Runtime-Programmable Pipelines for Model Checkers on FPGAs

Mrunal Patel
Stony Brook University
mkpatel@cs.stonybrook.edu

Shenghsun Cho
Stony Brook University
shencho@cs.stonybrook.edu

Michael Ferdman
Stony Brook University
mferdman@cs.stonybrook.edu

Peter Milder
Stony Brook University
peter.milder@stonybrook.edu

***Abstract*—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 as a promising accelerator because of their high throughput, inherent parallelism, and flexibility. 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 runtime-programmable pipeline 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. Our experimental results show that the runtime-programmable pipeline reduces the preparation time before checking a new or modified model from multiple hours to less than a minute while maintaining similar throughput as FPGA model checkers with model-specific pipelines.

I. 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 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) 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 [1].

Unfortunately, the model checkers themselves are facing performance challenges due to the massive number of reachable states that they must explore. 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 [2], [3]. 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 [4], [5], [6], [7], [8], [9], 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 wide 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 the BEnchmarks for Explicit Model Checkers (BEEM) [10]. 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 resource utilization and execution time overhead of only 26%.

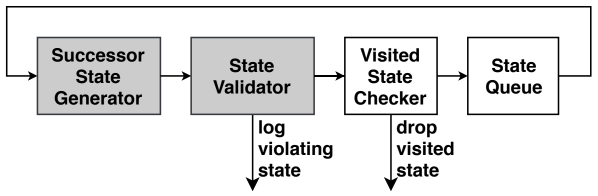

Fig. 1: Explicit state model checking processing flow. Gray boxes are model-dependent.

II. Background and Motivation

A. 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 paper, 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. 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, 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.

B. State Space Exploration

We use an example software model written in *Promela*, the PROcess MEta LAnguage, to further explain how explicit state model checkers explore the model state space. Promela is the modeling language used by SPIN, a widely adopted explicit state model checker, to describe concurrent systems such as multi-threaded software that communicate through global variables or message-passing. Using tools such as Modex [11], Promela models can be automatically extracted from software source code.

Listing 1: Simple Promela example.

```
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");
}
```

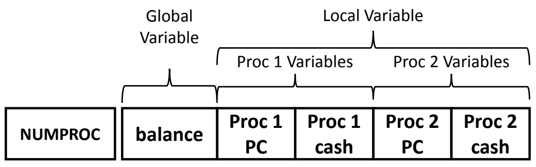

Fig. 2: State vector for the simple Promela example.

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, 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 propriety 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. The local variable *PC* represents the current state in the process state machine. A global constant *NUMPROC* indicates how many processes run concurrently, which is two in our example. 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.

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 II-A generates the state space on-the-fly by using the flow in Figure 1 and the state validator logs any discovered violating states.

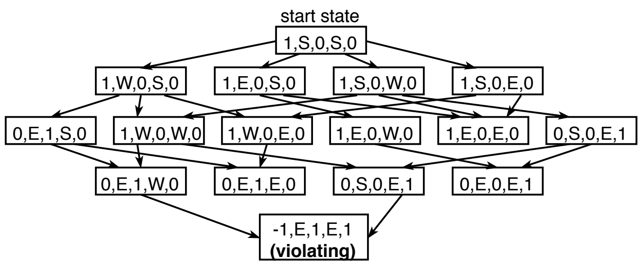

Fig. 3: State space generated from the Promela example.

C. 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 [3], 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 [12] divides the state exploration into many small and independent *verification tasks* (VTs) by explicitly limiting each VT's memory footprint. By utilizing 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 [12]. 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. On the other hand, FPGAs have been adopted for model checking and shown to have promising performance because of their flexibility, high parallelism, and massive on-chip memory bandwidth [2], [3]. FPGA model checkers exhibit up to three-orders-of-magnitude speedup over software based verification, making FPGA accelerated model checking an attractive approach for overcoming the performance limitations of general-purpose cores.

D. 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 [3] 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 size 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 [3]. Then, as we will describe in Section III, 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.

III. 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.

A. 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

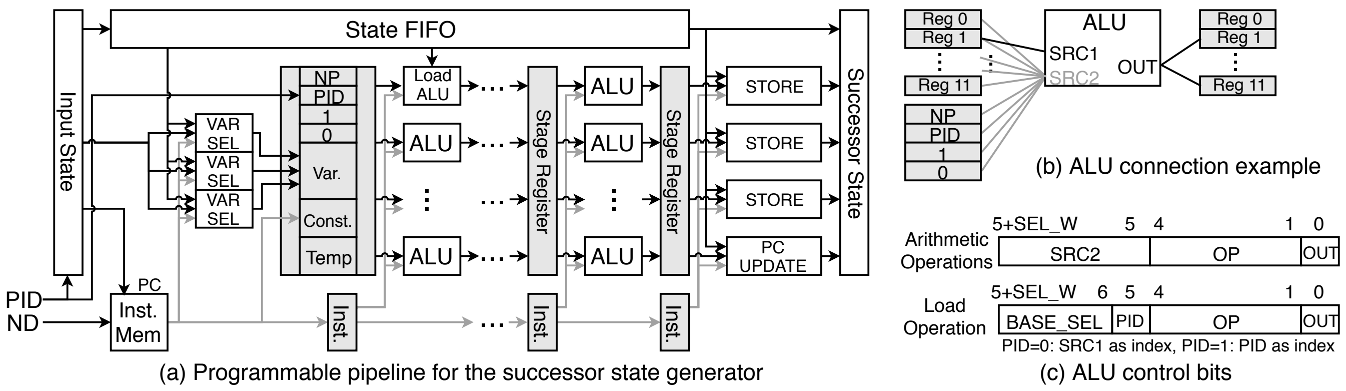

Fig. 4: The runtime-programmable pipeline for the successor state generator. Registers are marked in gray boxes

instructions, ensuring that there is no interference with the other pipelines running in parallel. Prior work [3] 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.

B. 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 PC values. To obtain the PC value, the PID is used to look up the corresponding PC in the state vector. The instruction is fetched from the pipeline's instruction memory (implemented as a 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 PC," 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, 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: NUMPROC (marked as NP in Figure 4), 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

TABLE I: 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

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 I. 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 3 addresses, each of size $\log_2 R$-bits, describing the location in the R intermediate pipeline registers.

Store Stage and PC Updating. 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, the variable in the state vector with position *base* + *index* is replaced with *value*. A special PC updating unit updates the PC for the given PID with the "next PC" constant stored in the instruction, using the same *condition* as the first store unit.

C. 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-PC 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.

TABLE II: Benchmarks from the BEEM database.

Benchmark	Processes	State Vec. Size	Var. Sel. Units	Constants	ALUs	Store Units	Inst. Size
Anderson.8	7	24 Bytes	2	2	2x3	3	131 bits
Bakery.8	5	28 Bytes	2	2	2x5	3	167 bits
Lamport.8	5	20 Bytes	2	2	2x3	2	114 bits
Leader_Filters.7	6	32 Bytes	1	1	2x4	1	107 bits
Mcs.6	5	24 Bytes	2	2	1x1	2	64 bits
Peterson.7	5	28 Bytes	3	1	2x4	2	129 bits
Superset	7	32 Bytes	3	2	2x5	3	172 bits

D. 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.

IV. Evaluation

We evaluated our programmable pipeline by embedding it into FPGASwarm, the state-of-the-art FPGA model checker [3]. Our evaluation studies the overhead of the programmable pipeline in terms of FPGA resource utilization and performance when compared to model-specific pipelines.

A. Implementation

We implement the programmable successor state generator pipeline using SystemC and Xilinx Vivado HLS similar to FPGASwarm, targeting medium-size FPGAs (Virtex-7) and projecting to large FPGAs (UltraScale+). The programmable pipeline replaces the model-specific successor state generator pipeline of the FPGASwarm design. We modified FPGASwarm [3] 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 (i.e., a 4K-deep state queue, 64KB visited state storage, and a 250MHz clock).

B. Methodology

The BEEM database comprises a large variety of model checking problems. We selected a sample from the suitable models available in the BEEM database as our benchmarks and found that support for many Promela features is not required for practical models. Most models can be implemented without Promela channels or even 4-byte integers. Notably, we found that some benchmarks use the expensive MOD operation for

TABLE III: Resource utilization and number of VT cores on Virtex-7 (actual) and UltraScale+ (projected) FPGAs.

	Programmable Pipeline					Model-Specific Pipeline					
Benchmark	**LUTs**	**FFs**	**BRAMs**	**Num. Cores Vtx-7**	**Num. Cores US+**	**LUTs**	**FFs**	**BRAMs**	**Num. Cores Vtx-7**	**Num. Cores US+**	**Prep. Time (m)**
Anderson.8	274,286 (63%)	435,023 (50%)	1,302 (90%)	30	213	202,836 (47%)	360,583 (42%)	1,247 (85%)	34	240	137
Bakery.8	265,146 (61%)	383,062 (44%)	1,202 (82%)	25	179	182,216 (42%)	335,360 (39%)	1,217 (83%)	31	204	120
Lamport.8	273,764 (63%)	448,645 (52%)	1,281 (87%)	33	221	187,138 (43%)	393,595 (45%)	1,189 (81%)	37	245	126
Leader_Filters.7	232,097 (54%)	370,423 (43%)	1,240 (84%)	25	185	168,776 (39%)	315,313 (36%)	1,227 (83%)	28	200	104
Mcs.6	255,897 (59%)	412,638 (48%)	1,252 (85%)	31	229	196,495 (45%)	342,937 (40%)	1,213 (83%)	34	240	183
Peterson.7	267,341 (62%)	441,781 (51%)	1,269 (86%)	27	183	186,615 (43%)	350,829 (40%)	1,217 (83%)	31	204	148
Superset	261,179 (60%)	382,669 (44%)	1,239 (84%)	24	178	-	-	-	-	-	-

bounds checking, which we rewrite by replacing the MOD operations with SUB ($N-M$) followed by a conditional assign of $N - M$ instead of N if $N >= M$.

Table II 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.

To evaluate the performance of our design, we directly compare with FPGASwarm [3] and report the relative differences. The FPGASwarm approach fills the FPGA with VT cores 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 cores run independent VTs, making the total execution time for a model proportional to the number of VT cores that fit onto an FPGA. Specifically, the execution time is the total number of VTs required to check the model multiplied by the average run time per VT, and divided by the number of VT cores 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 cores that fit on the FPGA, which in turn is dictated by the BRAM usage of the corresponding VT cores.

C. Overhead of Programmability

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

Table III 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 [3]. 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 cores. Table III also projects the number of VT cores that can fit onto a Xilinx UltraScale+ VU9P FPGA with 90% of BRAM and UltraRAM utilization, showing that the UltraScale+ FPGA can fit 6.5x to 7.5x more VT cores than our target Virtex-7 FPGA.

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 worst performance because it has the smallest state vector. The small state vector allows for a larger number of VT cores with model-specific pipelines to fit on the chip, resulting in higher performance. On the other hand, Leader_filter.7 has the same state vector size as the Superset configuration, resulting in only three VT cores 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.

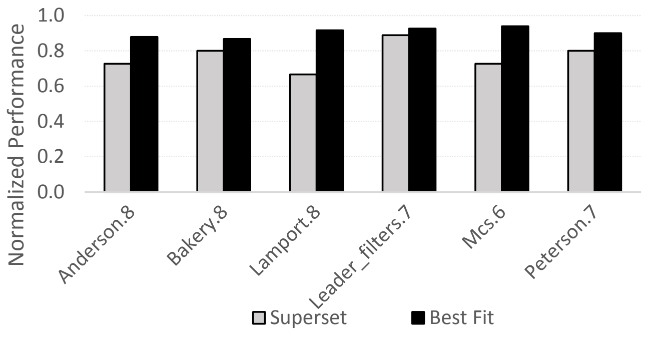

Fig. 5: Overhead of programmability

D. 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 overhead of programmability 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.

E. Preparation Time Comparison

Table III 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 E5-2620v3 CPUs and 64GB 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.

V. Related Work

Researchers have proposed using FPGAs to accelerate the model checking process. [2] built a Murphi model checker on an FPGA, showing 200x speedup over a software implementation using general-purpose cores for a relatively small model. [3] implemented the Swarm verification methodology using FPGAs, showing a 900x speedup over a software Swarm implementation on a synthetic model with 4 billion states. Although both of these FPGA model checkers report impressive performance, they suffer from the long preparation time which 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 [4], [5], [6], [7], [8], [9]. 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.

VI. 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 RTL 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] N. R. Storey, *Safety Critical Computer Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996.

[2] M. E. Fuess, M. Leeser, and T. Leonard, "An FPGA implementation of explicit-state model checking," in *2008 16th International Symposium on Field-Programmable Custom Computing Machines*, April 2008, pp. 119–126.

[3] S. Cho, M. Ferdman, and P. Milder, "FPGASwarm: High throughput model checking on FPGAs," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2018, pp. 435–442.

[4] E. Bartocci, R. DeFrancisco, and S. A. Smolka, "Towards a GPGPU-parallel SPIN model checker," in *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, ser. SPIN 2014. New York, NY, USA: ACM, 2014, pp. 87–96.

[5] D. Bošnački, S. Edelkamp, D. Sulewski, and A. Wijs, "Parallel probabilistic model checking on general purpose graphics processors," *International Journal on Software Tools for Technology Transfer*, vol. 13, no. 1, pp. 21–35, Jan 2011.

[6] T. Field, P. G. Harrison, J. Bradley, and U. Harder, Eds., *PRISM: Probabilistic Symbolic Model Checker*, ser. Computer Performance Evaluation: Modelling Techniques and Tools. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002.

[7] S. Edelkamp and D. Sulewski, "Efficient explicit-state model checking on general purpose graphics processors," in *Model Checking Software*, J. van de Pol and M. Weber, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 106–123.

[8] J. Barnat, L. Brim, and M. Ceska, "DiVinE-CUDA - A tool for GPU accelerated LTL model checking," in *Proceedings 8th Intl. Workshop on Parallel and Distributed Methods in verifiCation, PDMC 2009, Eindhoven, The Netherlands, 4th November 2009.*, 2009, pp. 107–111.

[9] J. Barnat, P. Bauch, L. Brim, and M. Ceska, "Employing multiple CUDA devices to accelerate LTL model checking," in *2010 IEEE 16th International Conference on Parallel and Distributed Systems*, Dec 2010, pp. 259–266.

[10] R. Pelánek, "BEEM: Benchmarks for explicit model checkers," in *Model Checking Software*, D. Bošnački and S. Edelkamp, Eds., 2007.

[11] (2018) Modex - model extraction. [Online]. Available: http://spinroot.com/modex/

[12] (2017) Swarm verification website. [Online]. Available: http://spinroot.com/swarm/