FPGASwarm: High Throughput Model Checking on FPGAs

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*—Explicit state model checking has been widely used to discover difficult-to-find errors in critical software and hardware systems by exploring all possible combinations of control paths to determine if any input sequence can cause the system to enter an illegal state. Unfortunately, the vast state spaces of modern systems limit the ability of current general-purpose CPUs to perform explicit state model checking effectively due to the computational complexity of the model checking process. Complex software may require days or weeks to go through the formal verification phase, making it impractical to use model checking as part of the regular software development process.**

In this work, we explore the possibility of leveraging FPGAs to overcome the performance challenges of model checking. We designed *FPGASwarm*, an FPGA model checker based on the concept of Swarm verification. FPGASwarm provides the necessary parallelism, performance, and flexibility to achieve high-throughput and reconfigurable explicit state model checking. Our experimental results show that, using a Xilinx Virtex-7 FPGA, the FPGASwarm can achieve near three orders of magnitude speedup over the conventional software approach to state exploration.

***Index Terms*—accelerator; model checking; formal verification**

I. Introduction

Testing and verification is one of the most important parts of the development process for mission-critical software used in systems such as medical devices, transportation, and aviation. Any possibility of errors, however rare, puts humans and property at risk and may have devastating impacts, including the loss of lives. Traditional unit-testing already places an immense burden on the developers, and yet is unable to cover all execution cases and identify rare bugs. Explicit state model checking enables automated state-space exploration, capable of verifying high-level safety and liveness properties of systems by methodically traversing the state transition graph of the software under all possible input conditions and verifying if any of the reached states violate the specification.

Unfortunately, the rapidly growing complexity of software makes it increasingly harder to verify. Even simple routines in modern applications can have tens of billions of states. Exploring each of the states requires a model checker to perform several computationally-expensive tasks, including successor state generation and state hashing, that cannot be executed efficiently by general-purpose CPU cores. As a result, verifying complex software with state-of-the-art model checkers [1] is extremely slow, requiring days or weeks of computation.

One promising solution to the scalability challenges of traditional explicit state model checking is the Swarm verification approach [2]. Swarm verification utilizes parallelism to independently execute a large number of verification tasks while relying on randomized diversification of these verification tasks to statistically ensure high state-space coverage. Instead of sequentially exploring all possible system states, which would require an enormous amount of memory and compute, Swarm verification partitions the model checking operation into small verification tasks (VTs) by limiting the memory consumption of each VT, and running a large number of such VTs in parallel. To achieve high coverage, Swarm verification leverages the statistical nature of hash collisions within the small VTs to ensure that the VTs explore different paths within the state space. By running a large number of small VTs across many cores, Swarm verification has been demonstrated to accelerate model checking jobs by an order of magnitude [2]. However, the overall model-checking performance is still limited by the capabilities of the general purpose CPUs on which the tasks run.

In this work, we observe that, unlike conventional explicit state model checking, which requires huge memory capacity to keep track of the already-visited states and the state exploration frontier, Swarm verification tasks require limited memory capacity and massive parallelism. In fact, these are ideal conditions for an FPGA implementation. Taking advantage of these characteristics, an FPGA implementation of Swarm verification can limit the memory capacity to the size of the on-chip Block RAM (BRAM). Doing so provides huge memory bandwidth (hundreds of GB/s random access with fixed latency) to the hardware design to support the required parallelism. Furthermore, FPGAs are an ideal fabric to implement dedicated logic for successor state generation, state hashing, and state validation, the most computationally-intensive parts of the model checking process.

Based on these observations, we developed FPGASwarm, a hardware architecture that adapts the concept of Swarm verification to the FPGA environment, resulting in a design that achieves unprecedented performance for explicit state model checking. The computational components of each FPGASwarm core are automatically generated using High-Level Synthesis (HLS) to avoid time-consuming RTL implementation and to enable the automated translation of software models into FPGA implementations of successor-state generator and state verification functions. These components are deeply pipelined to achieve one-state-per-cycle verification throughput at high clock rates. Because each FPGASwarm core requires little on-chip resources, we instantiate many cores to gain performance through parallelism, partitioning the on-chip BRAMs among the cores and studying the trade-off between core parallelism and per-core memory capacity.

To evaluate our FPGASwarm design, we leverage the methodology for evaluating Swarm verification [2], using a synthetic model with a controllable number of states and easily measurable exploration coverage. We implement the FPGASwarm using SystemC HLS, targeting a PCIe based FPGA board with a Xilinx Virtex-7 FPGA. Running at 250MHz, our FPGASwarm implementation achieves near three orders of magnitude speedup over Swarm verification running on a leading-edge multi-CPU server, bringing down the verification time of a model with 4B states from three hours down to a few seconds and definitively demonstrating the ideal fit of the FPGA platform for explicit state model checking.

The rest of this paper is organized as follows. Section II provides an in-depth discussion of explicit state model checking, the Swarm verification technique, and the motivation behind our FPGASwarm approach. Section III describes the FPGASwarm and its implementation. Section IV presents our evaluation and results. Section V discusses related work and Section VI concludes.

II. Motivation and Approach

A. Introduction to Model Checking

Explicit state model checking aims to exhaustively check all reachable states of a software application, known as the *state space*, to confirm that all reachable states meet the developer-specified safety and liveness properties [3]. This paper focuses on using explicit state model checking to determine whether a software implementation meets safety properties by checking if there exists a set of conditions that would result in the software entering an illegal state.

Explicit state model checking works by traversing the entire state transition graph of the software being verified. States are represented by the model checker as bit vectors that describe the values of the software variables and hardware registers (e.g., the program counter). Due to its large size, the state transition graph is generated on the fly by a successor state generator function, which transforms any valid system state (bit vector) into a list of the possible states that the system can transition to within one time step. As an example, when verifying a program that includes a “switch” statement whose active branch depends on an external program input, the successor generator function will produce a successor state for each of the possible execution paths.

Figure 1 shows the basic process of explicit state model checking. The *State Queue* is initialized with one or a set of starting states. The *Successor State Generator* takes states from the State Queue and produces all possible successor states based on the software model, passing the successor states on to the *State Validator* to check whether any of the generated states violate the specification. Each non-violating

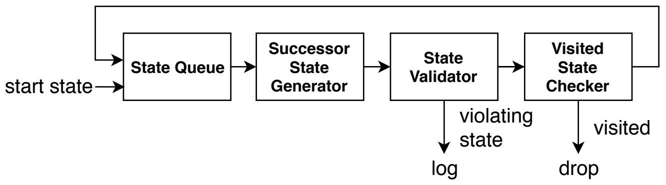


Fig. 1: Explicit state model checking processing flow. If all inputs to the visited checker are reported as visited, eventually the state queue will become empty and the model checking is considered finished.

state is then passed through the *Visited State Checker*, which determines whether or not this state was visited previously, also updating the underlying data structure to indicate that the state under consideration has now been visited. All previously unseen states are enqueued into the State Queue and the process continues until the State Queue becomes empty, indicating that all reachable states have been explored.

The Visited State Checker is a critical component of explicit state model checking, ensuring that the model checking process does not enter infinite loops. A naïve implementation of the checker would use a large lookup table, indexed by the state bit vector, pointing to a single bit for each state to indicate whether or not that state has been visited. However, the size of this lookup table would be prohibitively large for anything but the most trivial models. The commonly accepted solution, called bit-state hashing, limits the size of the data structure by hashing the state bit vector into a smaller index. However, due to the smaller structure and the possibility of hash collisions, the checker may yield false positives, erroneously identifying previously unseen states as having been visited and preventing the model checker from exploring the entire state space. Bit-state hashing potentially results in a reduction of the model checker *coverage* because not all states are visited. However, this trade-off is accepted for the verification of complex software, because bit-state hashing makes the memory requirements of the Visited State Checker feasible while still providing a statistical bound on the explored fraction of the state space.

B. The Challenge of Model Checking

The state space of a software application is described by the values of all software variables and hardware registers. As an example, modern avionics software has many tens of billions of states [4]. As the number of variables grows with the complexity of the software, the state space grows exponentially with the number of bits of the state vectors. This problem, dubbed as “state explosion,” is generally considered the main challenge of model checking.

Furthermore, the operations that the model checker must perform on each state, including successor state generation, state validation, and visited state check, are computationally expensive. This is particularly noticeable when the model checker runs on a general-purpose CPU, where each state needs a significant amount of processing time. The combination of the poor performance of general-purpose CPUs on these tasks and the immense state space size make verifying complex software extremely time consuming in the best case, if possible at all. For example, a single core can only visit roughly 10^5 states per second on the *fleet* model [2]. The size of the state space of *fleet* is not known, but is guaranteed to be more than 10^{11} states. Using a general-purpose core will require more than eleven days to explore these states.

C. Swarm Verification

One promising approach to overcome the limitations of traditional explicit state model checking is Swarm verification [2], which offers a way to limit the memory requirements of the model checker and to enable effective parallelization of the model checker across a large number of servers. Instead of running a single model checker whose goal is to explore the entire state space, Swarm verification breaks up the state space exploration into many independent *verification tasks* (VTs). Although each VT covers only a small fraction of the state space, running many VTs on many servers (e.g., in the cloud) allows the aggregate system to more quickly explore a large fraction of the overall state space across many VTs, compared to what can be done by a single large verification task.

As proposed in [2], Swarm verification limits the memory requirements and the run time of each VT by constraining the size of each hash table in the Visited State Checker of each VT. This organization creates a trade-off between the memory requirements of each VT and the coverage that it contributes to the overall state space exploration. A smaller hash table results in more false positives in the Visited State Checker due to the hash collisions, which reduces the contribution of each VT to the overall state-space coverage. However, while a single VT can explore only a small part of the state space, the Swarm verification approach relies on *diversification* of the VTs, causing VTs to randomly explore different parts of the state space to ensure that, together, the VTs statistically achieve high coverage. Diversification actually benefits and, in fact, relies on the statistical nature of false positives in small hash tables. Each VT in the Swarm uses a different random hash function for its Visited State Checker. Different hash functions result in each VT experiencing a different set of false positives when determining already-visited states, and therefore lead to each VT diverging in the paths that it explores compared to the other VTs in the Swarm.

When leveraging machines in the cloud, Swarm verification improves the performance of model checking by an order of magnitude [5]. However, the model checking process is still limited by the computational throughput of the individual VTs running on general-purpose processor cores. Moreover, while the parallelism afforded by running across many machines improves performance, this is achieved at a high cost where hundreds of machines are used to run diversified VTs.

D. Accelerating Swarm Verification With FPGAs

In this work, we recognize that the key computational bottlenecks of the verification tasks are a good fit for hardware acceleration. Specialized hardware can efficiently run successor-state generation state machines, and unrolled hash function implementations can be leveraged by the visited state checker. However, even if the hardware implementations can be deeply pipelined, the throughput of traditional explicit state model checking cannot be significantly improved with specialized hardware because of the inherent need to work with a large hash table and state queue structures. These structures must be stored in off-chip DRAM and access to them would form the fundamental bottleneck of the system.

Critically, we observe that adapting the Swarm verification approach to FPGAs removes the reliance on large data structures and enables the use of on-chip BRAMs to build tiny, but extremely fast, VT cores. Although each VT core can explore only a minute part of the state space compared to its software counterparts, it can do so very quickly. By developing effective diversification techniques for an on-chip swarm of VT cores, and by diversifying not just across space (running diverse VTs in parallel), but also across time (by serially running many diverse VTs on each VT core), the aggregate state space coverage of these tiny VTs can far outstrip the coverage achieved by a software Swarm, in a fraction of the time. Additionally, major cloud datacenters already have FPGA-equipped nodes [6], [7], making FPGASwarm verification readily accessible to practitioners.

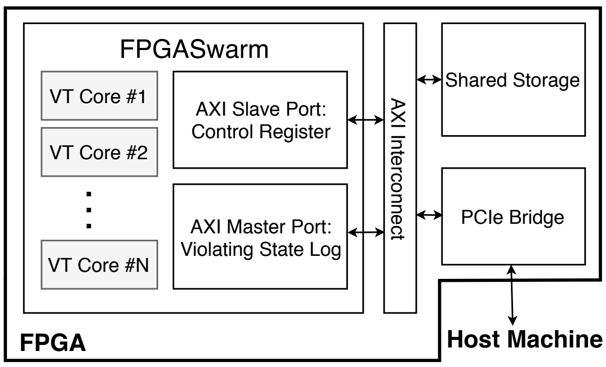
III. The FPGASwarm Architecture

In this work, we propose the FPGASwarm to achieve fast and reconfigurable explicit state model checking using FPGAs. Figure 2a presents the FPGASwarm architecture, comprising many *VT Core* model checkers that independently execute diversified VTs. The VT cores are connected to a set of shared AXI control registers and a memory port. The AXI interconnect provides a standard mechanism to interface with the accelerator and a small amount of shared storage between the accelerator and the host PCIe bridge. The control register includes the *start* and *reset* signals, which can be set with memory-mapped IO requests from the host software via PCIe. The shared storage is used by the host software to deposit the model checker starting state and to read out the violating state log. Because the logic needs of each VT core are relatively small, the number of VT cores that can be instantiated is dictated by the BRAM capacity of the target FPGA platform and the corresponding memory requirements of each VT core.

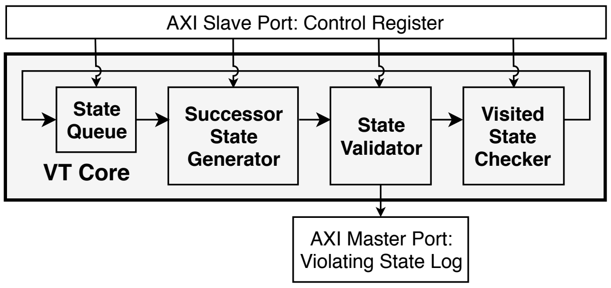
A. VT Core

Figure 2b shows the high-level design of a VT core. Each core has its own instance of the state queue, successor state generator, state validator, and visited state checker modules.

The state queue is a standard FIFO that holds the states representing the exploration frontier. The width of the queue is dictated by the width of the state vector in the software model, and the depth of the queue is limited by the allocated on-chip BRAM capacity, which is split between this queue and the visited state storage. The successor state generator dequeues states one by one, generating a new successor state on each clock cycle and passing it to the state validator module. After confirming that the newly-generated successor state is valid (or recording it in the violation state log), the successors



(a) FPGASwarm FPGA components


AXI Slave Port: Control Register
State Queue
Successor State Generator
State Validator
Visited State Checker
VT Core
AXI Master Port: Violating State Log

(b) FPGASwarm VT core

Fig. 2: The implementation of FPGASwarm. (a) shows the components and connections on the FPGA board. (b) shows the details of one of the FPGASwarm VT cores, including its connections with the shared AXI ports.

are passed to the visited state checker, which inserts them into the state queue after confirming that they have not been previously visited. This organization follows a breadth-first traversal of the state space dictated by the software model, although collisions in the visited state storage randomly prune parts of the state space explored by the running VT.

The successor state generator dequeues one state at a time from the state queue and generates all of the possible successor states, one per cycle. Depending on the model complexity, the successor state generator may be deeply pipelined to achieve high throughput. Similarly, the state validator and the visited state checker may require deep pipelines to ensure the target throughput of one state per cycle.

The state validator checks if a state violates the specification. Violating states can be described either using explicitly specified state vectors or as functions applied on the state vector Upon discovering a violation, the state validator logs the violating state into the shared storage, which is accessible to the host software. To write the violating states into shared storage, each VT includes a FIFO interface to a shared module, which uses an AXI master port to write to memory via the AXI interconnect. The frequency of discovering violating states is extremely low (after all, the goal of the model checker is to verify that there are zero violating states in the entire state space), which makes the performance of this module irrelevant and inspires this simple and low-cost implementation.

The visited state checker comprises a hash function, which transforms a state vector into an index, and a lookup table of bits corresponding to whether or not the state has been previously visited (or a potential false positive). If the hash structure indicates that the state has not been previously visited, the corresponding bit is set to mark the state as *visited*. If the state was previously unseen in this verification task, it is inserted into the state queue. Otherwise, if it has already been visited, the state is simply dropped. To minimize the implementation cost of this structure on an FPGA, the lookup table is arranged into 64-bit words, each corresponding to 64 consecutive states. Despite being deeply pipelined, full forwarding paths on this structure are unnecessary, because the structure need not be precise; making short-term mistakes (by occasionally failing to mark a state as visited or by revisiting a recently visited state) does not violate correctness.

Because the visited state storage is reused by VTs running on the same core, the storage must be cleared between VTs. Clearing the storage one bit at a time would result in the VT core being idle for approximately the same amount of time as running a VT. Arranging the storage as 64-bit words reduces the gap between two consecutive VTs by 64x by allowing the VT cores to clear the storage at coarse granularity. To further minimize the interval between the execution of consecutive VTs, we partition each visited state lookup table into multiple banks and clear the banks in parallel.

In theory, a verification task should end when the state queue is empty and there are no more states to explore. In practice, the deep pipelines of the VT core components make tracking how many states are in flight non-trivial, especially toward the beginning and the end of a verification task, when the state queue may be empty while successor states are making their way through the various component pipelines. Rather than introduce costly mechanisms to track the precise number of in-flight states, we rely on a simple timeout mechanism that tracks the number of consecutive cycles during which the state queue was empty. If the number of cycles exceeds a predefined value (set slightly greater than the pipeline depth of the entire VT core), the VT core considers the current verification task completed and automatically starts the next VT.

B. Memory Usage Strategy

To provide independent and high-bandwidth state queues and visited state storage, these structures are implemented using the on-chip BRAM, which is on the order of a few MBs in our target FPGAs. Compared to the software Swarm verification, where each core has a large visited state storage in main memory, the visited state storage for each VT core is relatively small. This consideration significantly skews the Swarm verification trade-off to a large number of VTs that are toward the extreme end of low-memory requirements.

In addition to the coverage, the size of the visited state storage also affects the run time of each VT. A visited state checker will fill smaller storage faster than larger storage. A full visited state storage causes the visited state checker to report every state as visited and preventing any new states from being inserted into the state queue. Software Swarm verification takes more time to finish each VT because of the larger visited state storage and slower successor state generator, state validator, and visited state checker operations. On the contrary, FPGASwarm VT cores process VTs using small visited state storage and fast dedicated hardware circuits, finishing each VT in a matter of milliseconds. As a result, to compensate for the low coverage of each VT, the FPGASwarm serially runs a large number of diversified VTs on each FPGASwarm core.

C. Diversification

Effective VT diversification is critical to FPGASwarm verification. The system must ensure that the VTs independently explore the state space with high entropy, leading to diversity in their exploration paths through the space. Without sufficient diversification, the VTs would perform redundant work, requiring much more time to fully explore the state space.

We rely on randomization in our diversification strategies. For this purpose, each VT core includes a linear-feedback shift register (LFSR) to generate a sequence of pseudo-random numbers. We initialize each LFSR to a unique value (based on the VT core ID) to ensure that each produces a different sequence. The LFSR state is preserved between VTs, making each VT in the FPGASwarm start with a different LFSR state.

The LFSR state provides seeds for several parts of the verification process. In the successor state generator, the LFSR value is used to randomize the order in which the successor states are generated. This provides *exploration path randomization*, ensuring that different VTs explore the state space in a different order, allowing the exploration to diverge very quickly even through VTs begin exploration at the same starting state. In the visited state checker, the index computation uses the Jenkins hash function [8] to map state vectors to indices in the visited state storage. At the start of each VT, the VT core stores a copy of the LFSR state and uses this value as the salt to initialize each state hash computation of the VT. As a result, the same state will hash to a different index in the visited-state lookup table in every VT, which leads to different hash collisions and promotes diversification and divergence of the VT exploration paths.

Beyond the above explicit randomization, we find that the limited FPGA BRAM capacity available for the state queue further contributes to diversification. Because the successor state generator typically produces multiple successor states for each state, the enqueuing rate of the state queues can be higher than the dequeuing rate, which could cause deadlock in the verification pipeline if the queue becomes full. To avoid deadlock, the FPGASwarm's visited state checker simply drops successor states if no space is available in the state queue. The selection of dropped states depends on the state exploration order and the size of the state queue, essentially dropping states randomly and introducing an additional highly effective diversification mechanism. Unlike the software Swarm implementation, the FPGASwarm benefits in diversification from both the collisions in the limited capacity of the visited state storage and the limited capacity of the state queue.

During the development of the FPGASwarm, we also considered a design where the size of the visited state lookup table would vary across the VT cores, mimicking the software Swarm implementation. However, we found that this technique does not improve performance of the FPGASwarm. In Section IV-D, we show the detrimental performance impact due to excessively small visited state storage tables. Because the BRAM capacity of the FPGA dictates the number of VT cores, varying visited state storage size across the VT cores would either reduce the total number of cores that fit on chip or would result in some of the VT cores incurring performance degradation due to under-sized storage.

IV. Evaluation

This section presents an evaluation of the FPGASwarm design on a Virtex-7 FPGA. The experiments described here allow us to: (1) identify the best set of parameters for the design, (2) measure its run time and compare it to the state-of-the-art software implementation of Swarm verification, and (3) evaluate the FPGA resource consumption and clock frequency.

A. Implementation

We prototyped the FPGASwarm targeting the Xilinx Virtex-7 XC7V690T FPGA. We implemented the design using SystemC with Xilinx Vivado HLS. High-level synthesis is particularly useful for creating pipelined implementations of the successor state generator, the state validator, and the visited state checker hash function. These blocks are specified as Boolean functions written in high-level languages, which are a simple use case for HLS. Critically, using the HLS flow to implement the FPGASwarm avoids the time-consuming and error-prone translation of the model (successor state generator and state validator) from the model specification to RTL implementation. Using a parameterized and template-based SystemC design also allows the FPGASwarm to be easily configured to support any number of VT cores and various amounts of BRAM for the state queues and hash tables, enabling easy experimentation and transition between different FPGA devices. Furthermore, constructing the FPGASwarm using HLS makes the framework much easier to use for practitioners. For example, by leveraging prior work for translating application C source code into Promela models [9] and then translating Promela models into branch-free C [10], a practitioner can implement a high-performance fully-pipelined FPGASwarm without any knowledge of RTL.

B. Evaluation Methodology

We measure the performance of the FPGASwarm in terms of the time to explore the state space. We follow the evaluation methodology used in [2], which uses a synthetic model checking problem: a coverage test that aims to explore all 32-bit integers. The model represents a system with eight software threads, each responsible for a group of 4 bits within a shared 32-bit value. At each time step, a random thread sets a random bit within its group. This model has 2^{32} reachable states, but the order in which the bits are set is chaotic, and the path to reach any specific number is entirely random. 100 of the states (100 random 32-bit integers) are designated as *violating states*. This model enables an easy estimation of the aggregate fraction of the state space that is explored by the VTs because the 100 violating states are randomly distributed throughout the space. The number of violating states discovered by the model checker serves as a proxy for the percentage of the state space explored (e.g., to find 20 unique violating states, the system must explore approximately 20% of the state space).

We determine the runtime required by different configurations of the FPGASwarm by setting the appropriate configuration parameters in our design and running SystemC simulation. This methodology allows us to easily evaluate various configurations without needing to perform synthesis and place-and-route for each design. To compare with software, we also run the same experiments using Swarm 3.2 with SPIN 6.4.7 [1], using an Intel dual-socket server that has two Xeon E5-2670v3 CPUs (24 cores total) running at 2.3GHz and Hyper-Threading enabled (48 hardware threads total), with 128GB of RAM.

To evaluate the implementation costs of a configuration of FPGASwarm, we synthesize its SystemC specification using Xilinx Vivado HLS 2017.4 and perform synthesis and place-and-route using Xilinx Vivado 2017.4, evaluating the clock frequency and resources required (logic and block RAM).

C. Number of VTs for Full State Space Coverage

We first explore whether the FPGASwarm is capable of achieving high state space coverage even with a limited visited state storage budget. As described in Section III-B, there is only a small amount of memory available for each VT core to store the states it has already visited; once this memory is full, the VT terminates, and a new VT must launch on the core. This consideration is further compounded when many VT cores are used, as the BRAM must be divided among the cores. Because the memory available per VT is much lower than in a software Swarm implementation, we first conduct an experiment to demonstrate that the entire state space can still be covered even with this limitation.

We created a set of designs whose overall BRAM utilization (for all state queues and visited state storage) consumes 70% of the FPGA's BRAM (leaving 30% headroom to avoid place and route congestion). These designs differ in the number of VT cores used, from 8 to 48 in steps of 4; as the number of cores increases, the per-core storage drops. We fixed the state queue size and evenly divided the rest of the BRAM among the visited state storage of all VT cores to examine a reasonable range of the visited state storage tables. We simulated each design until it discovered all 100 unique violating states. Table I shows the number of VTs required to discover all violating states for each design.

The general trend seen in this data shows the dominant trade-off in the FPGASwarm design space. Starting at the top of the table (four cores), we see that, as we start to increase the core count (and therefore decrease the storage available per core), the system generally needs to run fewer VTs to reach full coverage. This highlights the fact that Swarm verification actually benefits from hash collisions, because hash collisions can result in a higher degree of diversification among VTs.

TABLE I: Total number of VTs run until discovering all 100 violating states, shown for various numbers of cores and corresponding visited state storage size.

VT Cores	Visited State Storage per Core (KBytes)	Executed VTs
8	512	316,464
12	320	158,448
16	224	134,000
20	160	78,280
24	128	57,384
28	96	64,232
32	80	42,994
36	72	83,052
40	64	44,640
44	48	45,716
48	40	433,008

In other words, imperfect information regarding which states have already been visited can have a positive effect, leading VTs to explore different parts of the state space.

However, there is an intuitive limit to this trend at the bottom of Table I, showing that, once the per-core visited state storage becomes too small, the number of VTs needed for full coverage begins to grow dramatically. In this situation, the VT cores quickly fill up their under-sized storage and incur many false positives, terminating the VT before it has a chance to explore a significant part of the state space. This results in a dramatic increase in the number of VTs needed to explore the entire state space. We tried increasing the number of cores beyond 48, but observed a dramatic surge in the number of VTs required—more than one million—preventing completion of the cycle-accurate simulation in a practical amount of time.

D. FPGASwarm Performance

After verifying that the FPGASwarm is capable of achieving full coverage, despite limited storage capacity, we evaluate the overall speed of the system. First, it is important to note that the time required to run a VT is not constant; a VT will run until all states from the next state generator are dropped by the visited state checker, preventing further discovery of unvisited states. Therefore, the runtime of a VT depends heavily on the size of the visited state storage.

Figure 3 shows the average run time per VT for various core counts. (Recall that Table I shows how the number of VT cores corresponds to the amount of memory available for each.) Then, Figure 4 shows the total time the entire FPGASwarm takes to explore the entire state space (compounding the differences in the number of VTs that must execute and in the time each requires). For lower VT core counts (more storage), not only does the FPGASwarm need to execute more VTs using fewer cores, but each VT also takes longer to finish exploring its part of the state space. As a result, the total time to explore the entire 4B-state space suffers, taking approximately ten minutes. As the number of VT cores increases, more cores are available to run fewer VTs, and the average VT run time decreases, reducing the overall time to about 10 seconds with designs that use 32 to 44 cores. Further increasing the VT core count results in a increase in run time

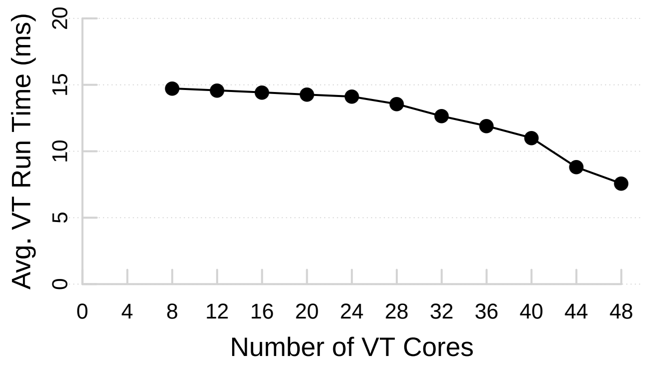


Fig. 3: Average run time per VT under different core count.

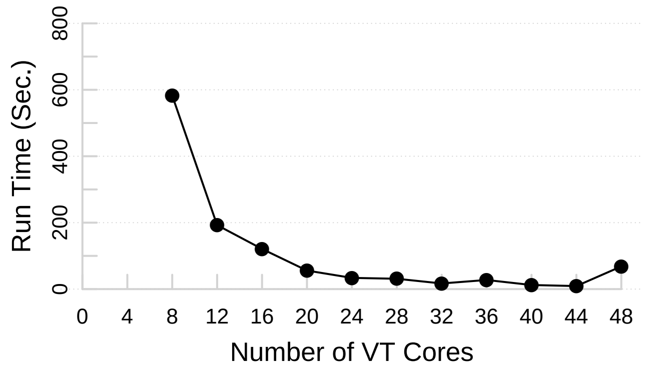


Fig. 4: Total run time required to find all 100 random integers using different numbers of FPGASwarm VT cores.

because the extra cores cannot compensate for the significantly increasing number of VTs that must run.

E. Comparison with Software Swarm

Ultimately, our goal is to demonstrate FPGAs as an effective platform for explicit state model checking, superior to general-purpose CPUs currently used for this work. We therefore compared the performance of FPGASwarm to SPIN, a popular open source software verification tool, running Swarm verification [2]. We configured the software system to use 24 cores (48 hardware threads) and enabled all of the diversification mechanisms available in the software. Swarm requires the user to specify the maximum amount of visited state storage to allocate per VT, in steps of powers of two. We explored this parameter from 32KB, which is similar to the FPGASwarm visited state storage, to 2GB, which is the upper bound of the per-core visited state storage for our machine.

Table II shows the run time required to reach all 100 violating states for the 40-core configuration of the FPGASwarm and the fastest configuration we observed using software Swarm (256MB per-core visited state storage), with the FPGASwarm outperforming software by about 900x. This level of performance improvement from the FPGASwarm is a potential game changer for explicit state model checking and formal verification as a whole. For the first time, complex software functions with billions of states can be run through a model checker at near-interactive performance during the development process. Moreover, the software Swarm utilized all cores of the system at full speed, which consumes approximately 200W. On the other hand, the FPGA we used for the FPGASwarm consumes much less power (approximately 21W reported by the FPGA vendor tools), with the host system being effectively idle (using approximately 70W), demonstrating that the FPGASwarm not only yields far better performance, but also consumes less power. Although a single machine's power consumption is not critical for model checking, it becomes an important consideration when scaling verification to a large farm of machines (e.g., the cloud) to tackle more complex and larger verification problems.

TABLE II: Performance of software Swarm and FPGASwarm.

Software Swarm	FPGASwarm	Speedup
3 hours	12 seconds	900x

TABLE III: Post P&R characteristics of the FPGASwarm.

LUTs	FFs	BRAMs	frequency	power
112,272 (26%)	271,723 (31%)	1,027 (70%)	250 MHz	21 W

F. FPGA Resource Consumption and Clock Frequency

Lastly, we use Xilinx Vivado 2017.4 to synthesize and place-and-route the 40-core configuration of the FPGASwarm, targeting the Xilinx Virtex-7 XC7V690T FFG1761-3. Table III presents the results, including the number of LUTs, FFs, and BRAMs, as well as the frequency and the estimated FPGA power consumption reported by Vivado. Our results show that the BRAMs, as expected, are the critical component, while abundant LUTs and FFs are available. Note that, although 44-core FPGASwarm has better performance than the 40-core configuration in our SystemC simulation, it cannot achieve 250MHz after being implemented on FPGA, thus has lower performance on real hardware.

V. Related Work

The SPIN formal verification tool [1], [11] has been widely used for software model checking. A number of prior works aim to improve the performance of SPIN. For example, [12] partitions the state space across multiple servers and communicates between them to meet the memory needs of larger models, and [13] describes parallelizing its BFS algorithm. [14] uses the Parallel Structured Duplicated Detection (PSDD) algorithm to parallelize SPIN. [15] uses Bloom filters to achieve fast visited-state storage checks with low conflict ratios. Other than improving SPIN, [16] utilizes a parallelization and randomization strategy similar to Swarm verification to speed up the Murphi verification system [17]. These works improve explicit state model checking through enhanced exploration algorithms or data structures. However, their performance is limited by the speed of the general-purpose cores on which they run. Our work utilizes a combination of the Swarm verification approach [2] and FPGA hardware acceleration to improve the performance of model checking, leveraging the trade-off between parallelism and limited available storage, and exploiting the benefit of dedicated hardware design for high-speed computation.

Beyond addressing the performance bottlenecks from the algorithm and data structure point of view, some prior approaches also use hardware accelerators to speed up explicit state model checking. However, these approaches are generally limited in scope or achieve only modest improvements. We are aware of one prior work that leverages FPGAs to accelerate explicit state model checking: [18] implements the Murphi verification checker on an FPGA, reporting a 200x speedup over general-purpose CPUs. However, the models considered by [18] are limited by the system's on-chip memory capacity, and the system was evaluated only on a small model (on the order of ten thousand states), while our FPGASwarm is specifically designed to verify large software models with billions of states using limited amounts of memory. Moreover, [18] relies on hand-implementing VHDL for each model, while FPGASwarm demonstrates the use of high-level synthesis to avoid the time-consuming and error-prone model re-implementation in RTL. Several prior works have leveraged GPGPU accelerators. [10] implemented the core SPIN BFS algorithm in CUDA to run on a GPGPU device, while [19] used a GPGPU to accelerate the PRISM model checker [20]. GPGPUs have also been used by [21], [22], [23] to run the DiVinE model checker [24]. All of these works report that GPGPUs accelerate explicit state model checking by less than 10x when compared with the same software running on general-purpose CPUs. In contrast, we show that the FPGASwarm achieves 900x improvement.

VI. Conclusions

Explicit state model checking has been widely used as a method to formally verify the correctness of software during the development process. However, it takes days or weeks to perform explicit state model checking even on small applications, whereas modern software continues to increase in complexity. In this work, we identified an opportunity to adapt Swarm verification, a technique developed to parallelize explicit state model checking across systems with limited memory, to model checking on FPGAs.

Taking advantage of the small memory requirements of Swarm verification, we pushed this paradigm to its limits in designing an FPGASwarm model checker that uses extremely small on-chip BRAM storage, but compensates by running an extremely large number of fast verification tasks. Our FPGASwarm design provided the necessary parallelism, fast computation, and flexibility to achieve unprecedented levels of performance for explicit state model checking. We showed that the FPGASwarm on a medium-size FPGA can achieve near three orders of magnitude speedup when compared to the state-of-the-art software implementation of Swarm verification running on a top-of-the-line server system, definitively demonstrating the superiority of FPGAs on this task.

References

[1] Spin - formal verification. [Online]. Available: http://spinroot.com

[2] G. J. Holzmann, R. Joshi, and A. Groce, "Swarm verification techniques," *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 845–857, Nov 2011.

[3] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Transactions on Software Engineering*, vol. SE-3, no. 2, pp. 125–143, March 1977.

[4] D. Cofer, "Model checking: Cleared for take off," in *Model Checking Software*, 2010, pp. 76–87.

[5] Swarm verification website. [Online]. Available: http://http://spinroot.com/swarm/

[6] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016, pp. 1–13.

[7] Amazon EC2 F1 FPGA instances. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1

[8] B. Jenkins. A hash function for hash table lookup. [Online]. Available: http://burtleburtle.net/bob/hash/doobs.html

[9] Modex. [Online]. Available: http://spinroot.com/modex/

[10] 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.

[11] G. J. Holzmann, "The model checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, May 1997.

[12] F. Lerda and R. Sisto, "Distributed-memory model checking with spin," in *Theoretical and Practical Aspects of SPIN Model Checking: 5th and 6th International SPIN Workshops Trento, Italy, July 5, 1999 Toulouse, France, September 21 and 24, 1999 Proceedings*, 1999, pp. 22–39.

[13] G. J. Holzmann, "Parallelizing the spin model checker," in *Proceedings of the 19th International Conference on Model Checking Software*, ser. SPIN'12, 2012, pp. 155–171.

[14] E. Burns and R. Zhou, "Parallel model checking using abstraction," ser. Model Checking Software, 2012, pp. 172–190.

[15] P. C. Dillinger and P. Manolios, "Fast and accurate bitstate verification for spin," in *Model Checking Software: 11th International SPIN Workshop, Barcelona, Spain, April 1-3, 2004. Proceedings*, 2004, pp. 57–75.

[16] H. Sivaraj and G. Gopalakrishnan, "Random walk based heuristic algorithms for distributed memory model checking," *Electronic Notes in Theoretical Computer Science*, vol. 89, no. 1, pp. 51 – 67, 2003, pDMC 2003, Parallel and Distributed Model Checking.

[17] D. L. Dill, "The Murphi verification system," in *Proceedings of the 8th International Conference on Computer Aided Verification*, ser. CAV '96, 1996, pp. 390–393.

[18] 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.

[19] 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.

[20] M. Kwiatkowska, G. Norman, and D. Parker, "Prism: Probabilistic symbolic model checker," ser. Computer Performance Evaluation: Modelling Techniques and Tools, 2002, pp. 200–204.

[21] S. Edelkamp and D. Sulewski, "Efficient explicit-state model checking on general purpose graphics processors," in *Model Checking Software*, 2010, pp. 106–123.

[22] 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.

[23] 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.

[24] J. Barnat, L. Brim, M. Ceska, and P. Rockai, "DiVinE: Parallel distributed model checker," in *2010 Ninth International Workshop on Parallel and Distributed Methods in Verification, and Second International Workshop on High Performance Computational Systems Biology*, Sept 2010, pp. 4–7.