write` with `write_mut`. Condition 4.2 relies on the fact that a conditional write, represented with a combination of `write` and `ite` operations, will result in a connected component that does not satisfy our first condition. Note that it would require at most two members of the component are live at a given location and time. If the connected component satisfies our first condition when the `ite` operation is removed, we can replace the conditional write with `write_mutz`.

Condition 4.1. *Let C be a connected component of G that has at least one* `write` *operation. The* `write` *operations in C can be replaced with* `write_mut` *if for every location l, only one member of the component is live at any one time.*

Condition 4.2. *Let C be a connected component of G where for every location l, at most two*

Algorithm 1: Transformation Algorithm.

```
Function transform(Q : Btor2 Circuit, G = ⟨V, E⟩ : def-use graph for Q):
    Q' := Q;
    for C = ⟨V_C, E_C⟩ ∈ G do
        if Cond2(C) then
            for (u, v) ∈ E_C do
                if u = write and v = ite then
                    /* u = write arr, idx, val; v = ite c, id(u), arr        */
                    arr, idx, val := u; c, _, _ := v;
                    Q'[id(v) := write_mutz c, arr, idx, val, arr];
        if Cond1(C) then
            for v ∈ V_C do
                if v = write then
                    /* v = write arr, idx, val                               */
                    arr, idx, val := v;
                    Q'[id(v) := write_mut arr, idx, val];
    return Q';
```

members of the component are live at any one time. Let w be a `write` *operation and t be an* `ite` *operation in C. Then, if $C \setminus t$ satisfies Condition 4.1 and uses(t, w) is true, t can be replaced with* `write_mutz`.

We present Algorithm 1 using $\langle Q, G\rangle$ as input, where Q is a Btor2 circuit and G is its corresponding def-use graph. Let *Cond1*(C) (resp. *Cond2*(C)) be a function that take a component C, of G, and evaluates the condition described in Condition 4.1 (resp. Condition 4.2). Algorithm 1 iterates over every component in G and checks if it satisfies *Cond2*(C) or *Cond1*(C). It is clear that the two conditions are mutually exclusive, hence, a component will satisfy at most one of our conditions.

We illustrate the algorithm using the running example on the left of Fig. 3 and its corresponding def-use graph on the left of Fig. 4. Observe that there is only one component in G and it does not satisfy *Cond1*(C). Therefore, since *Cond2*(C) is satisfied, Algorithm 1 iterates over every edge in the component to find a `write` that is succeeded by an `ite` operation. Then, the `ite` operation is replaced with `write_mutz` and the change is persisted in an updated circuit. Let Q' be the transformed circuit that results from running Algorithm 1 on Q. Note that the only update to Q happens at index $id(v)$. Therefore, Q and Q' differ only at line 14, where the `ite` operation of Q is replaced with `write_mutz` in Q'. It is important to note that the `write` operation at line 10 of Q has no uses in Q', i.e., the result of `write` is dead. We show Q' on the right of Fig. 3 and G', its corresponding def-use graph, on the right of Fig. 4. Note that, under the semantics we have provided for Btor2, Q' is equivalent to Q.

In the case where Q satisfies *Cond1*(C), Algorithm 1 iterates over every vertex in the component to identify a `write`. Then, the `write` operation is replaced with its mutable counterpart, `write_mutz`. Let Q' be the resulting transformed circuit. Since the update only happens at most once for an array in a component, Q and Q' will only differ at these locations. Similar to the previous case, under the semantics we have provided for Btor2, Q and Q' are equivalent.

Theorem 1. *Let Q be a* Btor2 *circuit. Let Q' be the result of Algorithm 1. Q and Q' are observationally equivalent.*

	BtorSim	Btor2MLIR	speedup	BtorSim	Btor2MLIR	speedup
	constraints disabled			**constraints enabled**		
w_18A	991	9 544	17	28	4 222	8
w_19A	409	2 577	7	385	1 752	4
w_19B	1 895	32 138	17	1	2 8747	28 747
w_19C	201	2 616	23	2	1	1
19_mann	3 429	13 697	4	1	1	1
20_mann	4 770	79 966	156	1 738	43 843	10

Table 1: Mean running time of BtorSim vs Btor2MLIR in execution cycles per second.

5 Implementation and Evaluation

Implementation. We implement Algorithm 1 in Btor2MLIR to convert persistent arrays in Btor2 to transient arrays in LLVM-IR. Our implementation is encapsulated in the `convert-btor-to-memref` pass i.e., a conversion from Btor Dialect to MemRef Dialect. An MLIR dialect is designed to capture the operations and types of a language through its syntax, instructions and properties. MemRef is an MLIR dialect for representing operations on transient array types and Btor Dialect [16] is used to represent Btor2 circuits. In addition to MemRef, there is a Vector Dialect for representing array operations on Single Instruction/Multiple Data (SIMD) vectors (i.e., persistent arrays). These define operations that use register allocated memory and can be useful for efficiently modelling small arrays. We implement this in the `convert-btor-to-vector` pass.

A core contribution of MLIR is that its users can define dialects that meet their needs and interoperate with other dialects using conversion and translation passes. Conversion (resp. translation) passes represent a conversion from a dialect to another dialect (resp. target language). This is why we use existing conversion passes for translating Btor2 to Btor Dialect, MemRef Dialect to LLVM Dialect and Btor Dialect to LLVM Dialect. Once all the passes have been run, we use a translation pass that generates LLVM-IR from LLVM Dialect. Working in the MLIR framework makes it easy to manipulate the intermediate representation structure for def-use analysis when checking the conditions in Algorithm 1 and pattern-based rewrites. Pattern-bases rewrites are how MLIR matches the operations to be transformed with their respective transformation. For example, to convert `write_mut` to a store in MemRef Dialect, we provide a function that performs the conversion and a pass that marks all `write_mut` operations for conversion.

Evaluation. An important metric in evaluating the efficacy of our approach is cycles per second. A fast simulation approach is beneficial because the user can explore more cycles, and potentially more states. In contrast to model checking, simulation does not exhaustively search a state space. Therefore, having a fast simulator makes guiding a simulation more effective than the slower counterpart. The goal of our evaluation is to show that Btor2MLIR makes it easy to produce efficient LLVM programs that can be compiled and executed for the purpose of simulation. In the future, we plan to do a case study using LLVM-based analysis tools, such as symbolic execution engine KLEE [6], and fuzzing framework LibFuzzer [15].

For the evaluation, we have chosen the array category of Btor2 benchmarks from the most recent Hardware Model Checking Competition (HWMCC) [5]. All our experiments are run on a Linux machine with x86_64 architecture, with a timeout of 1 second and memory limit of 65

GB. These results are reported in Table 1, grouped by competition contributor and whether constraints are enabled or disabled. For all benchmarks, safety properties are set to false so that we can maximize the number of cycles for both tools.

BtorSim was chosen because it is well integrated with the HWMCC environment and is specifically designed for Btor2. We evaluate BtorSim and show the results in the first column of Table 1. For each category, we show the mean number of cycles per second. For example, for the case where constraints are disabled, the `w_19B` category has an average of 1 895 cycles per second. Btor2MLIR is evaluated by compiling the harnessed LLVM-IR output with clang to create an executable that generates random values for inputs and reports its metrics after each cycle. We present the results for this run in the second column of Table 1. For example, for the case where constraints are disabled, the `w_19B` category has an average of 32 138 cycles per second. The speedup of Btor2MLIR compared to BtorSim is computed for each benchmark in a category. For each category, the mean of these values is presented in the third column of Table 1. When constraints are enabled, the running time varies heavily. This is expected when running simulation experiments since the approach does not exhaustively search through a circuits search space. The results show that Btor2MLIR is consistently faster regardless of what benchmarks we run.

6 Conclusion

In this paper we present semantics for existing and new Btor2 operations. We use the new operations to develop an algorithm for converting persistent arrays to transient arrays. This conversion provides a fast method for simulating formal designs for hardware circuits as demonstrated by our results. The implementation of the algorithm has been incorporated into Btor2MLIR and can be used to simulate formal designs of circuits represented in Btor2. In the future, we plan to extend this work with a case study that evaluates the application of testing and simulation technologies such as LibFuzzer and Klee, as well as model checking tools such as SeaHorn. Furthermore, it is interesting to explore the simulation of other formal designs with the goal of improving verification pipelines that have exhaustive search by design.

References

[1] Btor2MLIR (github). `https://github.com/jetafese/btor2mlir`.

[2] BtorSim (github). `https://github.com/Boolector/btor2tools/tree/master/src/btorsim`.

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[4] Bob Bentley. Validating the intel pentium 4 microprocessor. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, page 244–248, New York, NY, USA, 2001. Association for Computing Machinery.

[5] Armin Biere, Tom van Dijk, and Keijo Heljanko. Hardware model checking competition 2017. In *2017 Formal Methods in Computer Aided Design (FMCAD)*, pages 9–9, 2017.

[6] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[7] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2018.

[8] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The SeaHorn Verification Framework. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 343–361, Cham, 2015. Springer International Publishing.

[9] Arie Gurfinkel and Jorge A. Navas. Abstract interpretation of LLVM with a region-based memory model. In Roderick Bloem, Rayna Dimitrova, Chuchu Fan, and Natasha Sharygina, editors, *Software Verification - 13th International Conference, VSTTE 2021, New Haven, CT, USA, October 18-19, 2021, and 14th International Workshop, NSV 2021, Los Angeles, CA, USA, July 18-19, 2021, Revised Selected Papers*, volume 13124 of *Lecture Notes in Computer Science*, pages 122–144. Springer, 2021.

[10] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: A Compiler Infrastructure for the End of Moore's Law, 2020.

[11] Yuan Lu and Weimin Li. A semi-formal verification methodology. In *ASICON 2001. 2001 4th International Conference on ASIC Proceedings (Cat. No.01TH8549)*, pages 33–37, 2001.

[12] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. Btor2 , BtorMC and Boolector 3.0. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 587–595, Cham, 2018. Springer International Publishing.

[13] Siddharth Priya, Xiang Zhou, Yusen Su, Yakir Vizel, Yuyan Bao, and Arie Gurfinkel. Bounded Model Checking for LLVM. In *Formal Methods in Computer Aided Design, FMCAD 2022*, page 214, 2022.

[14] Silvio Ranise, Cesare Tinelli, and Clark Barrett. SMT-LIB The Satisfiability Modulo Theories Library (BitVectors). `https://smtlib.cs.uiowa.edu/theories-FixedSizeBitVectors.shtml`, 2017.

[15] Kosta Serebryany. Continuous Fuzzing with libFuzzer and AddressSanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*, pages 157–157, 2016.

[16] Joseph Tafese, Isabel Garcia-Contreras, and Arie Gurfinkel. Btor2MLIR: A Format and Toolchain for Hardware Verification. In *Formal Methods in Computer Aided Design, FMCAD 2023*, page 332, 2023.

[17] Cesare Tinelli. SMT-LIB The Satisfiability Modulo Theories Library (ArrayEx). `https://smtlib.cs.uiowa.edu/theories-ArraysEx.shtml`, 2017.

[18] Daylen Torres, Joaquin Cortez, and R. González. Semi-formal specifications and formal verification improving the digital design: Some statistics. *Journal of applied research and technology*, 7:15–40, 04 2009.