Practical Model Checking on FPGAs

SHENGHSUN CHO, Stony Brook University
MRUNAL PATEL, University of California, Los Angeles
MICHAEL FERDMAN and PETER MILDER, Stony Brook University

Software verification is an important stage of the software development process, particularly for mission-critical systems. As the traditional methodology of using unit tests falls short of verifying complex software, developers are increasingly relying on formal verification methods, such as explicit state model checking, to automatically verify that the software functions properly. However, due to the ever-increasing complexity of software designs, model checking cannot be performed in a reasonable amount of time when running on general-purpose cores, leading to the exploration of hardware-accelerated model checking. FPGAs have been demonstrated to be promising verification accelerators, exhibiting nearly three orders of magnitude speedup over software. Unfortunately, the "FPGA programmability wall," particularly the long synthesis and place-and-route times, block the general adoption of FPGAs for model checking.

To address this problem, we designed a runtime-programmable pipeline specifically for model checkers on FPGAs to minimize the "preparation time" before a model can be checked. Our design of the successor state generator and the state validator modules enables FPGA-acceleration of model checking without incurring the time-consuming FPGA implementation stages, reducing the preparation time before checking a model from hours to less than a minute, while incurring only a 26% execution time overhead compared to model-specific implementations.

CCS Concepts: • **Computer systems organization** → **Reconfigurable computing**; Very long instruction word; • **Theory of computation** → Program verification;

Additional Key Words and Phrases: Accelerators, overlay architecture

ACM Reference format:
Shenghsun Cho, Mrunal Patel, Michael Ferdman, and Peter Milder. 2021. Practical Model Checking on FPGAs. *ACM Trans. Reconfigurable Technol. Syst.* 14, 2, Article 8 (July 2021), 18 pages.
https://doi.org/10.1145/3448272

1 INTRODUCTION

The complexity of software systems has been growing for decades with no sign of slowing down. It has become challenging to verify and test systems because it is difficult, if not impossible, for the traditional unit-test methodology to yield full coverage of large, complex, and multi-threaded

Shenghsun Cho and Mrunal Patel contributed equally to this research.

Authors' addresses: S. Cho and M. Ferdman, 334 New Computer Science, Stony Brook University, Stony Brook, NY 11794-2424; emails: {shencho, mferdman}@cs.stonybrook.edu; P. Milder, 231 Light Engineering, Stony Brook University, Stony Brook, NY 11794-2350; email: peter.milder@stonybrook.edu; M. Patel, 468 Engineering VI, University of California, Los Angeles, Los Angeles, CA 90095-1596; email: mkpatel@cs.ucla.edu.

1936-7406/2021/07-ART8 $15.00
https://doi.org/10.1145/3448272

software. Software developers are increasingly turning to formal verification methods, such as explicit-state model checking, to test and check all states that a given software can reach. Explicit-state model checkers automatically generate the state transition graph of the software and check all reachable states exhaustively, making sure no violating state (i.e., no assertion failure) is reachable. Verification is especially important for mission-critical systems, including systems such as anti-lock braking systems in automobiles, fly-by-wire aircraft, and shut-down systems at nuclear power plants [16].

Unfortunately, the model checkers themselves are facing performance challenges due to the massive number of reachable states that they must explore [11]. Moreover, general-purpose CPU cores that run model checkers do not efficiently handle the computationally heavy model checking tasks, such as successor state generation and hashing, leading to extremely long execution times.

The poor performance of model checkers on general-purpose cores has led to the exploration of hardware-accelerated model checking. FPGAs have demonstrated impressive performance on model checking because of their flexibility, high-degree of parallelism, and a rich set of on-chip resources such as Block RAMs, which together can be used for building model-specific pipelines that achieve high throughput [5, 8]. However, when demonstrating hundreds-times speedup over model checking software, these FPGA implementations do not account for the "FPGA Programmability Wall." In particular, long FPGA compilation times (i.e., synthesis and place-and-route) can take hours before an FPGA-based model checker can start execution. Changes to the model being evaluated must go through this time-consuming process, diminishing the benefits of using FPGAs and making other accelerators, such as GPUs, more attractive [1–4, 6, 7], despite being an order of magnitude slower than FPGAs on this task.

We observe that the pipelines for the successor state generator and the state validator, the two components that change between models, can be executed on a simple programmable datapath without sacrificing the overall performance and efficiency of an FPGA model checker. A programmable pipeline would allow FPGA model checkers to be more flexible and applicable to a wider array of models.

In this work, we design an efficient instruction-driven runtime-programmable pipeline for model checkers. This pipeline replaces the model-specific pipeline found in prior works that hardwire the target model in FPGA logic. The result is a model checker on FPGAs that achieves high throughput model checking without requiring synthesis and place-and-route on every model change.

Using our model checking platform, we demonstrate the ability of our programmable pipeline to execute models from the **BEnchmarks for Explicit Model Checkers (BEEM)** [15]. Experimental results show that our programmable pipeline can reduce the "preparation time" from several hours, required by the model checkers with model-specific pipelines, to less than a minute, while incurring an average execution time overhead of only 26%.

2 BACKGROUND AND MOTIVATION

2.1 Explicit State Model Checking

Model checking is a formal verification methodology that aims to verify the correctness of software by confirming that all of its reachable states meet the safety properties (no state violates the specification) and the software as a whole meets the liveness properties (eventually reaches a desired state). In this article, we focus on checking the safety properties using one of the model checking variants called explicit state model checking, which maintains complete state information, such as software variables and program counters, in state bit-vectors.

Explicit state model checking confirms safety proprieties by generating the state transition graph on the fly and traversing and checking each of the states represented by the state vectors.

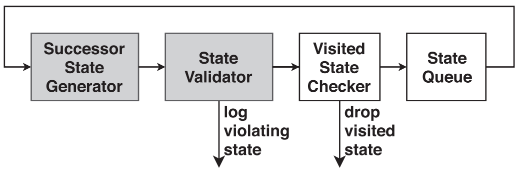


Fig. 1. Explicit state model checking processing flow. Gray boxes are model-dependent.

Figure 1 shows the flow of explicit state model checking. The two gray blocks, *Successor State Generator* and *State Validator*, are model-specific. Based on the software model, the successor state generator takes a state as input and generates all of that state's possible successor states as output. The successor states dictated by the model include not only those states arising from software control flow and varying user inputs, but also from system-level effects such as thread scheduling in multi-threaded software. The newly generated states are passed to the model-specific state validator to check if any of them violate the specification. If any violating states are discovered, then the model checker logs them for further investigation. The states are then passed to the *Visited State Checker*, which uses a hash table to check if the new states have been visited previously by the model checker. States that were previously visited are dropped to avoid an infinite loop in the model checker, while previously unseen states are placed into a *State Queue*, to be consumed by the successor state generator. The model checking loop continues until the state queue becomes empty.

2.2 State Space Exploration

We use an example software model written in *Promela*, the PROcess MEta LAnguage [9], to further explain how explicit state model checkers explore the model state space. Promela is the modeling language used by SPIN [10], a widely adopted explicit state model checker, to describe concurrent systems such as multi-threaded software that communicate through global variables or message-passing. Although traditionally Promela models were created by hand, to ease the process of creating Promela models and avoid potential discrepancies between model and software implementation, prior work has proposed techniques for automatically generating Promela models from software source code [12, 13].

Listing 1 shows the Promela example model for a banking application. Two customers (processes) are concurrently accessing the same account. The two processes, represented by their own *PIDs*, can both read and modify the global variable *balance*, while each process has its own local variable *cash*, which is not accessible to the other process. In the start state *S*, each process may decide (a non-deterministic, random choice between the possibilities, represented as *ND*) to withdraw money from the account (go to *W*) or to do nothing (go to *end*). If a process decides to make a withdrawal, then it goes into the withdraw state *W*, where the process decreases the global balance and increases its cash on hand, and then proceeds to the *end* state. The example also defines the initial values of the balance and cash variables, and a safety property that the global balance should never be negative.

Figure 2 shows the structure of the state vector for the example model. This type of model checking is called "explicit-state," because the state vector contains all unabstracted information of the software state. A constant, *NP*, indicates how many processes run concurrently, which is two in our example. The ***SID*** (**S**tate **ID**entifier), represents the current state of a particular process in the state machine. The model checker will iterate over every combination of the process execution order and non-deterministic choices to create and check the transition graph of all reachable states, known as the state space.

```
byte balance=1;
active [2] proctype customer() {
 byte cash=0;
 S: if :: goto W;
 :: goto end;
 fi;
 W: if :: d_step { balance=balance-1;
 cash=cash+1; };
 goto end;
 fi
 end:
}
active proctype safety() {
 (balance<0) -> printf("error");
}
```

Listing 1. Simple Promela example.

Figure 3 shows the state space of this example with one state marked as "violating," because it violates the safety property (the balance becomes negative). The successor state generator described in Section 2.1 generates the state space on the fly by using the flow in Figure 1 and the state validator logs any discovered violating states.

2.3 FPGA-Accelerated Model Checking

As the complexity of software keeps growing, the reachable state space of today's software can easily contain hundreds of millions or billions of states. At this scale, software-based model checkers are bottlenecked by the general-purpose CPU due to compute-bound operations such as the successor state generation, state validation, and visited state checking. Each state must go through all these operations, which can each take several microseconds. Considering that models can have tens of billions of states, the model checking task can take days or even weeks to finish [5], which is unacceptably long for the software development flow.

Software- and hardware-based solutions have been proposed to overcome this problem. Software solutions focus on improving the algorithms for traversing the state spaces, in particular, using multicore and distributed systems. For example, Swarm verification [17] divides the state exploration into many small and independent ***verification tasks*** **(VTs)** by explicitly limiting each VT's memory footprint. By using different hash function seeds and traversal algorithms, each VT explores a tiny, but different, fraction of the entire state space. Because all VTs are independent of each other, Swarm verification can run many VTs in parallel within a cloud environment to realize an order of magnitude speedup [17]. However, the overall throughput of software model checking remains fundamentally limited by the performance of the general-purpose CPU cores, preventing Swarm from achieving higher throughput without drastically increasing the costs of using more servers in the cloud.

The insufficient performance of general-purpose cores on the compute-intensive model checking tasks has inspired the exploration of hardware-accelerated model checking. Many studies use GPUs to conduct model checking tasks because of their massive parallel computation capability. However, because the compute units and memory systems of GPUs are not designed for model checking, model checkers on GPUs achieve only moderate improvements over general-purpose cores. However, FPGAs have been adopted for model checking and shown to have promising performance because of their flexibility, high parallelism, and massive on-chip memory

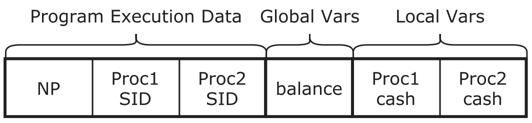


Fig. 2. State vector for the simple Promela example.

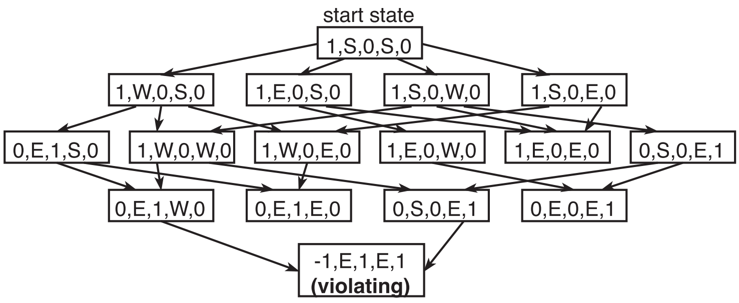


Fig. 3. State space generated from the Promela example.

bandwidth [5, 8]. FPGA model checkers exhibit up to three orders of magnitude speedup over software-based verification [5], making FPGA-accelerated model checking an attractive approach for overcoming the performance limitations of general-purpose cores.

2.4 Motivation for Programmable FPGA Swarm Verification

Although model checkers on FPGAs achieve promising performance for the "execution" part of the model checking tasks, the "preparation" time before the tasks can start running is a strong deterrent against using FPGAs in mainstream model checking. For example, FPGASwarm [5] advocates using model extractors to translate a model into synthesizable C or SystemC, then using HLS tools to generate RTL. This approach avoids the difficulty of writing software models manually in RTL and saves development time. However, the generated RTL must go through the time-consuming FPGA compilation (synthesis and place-and-route) process. Because FPGASwarm requires high FPGA resource utilization to maximize parallelism and throughput, the FPGA compilation can take more than an hour for a medium-sized FPGA, and many hours for large FPGAs, such as the ones available from the public cloud providers. Even worse, because of the high resource utilization target, place-and-route is unlikely to achieve timing closure on the first attempt, and the user may be forced to repeat the process multiple times.

When the long compilation time of the preparation process is considered, the usefulness of model checkers on FPGAs is drastically reduced, as the end-to-end turnaround time of checking a new model is no longer competitive. Modification of the model, which happens when the software developers make changes to the software being checked, requires generating new RTL. As a result, the amount of time saved by the accelerator is shifted from execution to the preparation phase.

To make FPGA-accelerated model checking practical, an FPGA model checker must be (1) fast in execution time and (2) fast in preparation time. For (1), we adopt the FPGASwarm verification methodology from Reference [5]. Then, as we will describe in Section 3, we achieve (2) by developing an instruction-driven runtime-programmable pipeline for the successor state generator and the state validator, which support checking different models without RTL changes.

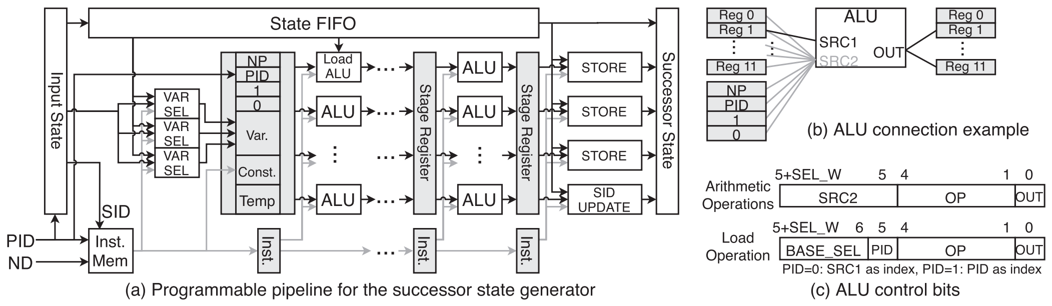


Fig. 4. Runtime-programmable pipeline for the successor state generator. Registers are marked in gray boxes.

3 ARCHITECTURE AND IMPLEMENTATION

The goal of our runtime-programmable model checker on FPGAs is to be compatible with a wide range of Promela models without RTL design changes, while maintaining high throughput. Recognizing that the successor state generation and state validation are the most significant components that change from model to model, we designed an interlock-free pipeline specifically for these operations. Our design also allows loading necessary parameters for different models, including initial state vector, number of processes, and the maximum value for ND, i.e., maximum number of non-deterministic choices to be made in any single state that need to be tested for the given model.

3.1 Programmable Pipeline Design Considerations

The programmable pipeline must have high throughput, handling one state per cycle to match the model-specific systems. To achieve such high throughput, each pipeline must have access to its own BRAMs containing a copy of the instructions, ensuring that there is no interference with the other pipelines running in parallel. Prior work [5] showed that the overall throughput of the Swarm-based FPGA model checker is bounded by the BRAM capacity. This bound is exacerbated by introducing the BRAM instruction storage needed for programmability. Therefore, a key consideration in the design is to minimize the footprint of the instruction memory.

3.2 Programmable Pipeline for Successor State Generator

Figure 4(a) shows the programmable pipeline of the successor state generator, which includes the instruction fetch, variable selection, multiple execute stages, a permutation stage, and a final store stage. Each stage takes a different slice of the instruction as its control bits.

Instruction Fetch. On each cycle, a state vector, along with a PID and an ND value, will be pushed into the pipeline to generate one successor state. The pipeline determines the address of the instruction to execute by concatenating the PID, ND, and SID values. To obtain the SID value, the PID is used to look up the corresponding process's SID in the state vector. The instruction is fetched from the pipeline's instruction memory (implemented using FPGA BRAM) and the instruction bits are sent down the pipeline. Because only part of the state vector is modified, but a full state vector must be produced, the entire input state vector is also passed through the pipeline.

Variable Selection and Constants. The first slice of the instruction bits is used to load values for the execute stages. Variables are selected from the state vector with the address encoded in the instruction, while constants are obtained from the instruction slice as immediate values. The number of variables and constants depends on the instruction format, which is dictated by the model and overall configuration of the pipeline. The pipeline must have enough variable selection units and constants to support all of the instructions for a given model. At a minimum, the pipeline must have one constant, the "next SID," corresponding to the possible next state of the model. For each selection unit, $\log_2 S$ bits of the instruction are used to select from among S variables in the state vector.

Pipeline Registers. The pipeline registers between the variable selection stage and the final store stage have a specific arrangement. For the first stage, there are M registers containing values selected from the state vector, where M is the number of value selection units, followed by N registers containing immediate values from the instruction. After the immediate values, there are several registers used for storing temporary values between the multiple execution stages. For each execution stage, if a register value is not modified by an ALU, then the value is passed onto the subsequent stage unaltered.

Execute Stages. Computation is performed by a series of execute stages, each having several parallel ALUs. The number of ALUs per stage and the number of stages in the pipeline should be large enough to accommodate any of the target models. In addition to the resource utilization of the ALUs, the number of ALUs also impacts the instruction size, because each ALU requires control bits from the instruction to dictate its operation.

Figure 4(b) shows an example of the connection between an ALU and the pipeline registers before and after it. Each ALU has two operands and one output. To reduce the connectivity complexity and the number of instruction control bits, the first operand of each ALU is fixed, and only the second operand is freely selected from the preceding stage registers and four read-only constants: NP, PID, value 1, and value 0. Each ALU output is restricted to be stored in one of two possible locations, which also reduces connectivity complexity without sacrificing the ability to efficiently map Promela models to instructions.

Because the indexed load operations are relatively infrequent in practical models, there is no need for all ALUs to support loads. We develop two ALU types: Normal ALUs and Load ALUs. Normal ALUs only perform arithmetic operations on the operands and output the result, while the Load ALUs can perform arithmetic operations and also load values from the state vector. The Load ALUs require connections to all values of the state vector, requiring significantly more FPGA resources compared to Normal ALUs. For this reason, we limit the number of Load ALUs to at most 1 per execute stage. The Load ALUs use a **base address register (BAR)** number and an offset as inputs. The final index into the state vector is calculated by adding the offset to the value read from the appropriate BAR. These BARs are specific to the model being processed and are loaded alongside the instructions for the model.

Figure 4(c) illustrates the format of the control bits for the ALU. There is a four-bit *OP* field that corresponds to 16 operations shown in Table 1. The first 15 operations are arithmetic operations, while the last one is the load operation available only in the Load ALUs. For arithmetic operations, the $\log_2 R$ bit *SRC2* field (marked as SEL_W in Figure 4(c)) is used to select the second operand from R intermediate pipeline register locations. For the load operation, a Load ALU uses the 3-bit *BASE_SEL* to select one base address register out of eight that point to predefined locations in the state vector as the base for the indexed addressing. The 1-bit *PID* field of the load operation is used to indicate to the ALU whether it should use the PID or the first operand as the address index. For all ALU operations, the result is stored into one of the two fixed registers in the following stage, based on the 1-bit *OUT* field.

Permutation Stage. The permutation stage sets up appropriate registers for the store unit by moving the calculated values from the execute stages to the correct location for the store unit to use. The instruction for this stage is a sequence of address groups, one for each store unit. Each address group consists of three addresses, each of size $\log_2 R$ bits, describing the location in the R intermediate pipeline registers. The permutation stage reads the specified intermediate pipeline

Table 1. ALU operations

OP	Output	OP	Output	OP	Output	OP	Output
0000	+	0100	>	1000	==	1100	!SRC1
0001	−	0101	<	1001	! =	1101	SRC2
0010	<<	0110	>=	1010	&	1110	!SRC2
0011	>>	0111	<=	1011	|	1111	LOAD

registers and provides them as inputs for the store units. This approach increases flexibility in terms of where results can be stored for the previous stages of the pipeline. The previous stages can use any available slot to store results and the permutation stage will rearrange the result appropriately. Without the permutation stage, a more complex approach, akin to a course-grained reconfigurable array, would be necessary to achieve similar results.

Store Stage and SID Update. The store stage comprises several store units to update variables in the state vector, forming the output successor state vector. Each store unit has three inputs from the previous stage: *condition*, *value*, and *index*. There are $\log_2 S$ store control bits to indicate the *base* location in the state vector with S variables. If the *condition* is non-zero, then the variable in the state vector with position *base* + *index* is replaced with *value*. A special SID updating unit updates the SID for the given PID with the “next SID” constant stored in the instruction, using the same *condition* as the first store unit.

3.3 Programmable Pipeline for State Validator

The ALUs in the state validator pipeline are arranged similarly to the successor state generator. However, the validator pipeline is simpler. Since all state vectors require the same validation checks, unlike the successor state generator, there is no need for separate instructions for each PID-ND-SID combination. As a result, the state validator pipeline does not need instruction storage or fetch logic. The state validator pipeline also does not need a store stage, as it never updates the state vector. The one instruction for validating states is loaded into the pipeline when a new model is programmed into the system. The final output of the validator pipeline is a single value, selected from a register in the last execution stage, which indicates whether or not the input state is a model violation. The validator pipeline does not consume BRAMs, as it has no instruction memory footprint and, therefore, does not affect the number of pipelines that can fit into a target FPGA.

3.4 Initialization and Parameterization

To be able to verify models with arbitrary initial state vectors, the design needs to load the initialization data in addition to the model’s instructions. Other parameters that are critical for processing and correctly verifying a model, namely, the number of processes and the maximum value of ND, are also loaded upon initialization. Specific addresses used to index into the state vector are loaded into the BARs upon initialization.

4 COMPILATION

To use the programmable pipeline for checking models, we need to convert the Promela models into instructions that the pipeline can execute. To validate our design and approach, we perform this conversion manually, but we identify the key steps that would allow this process to be automated using a compiler in the future. This section provides an overview of this process. First, we must analyze the Promela model to determine the number of variables and constants, as well as

```
byte pos[3];
byte step[3];

active proctype P_0() {
 byte j=0; byte k=0;
 NCS: if :: j = 1; goto wait; fi;
 CS: if :: pos[0] = 0; goto NCS; fi;
 wait: if :: d_step {j<3;pos[0] = j;} goto q2;
 :: j==3; goto CS; fi;
 q2: if :: d_step {step[j-1] = 0;k = 0;} goto q3; fi;
 q3: if :: d_step {k<3 && (k==0 || pos[k]<j);k = k+1;} goto q3;
 :: d_step {step[j-1]!=0 || k==3;j = j+1;} goto wait; fi;
}

active proctype P_1() {
 byte j=0; byte k=0;
 NCS: if :: j = 1; goto wait; fi;
 CS: if :: pos[1] = 0; goto NCS; fi;
 wait: if :: d_step {j<3;pos[1] = j;} goto q2;
 :: j==3; goto CS; fi;
 q2: if :: d_step {step[j-1] = 1;k = 0;} goto q3; fi;
 q3: if :: d_step {k<3 && (k==1 || pos[k]<j);k = k+1;} goto q3;
 :: d_step {step[j-1]!=1 || k==3;j = j+1;} goto wait; fi;
}

active proctype P_2() {
 byte j=0; byte k=0;
 NCS: if :: j = 1; goto wait; fi;
 CS: if :: pos[2] = 0; goto NCS; fi;
 wait: if :: d_step {j<3;pos[2] = j;} goto q2;
 :: j==3; goto CS; fi;
 q2: if :: d_step {step[j-1] = 2;k = 0;} goto q3; fi;
 q3: if :: d_step {k<3 && (k==2 || pos[k]<j);k = k+1;} goto q3;
 :: d_step {step[j-1]!=2 || k==3;j = j+1;} goto wait; fi;
}
```

Listing 2. Peterson.1 Promela model.

the values of the constants. Then, for each PID-ND-SID combination, we generate one instruction based on the corresponding Promela operations. After all statements are translated into instructions, the process ends by determining the minimum requirements that must be met by a pipeline to support checking this particular model. The following subsections demonstrate this process, converting a Promela model to instructions using the Peterson.1 Promela model as an example, shown in Listing 2.

4.1 Constants and Variables Analysis

The compilation process starts with the analysis of the Promela model to identify the NP, ND, and SID constants and the number of variables. The first constant to identify is NP, which is the number of processes in the model. A Promela process is denoted by the keyword `proctype`. Listing 2 shows there are three nearly identical processes in the Peterson.1 model. The next constant, ND, represents the maximum number of options in any one state in any process. An *option* in Promela is a series of statements in an `if` block that is randomly selected to be executed. Each line in Listing 2 that has `::` in it represents one option. For example, Listing 2's `wait` and `q3` states have two options each. Third, we assign a numerical identifier to each state, i.e., the SID. In Listing 2, the states are: `NCS`, `CS`, `wait`, `q2`, and, `q3` and their corresponding SIDs are: 0, 1, 2, 3, and, 4. Then, we must determine the overall size of the state vector. We begin by determining the space required to store the variables, which can easily be calculated based on the model's declarations. Peterson.1 uses six bytes in two global arrays, and each process has two local variables (a byte each), bringing the total size of the variables to 12 bytes. Each process also requires a slot to store its SID to keep track of its state. By adding an SID variable for each process into the state vector, we know that the state vector size of Peterson.1 is 15 bytes.

The last step is to identify numerical constants in the Promela description that are equal to (and thus can be replaced by) the NP and PID constants. In Listing 2, we can see that, in `wait` and `q3` of each process, the number `3` is equal to NP, so we can replace it with NP to avoid having to encode `3` as a constant in the instruction. Similarly, by analyzing the Promela code, we can see that in the `q2` state, `step[j-1]` is always set to the PID of each process. We can therefore replace `step[j-1]=0(,1,2)` in line 10(,21,32) of Listing 2 to `step[j-1]=PID` to avoid another constant in the instruction. The same analysis is also done for the `q3` state of each process, where `k==0(,1,2)` can be replaced with `k==PID`, and `step[j-1]!=0(,1,2)` can be replaced with `step[j-1]!=PID`. More generally, anywhere a numeric literal can be replaced by either the PID of the process or NP, it should be replaced. The replacement of constants with NP and PID reduces the length of the instruction, which is critical to save on-chip memory resources.

4.2 Instruction Generation

The process of generating instructions can be compared to that of a VLIW pipeline. Each stage of the pipeline processes a micro-op that is dictated by a section of a large instruction. One instruction contains all the micro-ops required for processing a particular PID-ND-SID combination. The micro-ops are stored sequentially in the instruction with the micro-ops of the first stage—i.e., variable selection—in the lowest-order bits, followed by the micro-ops of the execution stages and the permutation stage.

Each Promela option in a given state of a process becomes one instruction and maps to a specific PID-ND-SID combination. In fact, we use the concatenation of the PID, ND, and SID to address the instructions. The PID selects the process, the SID selects the state, and the ND selects one of the several potential options in the if block. The PID-ND-SID combinations that do not have an associated Promela option, by default store the NOP instructions. To begin converting a Promela option to instructions, first, we scan through the statements in the option to determine the number of variables, constants (other than the four read-only constants, described in Section 3.2), loads, and stores (other than updating SID). Referring back to Listing 2 as an example, the option at line 6 (PID=0,ND=0,SID=0)] only requires one store to variable `j`, whereas the option at line 11 (PID=0, ND=0, SID=4) requires two variables to be selected in the beginning (`k` and `j`), one load during the execution (`pos[k]`) and one store to the variable `k`.

Next, we create a dataflow graph and use it to determine the minimal depth of the execution stages required for a particular PID-ND-SID combination, e.g., one stage for line 9 (PID=0,ND=1,SID=2), and four stages for line 11 (PID=0,ND=0,SID=4). Then, we proceed by scheduling all operations into the execution stages with minimized width (i.e., the number of ALUs per execution stage), by first trying width equal to 1. If the attempt fails, then we increase the width to 2 and repeat the scheduling process, until all operations can fit into the execution stages, e.g., one for line 9 and two for line 11. Based on the result of the scheduling, micro-operations for each

Table 2. Transformation Conditions

Transformation	Conditions	Original src	Transformed src

Replacing % (mod)	1. Variable is used to index an array.
	2. Divisor is the same as the number of elements in the array.
	3. Dividend is only updated in increments OR if the dividend is assigned a variable, that variable is bounded in a range that is ≤ the number of elements in the array.
		`byte array[3]`
		`...`
		`x = array[my_place]`
		`...`
		`my_place=my_place%3`
		`...`
		`my_place++`
			`byte array[3]`
			`...`
			`x = array[my_place]`
			`...`
			`my_place=(my_place==3)`
			` ? my_place-3`
			` : my_place`
			`...`
			`my_place++`

Replacing * (mul)	1. Multiplication has a Boolean as an operand. 2. Multiplication is part of a bitwise OR operation.	`x=(x-1)|(x==255)*255`	`x=(x==255) ? x : x-1`
Replacing Numbers	1. Numeric literal matches the PID or NP.	(Proc0:) `x=0,y=3` (Proc1:) `x=1,y=3` (Proc2:) `x=2,y=3`	(Proc0:) `x=PID,y=NP` (Proc1:) `x=PID,y=NP` (Proc2:) `x=PID,y=NP`

of the ALUs in the execution stages can then be generated. It should be noted that the flow of data between pipeline execute stages is restricted by the connections of the ALUs, which have one fixed operand and two fixed destination registers (Section 3.2). Finally, the micro-operations for the permutation stage are needed to put the conditions and values into the correct inputs of the store units. The result of this process is the conversion of one Promela option into an instruction that can be executed by the pipeline.

Code Transformations. When analyzing the Promela models in the BEEM benchmark set, we observed the use of two complex operations: modulo and multiplication. Although it would be possible to include support for these operations within our programmable pipeline, we observed that they were used only for simple array bound checking. Therefore, we are able to transform these operations into a sequence of already-supported micro-ops. This allows our pipeline to perform the required operations without the unnecessary expense and complexity of extending the ALU.

Table 2 shows the conditions under which the transformations are applicable along with example source transformations. First, if the modulus operator is used as a rudimentary mechanism to perform bounds checking, then the conditions will hold and we can replace it with a conditional assignment. Second, if one of the two multiplication operands is a Boolean, then the multiplication operation is simply a conditional assignment and can be replaced. Finally, instead of explicitly encoding constants into instructions, we can use the special constants PID or NP when applicable.

4.3 Determining Minimal Pipeline Requirements

After generating the instructions for all statements in the Promela model, we have all the information to determine the minimal configuration requirement of the pipeline to support execution of this model. First, the size of the state vector is determined in Section 4.1. Other parameters, including the number of variable selectors, variables, constants, load ALUs, and store units, as well as the depth and width of the execution stages, are determined based on the maximum dimensions required across the set of instructions generated (i.e., the parameters are found such that all instructions can be executed on the pipeline).

Table 3. Benchmarks from the BEEM Database

Benchmark	Processes	State Vec. Size	Var. Sel. Units	Constants	ALUs	Store Units	Inst. Size
Anderson.8	7	22 Bytes	2	2	2x3	3	131 bits
Bakery.8	5	25 Bytes	3	2	2x5	3	172 bits
Lamport.8	5	17 Bytes	2	2	2x3	2	114 bits
Leader_Filters.7	6	30 Bytes	2	1	2x4	1	107 bits
Mcs.6	5	21 Bytes	1	2	1x1	2	64 bits
Peterson.7	5	25 Bytes	3	1	2x5	2	147 bits
Superset	7	30 Bytes	3	2	2x5	3	172 bits

5 EVALUATION

We evaluated our programmable pipeline by embedding it into FPGASwarm, the state-of-the-art FPGA model checker [5]. Our evaluation studies the overhead of the programmable pipeline in terms of FPGA resource utilization and performance when compared to model-specific pipelines, and presents a study of the FPGASwarm design space that was previously prohibitive due to the per-model preparation overhead of the non-programmable FPGASwarm.

5.1 Implementation

We implement the programmable successor state generator pipeline using SystemC and Xilinx Vivado HLS similar to FPGASwarm, targeting medium-sized FPGAs (Virtex-7). The programmable pipeline replaces the model-specific successor state generator pipeline of the FPGASwarm design. We modified FPGASwarm [5] to eliminate the state validation pipelines as the benchmarks in the BEEM database do not include violation states. To compare the experiments directly with FPGASwarm, we use a similar configuration. Like FPGASwarm, we use a 4K-deep state queue and 64 KB visited state storage. However, due to the increased complexity of the pipeline, the 200 MHz clock frequency we use is 50 MHz slower than the frequency of the FPGASwarm system.

5.2 Methodology

The BEEM database comprises a large variety of model checking problems. Most (all but three) models in the BEEM database do not require 4-byte integer support. Additionally, the current architecture of the pipeline does not support certain Promela language features such as channels. Adding support for these feature remains an avenue for further exploration. Beyond the limitations imposed by the above architectural constraints, we limited the set of benchmarks we present to those commonly used in the prior model checker work due to a lack of compiler automation, as it limited the number of models we could readily translate by hand. Automating the translation process is part of our future work.

Table 3 shows the characteristics of the models we use in our evaluation, as well as the minimum configuration for our programmable pipeline to support the models. The characteristics and configurations shown in the table include (from left to right) the number of processes, the size of the state vector in bytes, the number of variable selector units, the ALU configuration (width and depth), the number of store units, and the width of the instruction in bits. The Superset row shows a configuration that can support all benchmarks.

Table 4. Resource Utilization and Number of VT Engines on Virtex-7 FPGA

	Programmable Pipeline				Model-Specific Pipeline				
Benchmark	**LUTs**	**FFs**	**BRAMs**	**Num. Cores**	**LUTs**	**FFs**	**BRAMs**	**Num. Cores**	**Prep. Time (m)**
Anderson.8	267,077 (62%)	440,494 (51%)	1,220 (83%)	30	216,662 (50%)	473,930 (55%)	1,280 (87%)	34	116
Bakery.8	273,876 (63%)	431,966 (50%)	1,215 (83%)	25	206,532 (48%)	312,356 (36%)	1,250 (85%)	31	119
Lamport.8	260,582 (60%)	424,673 (49%)	1,171 (80%)	33	211,371 (49%)	369,786 (43%)	1,222 (83%)	37	132
Leader_Filters.7	237,981 (55%)	378,117 (44%)	1,190 (81%)	25	175,010 (40%)	285,586 (33%)	1,260 (86%)	28	129
Mcs.6	234,048 (54%)	402,307 (46%)	1,181 (80%)	31	210,795 (49%)	377,189 (44%)	1,246 (85%)	34	160
Peterson.7	266,543 (62%)	411,020 (47%)	1,172 (80%)	27	191,031 (44%)	333,485 (38%)	1,250 (85%)	31	132
Superset	267,743 (62%)	420,460 (49%)	1,169 (80%)	24	-	-	-	-	-

To evaluate the performance of our design, we directly compare with FPGASwarm [5] and report the relative differences. The FPGASwarm approach fills the FPGA with VT engines that each have a model-specific pipeline for their successor state generator. We use the same organization for our designs, but replace the model-specific pipelines with programmable pipelines. Independent VT engines run independent VTs, making the total execution time for a model proportional to the number of VT engines that fit onto an FPGA. Specifically, the execution time is the total number of VTs required to check the model multiplied by the average runtime per VT, and divided by the number of VT engines that fit onto the FPGA. By replicating the same model checker infrastructure as the prior work (the same depth of the state queue and size of the visited state storage), we ensure that the total number of VTs of the model-specific pipeline and our programmable pipeline are identical. Furthermore, because both systems have a one-state-per-cycle pipeline throughput and the same FPGA chip and clock frequency, the difference in the performance of these two systems comes from the number of VT engines that fit on the FPGA, which in turn is dictated by the BRAM usage of the corresponding VT engines.

5.3 Overhead of Programmability

The goal of using programmable pipelines in the model checker is to ensure that the VT engines can accommodate different models with different sizes and configurations without RTL re-compilation. However, supporting programmability requires over-provisioning the VT engines and using more than the bare minimum of FPGA resources needed by the model-specific pipelines. Higher resource usage of the VT engines with programmable pipelines results in a reduction in the number of VT engines that can fit onto the FPGA. We call this reduction in the number of VT engines the *overhead of programmability*, which directly translates to a reduction in performance as described in Section 5.2.

Table 4 presents the post-P&R FPGA resource utilization for our benchmarks and the Superset design, using our programmable pipeline on a Xilinx Virtex-7 XC7V690T FFG1761-3 FPGA. The BRAM utilization is the highest among the FPGA resources, which corroborates the prior work [5]. As expected, the logic (LUTs and FFs) utilization is higher for the programmable pipeline. However, despite the increased resource usage, logic remains under-utilized and BRAM utilization remains the primary determinant of the number of VT engines.

Figure 5 presents the relative performance of the model checker with the programmable pipeline compared to the model-specific FPGASwarm system. The gray bars show the normalized performance when using the runtime-configurable Superset pipeline that supports all benchmarks. In the worst case, the programmable model checker is 37% slower than the model checker with model-specific pipelines, with an average of 26% slowdown across all benchmarks. Lamport.8 has the

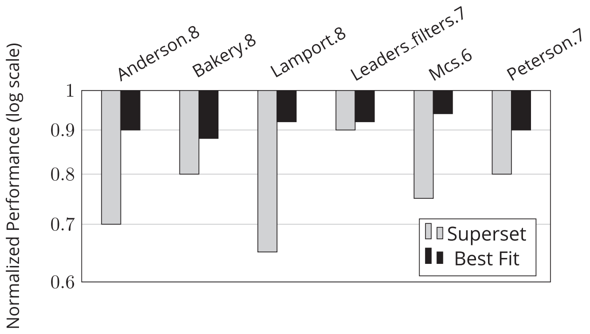


Fig. 5. Overhead of programmability.

worst performance, because it has the smallest state vector. The small state vector allows for a larger number of VT engines with model-specific pipelines to fit on the chip, resulting in higher performance. However, Leader_filter.7 has the same state vector size as the Superset configuration, resulting in only three VT engines difference (14%) compared to the model-specific FPGASwarm pipelines, with the difference arising solely due to the instruction memory BRAM usage of the programmable pipelines.

Despite the runtime programmability resulting in up to 37% slowdown, the overall benefit of using FPGAs for model checking (yielding multiple orders of magnitude performance improvement relative to CPUs and GPUs) remains significant. Critically, despite the slowdown, the programmable pipeline is a major advance for FPGA model checking, as it allows using the system without paying the multi-hour preparation time cost of the model-specific pipeline.

5.4 Optimization with Best Fit Configurations

The efficiency of the programmable pipeline can be improved by using a configuration of the pipeline that is tailored more specifically for the model being checked. This idea can be used to minimize the performance gap between the programmable and model-specific pipelines. After we translate a Promela model into programmable-pipeline instructions, we know the minimum parameters the pipeline needs to run the model (i.e., the size of the state vector, number of variables and constants, depth and width of the pipeline, and number of store units). We can create a library of pre-compiled programmable model checkers for models with different parameters and configuration requirements. To prevent the inefficiencies of using a larger configuration to run a smaller model, we can select the best-fitting model checker bitfile from the library to minimize the amount of resources that are wasted. Such a library will provide the ability for different groups of similarly sized models to be tested quickly, as switching configurations is as simple as installing a different bitstream.

The black bars in Figure 5 show the normalized performance when choosing the best-fitting checker for each of our benchmarks. Compared to the model-specific pipeline, the best-fit programmable pipeline only incurs the additional cost of instruction memory, as the amount of BRAM used for state queues and visited state storage is the same for both. The models that have the smallest instructions will have the best performance (Mcs.6) and larger instructions have worse performance (Bakery.8).

The reduced overhead of the best-fit programmable pipelines from a library compared to the Superset pipeline is primarily attributed to the reduced state vector size (i.e., for each model, the programmable pipeline and model-specific pipeline have the same state vector size) and partially to the reduced instruction width (i.e., best-fitting pipeline dimensions). The Superset pipeline incurs a performance loss when the state vector size is unnecessarily large for the given model, an effect that is emphasized in the case of Lamport.8.

5.5 Preparation Time Comparison

Table 4 also shows the preparation time in minutes, including using HLS to generate RTL code, FPGA synthesis, and place-and-route for the model checker with model-specific pipelines on the Virtex-7. The times were obtained on a modern server with two Xeon E5-2670v3 CPUs running at 2.3 GHz and 256 GB DDR4 RAM. The results clearly demonstrate why our programmable pipeline is desired and why the model-specific approach is not practical. While the model-specific designs require 2 to 3 hours of preparation, our programmable FPGA model checker only needs to download the bitstream and under a second to transfer the programmable pipeline instructions to the FPGA. For larger FPGAs, such as the UltraScale+, the preparation time can reach 10 hours, making the sub-one-minute preparation time of the programmable pipeline design even more attractive.

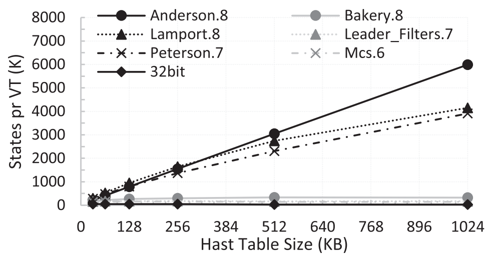
5.6 Scalability to Larger FPGAs

Prior work [14] observed that the number of VT engines that can fit into a Xilinx Virtex-7 FPGA is limited by the BRAM size, and therefore projected the number of VT engines that would fit on larger FPGAs based on this fact. However, modern FPGA products exhibit a different balance between logic resources and on-chip memory resources. The total on-chip memory capacity of FPGAs in the Xilinx UltraScale and UltraScale+ families is greatly increased by the introduction of the high-density Ultra RAM blocks in addition to Block RAM, but the density of LUTs remains the same. For example, relative to the Virtex-7 XC7V690T FPGA used in this work, the VU37P FPGA has six times higher on-chip memory capacity, but only three times more LUTs. This indicates that the number of VT engines will be limited not by on-chip memory, but in fact by the available LUTs on the VU37P. We quantify this difference based on the resource utilization of our VT engines. If we scale our Superset design to target 80% LUT utilization on the VU37P, then we find that the FPGA could hold 96 VT engines, which is lower than the estimated 178 that the prior work estimated based on the on-chip memory capacity.

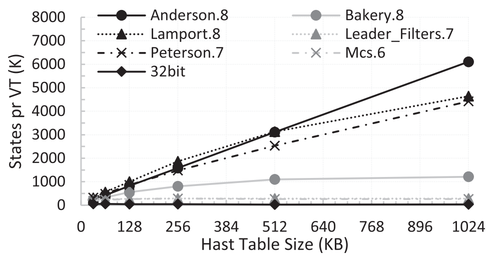
5.7 Model Sensitivity to Queue Depth and Hash Table Size

The programmable pipeline we designed allows us to explore the characteristics of different models, without going through the time-consuming FPGA preparation for each model. Programmability not only greatly increases the usability of the system, but it also creates opportunities for studying the FPGASwarm design space, which was previously intractable due to the need to perform place-and-route of each candidate design and model. We utilize this capability to study how the two key design parameters (state queue depth and hash table size) affect various benchmark models, with the goal of understanding the behavior of different models on an FPGASwarm model checker. We include the BEEM benchmarks used described above, as well as the 32-bit random integer benchmark from prior work [5].

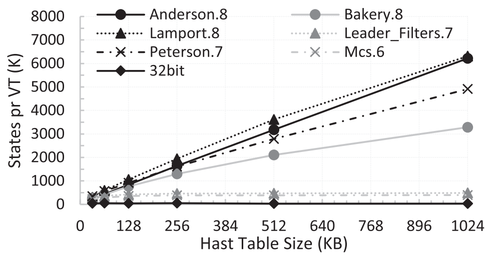
The state queue depth and the hash table size are interesting to study, because both affect the states each VT can explore, which affects the overall performance of swarm verification. The depth of the state queue determines the number of new states that can be queued for a VT to process. A full queue means the visited state checker must drop states even if they were not visited before, and an empty queue signifies the termination of a VT. The size of the hash table determines the maximum number of new states a VT can process, because it is used to track the visited states and terminate an exploration path if a state repeats (or even if a state is previously unseen, but hashes to the same location as a previously visited state).



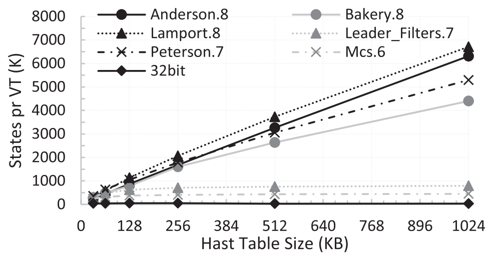
(a) 2K State Queue Entries



(b) 4K State Queue Entries



(c) 8K State Queue Entries



(d) 16K State Queue Entries

Fig. 6. Average number of states discovered per VT.

In this experiment, we create 24 configurations of the programmable pipeline by combining four choices of queue depth (2K, 4K, 8K, and 16K) with six choices of hash table size (from 32 KB to 1 MB). For each design, we run 100 VTs of each model on each configuration. For each VT, we record the number of states that were put into the state queue and average the result across all 100 VTs. This provides a metric that represents the amount of state exploration performed. Figure 6 illustrates the results. Each of the four graphs indicates a different state queue depth; within each graph, the x-axis indicates the hash table size. The y-axis of each graph shows the average number of states placed into the state queue.

We observe that some models, such as Anderson, Lamport, and Peterson, are able to benefit from larger hash tables, but they are not significantly affected by the queue sizes. In these models, larger hash tables enable a VT to explore wider and deeper into the state space; because there is a considerable amount of duplicated successor states from the same parent state, the visited state checker can filter them out without filling up the state queue, reducing the rate of dropped states that are not explored by the current VT.

The remaining models illustrate another behavior, where the state space expands quickly with many unique states, causing the state queue and hash table to fill up at the same rate. Because the state queue has fewer entries than the number of bits in the hash table, the state queue is quickly saturated, causing the visited state checker to begin dropping states after marking them as visited in the hash table, resulting in fewer unseen states being queued.

Notably, we find that an extreme case is observed in the 32-bit random integer model used to demonstrate the original FPGASwarm work. Although FPGASwarm was shown to be effective at exploring the design state space much faster than a software model checker, here, we observe that the nature of the 32-bit random integer benchmark exerts more pressure on the VT engine and frequently experiences corner-case behaviors, such as dropping unexplored states, compared to the BEEM benchmarks. We observe that the 32-bit random integer state space expands extremely fast at the beginning of exploration—each state produces 32 successors—quickly filling up the state queue. However, the visited state checker continues filling the hash table at the same rate as it keeps seeing more previously unseen states, forcing 31 out of 32 of these states to be dropped. When the hash table reaches capacity and terminates the VT, preventing further state space exploration, relatively few new states were put into the state queue. Enlarging the state queue from 2K to 16K does not help much, because the larger queue depth is still too small to successfully absorb the growth of the 32-bit random integer's state space. In a way, this result shows that swarm verification actually struggles when exploring 32-bit random integer model, even though it is able to do it effectively because of the very high performance (high state verification throughput) of the FPGA implementation.

Other models, including Bakery, Leader Filters, and Mcs, do not behave as poorly as the 32-bit random integer model at the start of the exploration. For these models, increasing queue depth helps to put more states into the state queue, which can be seen clearly with Bakery and Leader Filters. Last, we can see the benefit of larger queue and hash table for Mcs is not significant. This is because the total number of unique states in this model is small, so even with medium-sized storage, a single VT is already able to visit almost all unique states, without leveraging the benefits of larger storage.

6 RELATED WORK

Researchers have proposed using FPGAs to accelerate the model checking process. Fuess et al. [8] built a Murphi model checker on an FPGA, showing 200× speedup over a software implementation using general-purpose cores for a relatively small model. Cho et al. [5] implemented the Swarm verification methodology using FPGAs, showing a 900× speedup over a software Swarm implementation on a synthetic model with 4B states. Although both of these FPGA model checkers report impressive performance, they suffer from the long preparation time that is not included in their runtime considerations, and thus they do not support rapidly changing the models being checked. Our instruction-driven runtime-programmable pipeline for FPGA model checkers eliminates the preparation process for switching models while maintaining similar speedups relative to software.

Although the speedup on GPGPU model checkers relative to general-purpose CPUs are less than one order of magnitude, they remain a popular accelerator choice for model checking [1–4, 6, 7]. This is primarily attributable to the relatively easy-to-use programming model and fast compilation time of GPGPUs compared to FPGAs. Our work bridges this gap for FPGAs, enabling both rapid preparation and high model checker throughput, yielding the most practical solution to hardware-accelerated model checking to date.

7 CONCLUSIONS

Software verification using explicit state model checking with general-purpose cores is extremely time-consuming due to the ever-increasing complexity of software designs. Although model checkers on FPGAs have demonstrated significantly higher performance, the long FPGA preparation time required to set up the model checking operation hampers the general adoption of FPGA-accelerated model checking.

In this work, we presented instruction-driven runtime-programmable pipelines for explicit state model checking on FPGAs. Using our programmable pipelines in place of a model-specific successor state generator and state validator, model checkers on FPGAs can be made programmable and eliminate the long preparation time. Our results indicate that model checkers with our programmable pipelines reduce the model preparation time from hours to less than a minute, with only a small cost in runtime performance, making FPGAs practical for hardware-accelerated model checking.

REFERENCES

[1] J. Barnat, P. Bauch, L. Brim, and M. Ceska. 2010. Employing multiple CUDA devices to accelerate LTL model checking. In *Proceedings of the IEEE 16th International Conference on Parallel and Distributed Systems*. 259–266. DOI: https://doi.org/10.1109/ICPADS.2010.82

[2] Jiri Barnat, Lubos Brim, and Milan Ceska. 2009. DiVinE-CUDA – A tool for GPU accelerated LTL model checking. In *Proceedings of the 8th International Workshop on Parallel and Distributed Methods in Verification (PDMC'09)*. 107–111. DOI: https://doi.org/10.4204/EPTCS.14.8

[3] Ezio Bartocci, Richard DeFrancisco, and Scott A. Smolka. 2014. Towards a GPGPU-parallel SPIN model checker. In *Proceedings of the International SPIN Symposium on Model Checking of Software (SPIN'14)*. ACM, New York, NY, 87–96. DOI: https://doi.org/10.1145/2632362.2632379

[4] Dragan Bošnački, Stefan Edelkamp, Damian Sulewski, and Anton Wijs. 2011. Parallel probabilistic model checking on general purpose graphics processors. *Int. J. Softw. Tools Technol. Transf.* 13, 1 (01 Jan. 2011), 21–35. DOI: https://doi.org/10.1007/s10009-010-0176-4

[5] S. Cho, M. Ferdman, and P. Milder. 2018. FPGASwarm: High throughput model checking on FPGAs. In *28th International Conference on Field Programmable Logic and Applications (FPL'18)*. 435–442. DOI: https://doi.org/10.1109/FPL.2018.00080

[6] Stefan Edelkamp and Damian Sulewski. 2010. Efficient explicit-state model checking on general purpose graphics processors. In *Model Checking Software*, Jaco van de Pol and Michael Weber (Eds.). Springer Berlin, 106–123.

[7] Tony Field, Peter G. Harrison, Jeremy Bradley, and Uli Harder (Eds.). 2002. *PRISM: Probabilistic Symbolic Model Checker*. Springer Berlin. DOI: https://doi.org/10.1007/3-540-46029-2_13

[8] M. E. Fuess, M. Leeser, and T. Leonard. 2008. An FPGA implementation of explicit-state model checking. In *16th International Symposium on Field-programmable Custom Computing Machines*. 119–126. DOI: https://doi.org/10.1109/FCCM.2008.36

[9] Gerard Holzmann. 2011. *The SPIN Model Checker: Primer and Reference Manual* (1st ed.). Addison-Wesley Professional.

[10] G. J. Holzmann. 1997. The model checker SPIN. *IEEE Trans. Softw. Eng.* 23, 5 (May 1997), 279–295. DOI: https://doi.org/10.1109/32.588521

[11] G. J. Holzmann, R. Joshi, and A. Groce. 2011. Swarm verification techniques. *IEEE Trans. Softw. Eng.* 37, 6 (Nov. 2011), 845–857. DOI: https://doi.org/10.1109/TSE.2010.110

[12] K. Jiang. 2009. Model checking C programs by translating C to Promela.

[13] Modex 2018. Modex - Model Extraction. Retrieved from http://spinroot.com/modex/.

[14] M. Patel, S. Cho, M. Ferdman, and P. Milder. 2019. Runtime-Programmable pipelines for model checkers on FPGAs. In *29th International Conference on Field Programmable Logic and Applications (FPL'19)*. 51–58.

[15] Radek Pelánek. 2007. BEEM: benchmarks for explicit model checkers. In *Model Checking Software*, Dragan Bošnački and Stefan Edelkamp (Eds.).

[16] Neil R. Storey. 1996. *Safety Critical Computer Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

[17] SwarmWeb 2017. *Swarm Verification Website*. Retrieved from http://spinroot.com/swarm/.

Received July 2019; revised November 2020; accepted January 2021