Dolmen: FPGA Swarm for Safety and Liveness Verification

Emilien Fournier
Lab-STICC, ENSTA Bretagne
Brest, France
emilien.fournier@ensta-bretagne.org

Ciprian Teodorov
Lab-STICC, ENSTA Bretagne
Brest, France
ciprian.teodorov@ensta-bretagne.fr

Loïc Lagadec
Lab-STICC, ENSTA Bretagne
Brest, France
loic.lagadec@ensta-bretagne.fr

***Abstract*—To ensure correctness of critical systems, swarm verification produces proofs of failure on systems too large to be verified using model-checking. Recent research efforts exploit both intrinsic parallelism and low-latency on-chip memory offered by FPGAs to achieve 3 orders of magnitude speedups over software. However, these approaches are limited to safety verification that encodes only what the system should not do. Liveness properties express what the system should do, and are widely used in the verification of operating systems, distributed systems, and communication protocols. Both safety and liveness properties are of paramount importance to ensure systems correctness. This paper presents Dolmen, the first FPGA implementation of a swarm verification engine that supports both safety and liveness properties. Dolmen features a deeply pipelined verification core, along with a scalable architecture to allow high-frequency synthesis on large FPGAs. Our experimental results, on a Xilinx Virtex Ultrascale+ FPGA, show that the Dolmen architecture can achieve up to 4 orders of magnitude speedups compared to software model-checking.**

***Index Terms*—Model-checking, safety verification, liveness verification, reconfigurable architecture, FPGA**

I. Introduction

Design verification is one of the most important part in the development process of mission critical systems. Moreover, the raising complexity of cyber-physical systems leads to a raise of both the criticity, and the probability of occurrence of faults, with dramatic potential consequences. Amongst the numerous verification techniques proposed in the literature, model-checking [1] is an intuitive automated method, showing good debugging capability, and is already widely used in the industry. This technique is based the computation of every reachable state of the system, verifying the related properties concurrently. However, in a context of growing complexity of systems, model-checking is facing a state-space explosion problem.

Amongst numerous research efforts to repel this problem [2]–[4], *swarm verification* [5] is an approximation technique relying on the execution of a large number of randomly diversified verification tasks to statistically ensure high state-space coverage. This technique provides a proof of failure, on problems intractable using model-checking, by trading off the completeness of the analysis for significantly improved scalability. The use of reconfigurable architectures for swarm verification shows orders of magnitude better results [6], [7] than conventional software. FPGASwarm [6] shows a 900x speedup over SPIN [8]. Through algorithmic improvements and pipeline optimizations, the Carnac verification core [7] raises the gains by 7.58X, arriving at a 7020X speedup over conventional software. However, these cores are **limited to the verification of safety properties**, which check that "something bad will never happen". This class of properties can be verified by reachability analysis of the state-space. Liveness properties [9], on the other hand, complete the systems specification by expressing what the system should do, "something good must happen". Verifying liveness properties requires reasoning on infinite execution traces, which cannot be accomplished by reachability analysis alone. In practice both safety and liveness verification are of paramount importance to ensure system correctness.

In this paper, we address this limitation by introducing the Dolmen architecture, the first high-performance FPGA swarm verification engine that support both safety and liveness verification. The verification procedure searches for accepting traces on the intersection between a model and a complemented specification (property), both interpreted as büchi automata. To achieve this, the Dolmen verification cores (VCore) implement a bloom-filter-based variation of the nested depth-first-search (DFS) algorithm [10]. The VCores are integrated in a distributed swarm architecture. The swarm behavior is orchestrated through a n-ary distribution tree, which reduces the data-locality problems, improving the scalability on large Stacked-Silicon Interconnect FPGAs (SSI FPGA).

The approach is validated on ten large models from the Beem model-checking benchmark [11] on a Virtex Ultrascale+ FPGA. The models are converted to synthesizable VHDL, and integrated in the VCore using a generic interface. The Dolmen swarm runs with up to 32 cores at 100MHz. The results show up to 4 orders of magnitude faster error detection compared to the Divine software model-checker [12].

The paper is organized as follows. Sec. II provides some background and overviews the related works. The Dolmen swarm engine is presented, in Sec. III. Sec. IV discusses the FPGA evaluation results. Finally, Sec. V concludes this paper

This project has been supported by the French Directorate General of Armaments (DGA), the European Regional Development Fund (ERDF) of the EU, the Brittany Region, the Departmental Council of Finistère and Brest Métropole as part of the Cyber-SSI project within the framework of the Brittany 2015-2020 State-Region Contract (CPER).

emphasizing some future research directions.

II. Background And Related Work

Model-checking provides a generic and powerful automated proof technique based on the analysis of the state-space underlying the execution of a model. Its applicability is based on the hypothesis that the model, viewed as transition-system, induces a finite state-space that can be exhaustively enumerated. The underlying verification problem is to check if the model satisfies a specification. Both, the model and its specification, are viewed as transition systems. Model-checking verification consists in checking if the model includes paths of the complemented specification. If the specification is considered correct, its complement includes all unwanted behaviors. Thus, obtaining an empty intersection *proves that the model does not contains unwanted behaviors* [13].

To simplify the verification problem, *non-deterministic finite automata* (NFA) – recognizing finite words – are typically used for the subclass of **regular safety** specifications. In this case the verification procedure simply requires the computation of the state-space, by any reachability procedure while looking for the unsafe states [13]. In general, however, the specification language is based on ω*-automata*, which can recognize infinite-words in ω*-regular* languages. This encoding is generic enabling the specification of both safety and liveness properties. In this study we rely on a class of ω*-automata*, named non-deterministic büchi-automata (NBA), for encoding of the specification. NBA are syntactically similar to NFAs, however, their interpretation of the accepting states is different. A NFA accepts a word if it contains a trace of the word ending in an accepting state. A büchi automaton accepts a word if it contains an infinite trace in which at least one of the infinitely often occurring states is accepting. From a verification algorithm perspective, to obtain a proof of failure with büchi automata requires the detection of a lasso starting in the initial state and ending with a cycle containing at least one accepting state [13], see Fig. 2a for an example of an accepting lasso.

Model-checking suffers from the state-space explosion problem. The literature is rich with approaches addressing this problem [2]–[5]. Amongst these, swarm verification [5] drops the exhaustivity requirement in exchange for better scalability. Swarm verification relies on the diversified execution of a large number of *Verification Tasks* (VTask) to statistically ensure a high state-space coverage. The diversified execution is achieved by mapping each VTask to specialized instances of the *Verification Cores* (VCore) available on the underlying computing architecture. VCore-specialization assures that the execution of a VTask is statistically focused on a partition of the state-space. Put differently, the execution of the same VTask on different VCores should unravel different partitions of the state-space. When no counter-example is found, the termination of each VTask is ensured by the saturation of the bloom-filter. However, without a global state-storage to ensure termination (by full state-space coverage), the swarm verification can only terminate through a timeout. Thus, the raw execution speed of each VCore becomes the dominating factor impacting verification coverage.

The democratization of reconfigurable devices, in conjunction with the long verification times, pushed the verification community to consider the use of FPGA accelerators, which offer a high-degree of parallelism at a reduced cost. PHAST [14], the first exhaustive hardware model-checker, inspired by murphi [15], showed a 200X faster than Murϕ. Based on the swarm verification technique [5], FPGASwarm [6] shows a 900x speedup over SPIN [8]. Building on these previous results, Menhir [16] proposes a polymorphic verification core, which offers a continuum between partial and exhaustive verification. However, its architecture is limited to a single verification core. The Carnac swarm verification [7] builds on these previous works, raising the gains by 7.58, which today arrive at a 7020X speedup over conventional software.

While showing impressive performance results, these research efforts are exclusively focused on the verification of safety properties. Our contribution addresses this limitation and proposes a swarm verification architecture supporting both safety and liveness verification.

III. Swarm Verification of Safety and Liveness on FPGA

This section overviews the Dolmen architecture, discusses the implementation of the acceptance cycle detection in the verification core, and its integration in a distributed swarm. For homogeneity, in the following, we will adhere as much as possible to the vocabulary proposed in [7].

A. A Verification Core for Acceptance Cycle Detection

The verification of both safety and liveness properties can be reduced as the problem of finding acceptance cycles in a büchi automaton, resulting from the composition of the model with the property.

1) Composing the Model with the Property: Both the model (*Next-state Generator* in Fig. 2b) and the property (*Buchi Property* in Fig. 2b) are encoded as NBA transition-relations, each with it's own internal state. Once a state is produced by the transition-relation of the model it is forwarded to the transition-relation of the property, which observes the state of the systems, and evolves it's own internal state. A composite *state* is simply the concatenation of the system's internal state, and the property state.

The *Predicate Checker* is a generic assertion component which evaluates each composite state produced. In our case, the *Predicate Checker* asserts: 1) if a given state is a büchi accepting state, in the Prefix core; 2) if a given state is equal to the searched accepting state, in the Cycle core. In the pipeline, the Predicate checker is located before the next-state generator. This ensures that each accepting state is detected only once, since the *Known Set* filters the duplicates.

2) Partial Reachability on a Nested-DFS VCore: The high-level behavior of each VCore is based on Nested DFS, the conventional cycle detection algorithm [10]. This algorithm,

sketched in Fig. 1, detects büchi acceptance cycles by interleaving a prefix detection phase with a cycle identification phase. A prefix task starts from the initial state of the model, and performs a reachability analysis of state-space on the composition between the system and it's property (both interpreted as NBA) looking for accepting states. Once an accepting state is reached, a secondary reachability task is started from the accepting state, looking for a cycle, corresponding to a path in the state-space from the accepting state to itself. 1) If the accepting state is not found during the exploration, the prefix task resumes, and the previous operation is repeated for each accepting state until the whole state-space is searched by the prefix core. 2) Alternatively, if a cycle is found both tasks end and the property violation is propagated to the VCore controller.

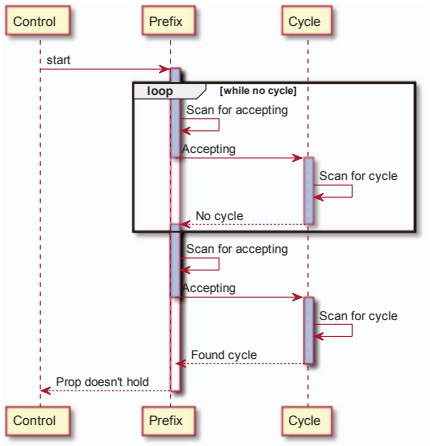

Fig. 1. High-level overview of the Nested-DFS algorithm

The conventional Nested-DFS algorithm supposes both reachability phases are performed using exhaustive algorithms. On FPGA, however, exhaustive reachability is limited to small models, for which the gains of hardware acceleration are limited, as software verification times are acceptable. Nevertheless, partial reachability on FPGA shows promising results [6], [7], [16].

The *Dolmen Verification Core* is composed of two partial reachabily cores (a prefix core, and a cycle core) and a controller, as illustrated in the Fig. 2b.

a) The Prefix Core: left of Fig. 2b, is responsible for detecting the accepting states. To accommodate FGPA constraints, especially the scarce amount of disposable on-chip memory, the *Known Set* is implemented as a Bloom filter, and the Frontier Stream is a stack backed up by a bounded-memory circular buffer, overwriting oldest states when full. This design choice results in the same algorithmic behaviour as the Frontier-Bounded DFS algorithm, presented in [7]. Using this algorithm, the partiality of the exploration, mandatory for efficient swarm behaviour, is introduced by two points: 1) The Known set is implemented as a Bloom filter. This hash-based probabilistic set representation introduces false-positives when filling. In our case the false-positives corresponds to states not yet visited marked as visited, thus not forwarded to the Frontier Stream for further exploration. This prunes paths of the state-space, resulting on a partial coverage of the state-space. This behaviour is used advantageously in a swarm context, as each verification task is started with a different hash seed, resulting in different sections of the state-space pruned by each task. 2) The Frontier Stream is memory-bounded, storing a relatively small number of states (relatively to the bloom-filter capacity), which further prunes the state-space.

b) Cycle Core: When receiving an accepting state from the *Prefix Core*, a second, nested, reachability analysis is started by the *Cycle Core*. This exploration starts from the received accepting state that is forwarded to the *Next-State Generator* as an initial point of the analysis. Globally, the *Cycle Core* implements the same partial reachability algorithm as the *Prefix Core*. Each successor state is forwarded to the Property, which produces the composite successors, the duplicates are filtered by the *known set*, and forwarded to the Frontier Stream, waiting to be popped by the *Next-State Generator*. The main difference in this case is the *Predicate Checker*, which now checks the equality between each state and the accepting state that seeded the current run of the *Cycle Core*. The termination occurs either by finding a cycle from the accepting state to itself, or by emptying the pipeline when the *Frontier Stream* is empty. While not being represented in Fig. 2b for clarity concerns, the termination checkers of both the *Prefix* and *Cycle* cores receive the idle status signals from each entity in the pipeline, which is asserted only when no state is being processed.

Once a termination condition is reached by the *Cycle Core*, the core is fully reset, including the URAM slices used internally in the Frontier Stream and the Known Set, which should be reinitialized by writing sequentially zero's on each address.

3) VCore controller: Both verification cores run under the supervision of a local core controller, which connects the *VCore* to the n-ary communication tree of the swarm engine. This component receives start orders from the central swarm controller, along with the maximum number of tasks to be ran on the core. After receiving the start order, it resets both cores, and sends the start signal to the *Prefix Core*, along with the hash seeds needed by the bloom filters. The seeds are generated through a linear feedback shift register, initialised with the index of each core, so that seeds used by each Büchi-VCore are different from one-another.

This architecture has the advantage of simplicity, and a good level of explicability. Moreover, it is critical that the *Cycle* core starts with a clean Known set, as the acceptance cycle may share states with the trace prefix. These states would be pruned if the exploration is performed with a single Known set without modification. Similarly, the Frontier Stream should be separated between the prefix, and cycle task.

In practice, however, most of the model-frontend, namely the transition relations of the system and of the property may be shared, as the prefix core is idle when the cycle core runs.

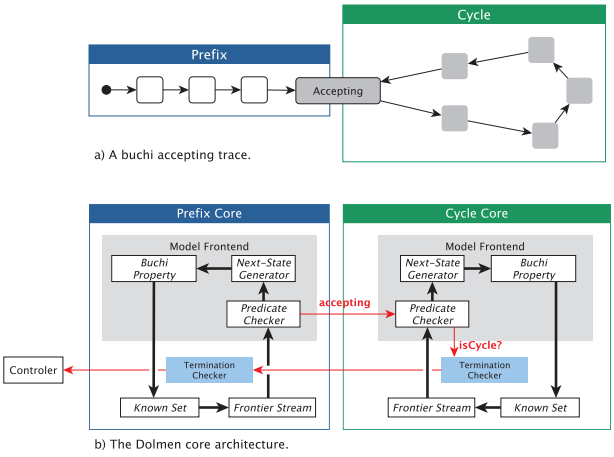

Fig. 2. A büchi accepting trace and the Dolmen core architecture overview. The *Prefix* and *Cycle* boxes imposed over the trace (a) show the role of the two cores (in b) in the acceptance cycle detection algorithm. The *Prefix Core* identifies the finite prefix of the trace. The *Cycle core* unravels the cyclic suffix of the trace.

This optimisation is out of the scope of this study, and it is left for future work.

B. Integration in a Distributed Swarm Architecture

The VCore is integrated in a distributed swarm architecture.

1) Swarm controller: The Dolmen swarm architecture is inspired by the Carnac architecture [7], which is structured around a central *Swarm controller* driving the scheduling of the verification tasks on a large number of *Verification Cores* (VCore). A high-level overview is presented in Fig. 3, which describes the structure of the verification swarm. The *Swarm controller* is responsible for the task-level behaviour of the swarm engine. It receives the start signals, along with the maximum number of *VTasks* to be run, from the driving software. Moreover, it retrieves the results from the cores, whether a *ended* signal, when all tasks have been ran without finding a counter-example, or the accepting state involved in a cycle. As VCores operate independently, performing the same task with different parameters, no individual control is needed by the *Swarm controller*, which avoids the implementation overhead added by a memory-mapped bus.

2) N-ary tree for control distribution.: The spectacular growth in size of reconfigurable devices, along with high reachable frequencies, while being an exciting opportunity, introduces data-locality problems, as several clock cycles are needed to cross the device when running at a high frequency. This issues becomes even more critical with recent *stacked-silicon* platforms. SSI-FPGAs, which tends to widespread among in large high-end FPGA devices, are built through the superposition of several reconfigurable dies connected together to form a larger matrix. These devices have the advantage of size, but the non-uniformity of the matrix complexifies the physical synthesis, as die-crossing net delays are far higher than intra-die net delays.

Then, the architecture of the design should be adapted, aiming to reduce net-crossing signals. The Dolmen distributed swarm architecture reduces those dramatically by implementing the communication with the *VCores* is performed through a n-ary tree distribution architecture, conceptually represented in Fig. 3.

Aiming both to reduce congestion around the central controller, and to reduce die-crossing communications, the *Swarm controller* communicates to a single *Tree Node*, which recursively delegates commands to several sub-trees. At the root level, each subtree placement is enforced on SSI dies, reducing die-crossing signals to a single communication channel per die. Each internal *tree node* registers both downwards and upwards signals, respectively from and to the controller, creating a shift register chain from the *Swarm controller* to each *VCore*. Compared to the linear implementation presented in Carnac [7], the n-ary tree distribution shares ressources, and dramatically reduces the fanout of the central *Swarm controller*, reducing routing congestion around the latter.

Moreover, we added shift registers on the links between each internal node, to reduce locality constraints between them. Each parameter of the distribution tree, namely the branching factor, the number of registers between internal nodes, and the number of *VCore*, is configurable, allowing flexible adaptation to any platform. Practically, this flexibility is achieved through VHDL *generics*, along with recursive instantiation.

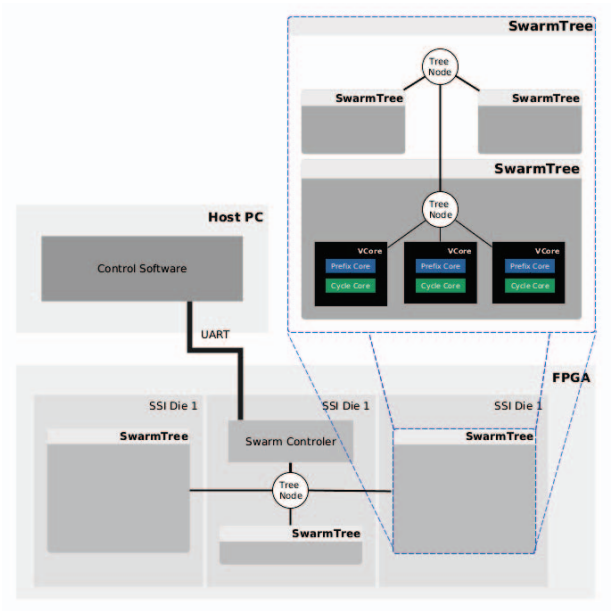

Fig. 3. Architecture overview of the Dolmen Swarm Architecture

C. Software-assisted Counter-Example Generation

When a VCore identifies an acceptance cycle during the execution of a VTask, it propagates a *cycle found* signal, up

through the tree distribution network, to the global swarm controller. This is sufficient for identifying a property violation. However, for diagnosis purposes, the whole trace that contains the accepting cycle is needed. In software model-checking, the counter-example trace is directly obtained from the stacks used by the nested-DFS algorithm. In our case, the usage of a memory bounded stack prevents this opportunity. Nevertheless the counter-example trace can be computed by replaying the VTask with identical random seeds in software, tracing the ancestors in a separate tree data-structure.

IV. Experimental results

This section presents an evaluation of the Dolmen swarm verification design on a Virtex Ultrascale+ FPGA. The experiments described here allows us to : 1) measure the runtime of the design and compare it to state-of-the-art software, 2) evaluate the FPGA ressource consumption and running frequency 3) identify further improvement opportunities.

A. Evaluation Methodology

We measure the performance of the Dolmen architecture in terms of time to find a violation of a liveness property. Previous work proposing hardware swarm verification engines relied on a synthetic case, described in [17]. This synthetic case is a coverage test, looking for 100 random states among all values of a 32 bits integer. This test-case is a good proxy for the performance for reachability, in the case of safety properties, but isn't sufficient to evaluate the acceptance cycle detection algorithm, used here.

For realistic performance measures, we rely on the BEEM Benchmark [11], a comprehensive set of models regrouping common model-checking problems such as mutual exclusion, communication protocols or scheduling. BEEM models are specified using the DVE Language, a low-level specification language based on communicating extended finite-state machines. Both safety and liveness properties are included with each model. For our evaluation purposes, we reformulated these models to a flat transition system, as condition-actions tuples, before generating the synthesizable VHDL 2008 code, which was included in the *Model Frontend* of the VCore. Please note that the model generation chain is out-of-scope of this article, which focuses on the verification algorithm.

Considering the size of the design, RTL simulation times are prohibitive. Thus, the performance values are obtained on a Xilinx VCU128 board which features a Virtex Ultrascale+ XCVU37P FPGA, synthetized using Xilinx Vivado 2019.2. The FPGA implementation results are confronted to software measurements, obtained using the Divine model-checker, version 3.0.90, running on Intel(R) Xeon(R) CPU E7-8890 v4 with 128gb of RAM. To evaluate our architecture, we selected 10 large models from the BEEM benchmark, with sizes varying between 1 million and 506 million states. The selected models are the largest models with properties, which are compatible with hardware generation, skipping models needing complex-to-synthetise features, such as nested array access. Then, among the set of liveness properties furnished by the BEEM Benchmark, for each model, we selected model-properties couples that contains cycles, corresponding to property violations. As stated before, an intrinsic characteristic of swarm verification, whether used for safety or liveness, is that it doesn't provide a termination condition when the property holds. In this case, for partial state-space exploration the performance measurements should be geared towards work duplication metrics, and coverage statistics, in the spirit of [7], which are outside the scope of this study.

The software evaluation is performed on a single exhaustive task, without memory limitations. On the other hand, the FPGA swarm execution suffers from both the partial coverage and from work duplication between the VCores. Nevertheless, the results, shown in Table I, are positive, the FPGA swarm correctly identifies the acceptance cycles if they are present, with reduced bloom-filter capacities.

B. Performance evaluation

The first two columns, of table I, present both the runtime and the number of visited states while verifying the models using the Divine3 model-checker [12], ran with its default parameters (OWCTY algorithm, two threads). The following columns presents Dolmen results in two configurations : Dolmen(19,19), and Dolmen(19, 12). These parameters refers to the address width of the bloom-filter table, respectively for the prefix and cycle core. In the $(19, 19)$ configuration, the dolmen swarm engine shows an average speedup of 50x over Divine, while the $(19, 12)$ configuration is on average 96 times faster (4 874.9X over Divine). This significant performance gain is explained by the faster convergence of the cycle core, due to the smaller bloom-filter size. In the $(19, 12)$ configuration, the cycle core's bloom-filter is $2^7 = 128$ times smaller. In this case, the cycle core executes significantly faster when no cycle is found (the termination occurs when the bloom filter is saturated or almost saturated). This termination case is preponderant during a verification tasks, a large number of accepting states are visited before finding a violation, as shown in the Accepting column of the software results.

From these measurements, another observation is that speedups are highly dispersed, ranging from 1 to 4 orders of magnitude in the $(19, 12)$ configuration. This dispersion comes from several factors. The main factor is the width of model's state. The VCore pipeline is 32 bits wide, which implies that wider configurations are serialized into contiguous chunks. This serialization implies that several cycles are needed to process a single configuration : from 6 cycles for *bakery6*, to 11 cycles for *elevator4*. Another factor that may explain the low speedup of the *elevator* is the relatively low-size of the state-space, that advantages classical model-checking over swarm verification.

The last four column, in Table I, presents the resource utilization results for each model, synthesised with 32 cores, with an exception for the szymanki models, for which the size of the model, in terms of used space, constrained us to synthesise only 16 cores. A point to note is the very high FPGA utilization in any configuration of the core. Resources

TABLE I
FPGA EVALUATION OF THE DOLMEN ARCHITECTURE

model	Divine			Dolmen(19,19)		Dolmen(19,12)					
	Time(s)	States	Accepting	Time(s)	Speedup	Time(s)	Speedup	Cores	LUT	FFs	Memory (kbit)
bakery6prop2	35.52	3.44E+06	1.97E+06	4.27	8.32	0.0406	874.38	32	753737	1377486	40320
bakery6prop3	25.34	2.56E+06	1.73E+06	2.54	9.96	0.0936	270.78	32	753907	1377481	40320
bakery7prop2	307.10	2.05E+07	9.07E+06	4.29	71.63	0.0407	7544.51	32	753847	1377483	40320
bakery7prop3	240.13	1.78E+07	9.39E+06	9.13	26.31	0.0941	2552.02	32	753850	1377487	40320
bakery8prop3	793.45	4.03E+07	2.05E+07	14.32	55.41	0.1361	5827.93	32	969126	1755526	40320
elevator4prop2	84.34	1.36E+06	4.74E+05	258.02	0.33	1.7922	47.06	32	1138247	1809188	40320
elevator4prop3	49.03	1.01E+06	1.18E+05	647.88	0.08	4.5003	10.90	32	1138149	1809189	40320
iprotocol5prop4	1171.75	7.73E+07	1.57E+07	320.82	3.65	3.2683	358.51	32	1102224	1759617	40320
szymanski3prop3	35.21	2.06E+06	1.06E+06	0.42	83.28	0.0043	8172.40	16	942043	1830547	20160
szymanski4prop3	103.37	4.61E+06	2.31E+06	0.42	243.98	0.0045	23090.60	16	942997	1830545	20160

are, for the most part, consumed by the generated model, for instance the iprotocol5prop4 uses 93% of the circuit. Sharing the implementation of the model between the prefix and the cycle core would significantly reduce the area usage of the model. However this point is left for future works.

V. CONCLUSION

In this paper we introduce Dolmen, the first FPGA swarm verification engine supporting both safety and verification properties. The architecture is based on a VCore that implements an on-the-fly nested-DFS acceptance cycle detection algorithm on büchi automata. The VCore is replicated in an distributed swarm controlled through a n-ary tree distribution network, which reduces the locality constraints. The contribution was evaluated on a Xilinx VCU128 board, with a Virtex Ultrascale+ FPGA. Dolmen architecture runs at 100MHz and achieves 4 874.9X speedup when compared to exhaustive software verification using Divine 3.0.90. Currently we are working on: 1) Reducing the physical resource utilisation necessary for the model, by sharing the model between the prefix and cycle modules of the verification core. 2) Performing a large scale design-space exploration for tuning the architectural parameters to the model characteristics, and maximizing the state-space coverage.

REFERENCES

[1] E. Clarke, E. Emerson, and A. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Program. Lang. Syst.*, vol. 8, no. 2, pp. 244–263, 1986.

[2] E. M. Clarke, O. Grumberg, M. Minea, and D. Peled, "State space reduction using partial order techniques," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 3, pp. 279–287, 1999.

[6] 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. 435–4357.

[3] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, Y. Zhu *et al.*, "Bounded model checking." *Advances in computers*, vol. 58, no. 11, pp. 117–148, 2003.

[4] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer Aided Verification*, E. A. Emerson and A. P. Sistla, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 154–169.

[5] G. J. Holzmann, R. Joshi, and A. Groce, "Swarm verification," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, Sep. 2008, pp. 1–6.

[7] E. Fournier, C. Teodorov, and L. Lagadec, "Carnac: Algorithm Variability for Fast Swarm Model-Checking on FPGA," in *2021 31th International Conference on Field Programmable Logic and Applications (FPL'21)*, Aug 2021.

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

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

[10] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis, "Memory-efficient algorithms for the verification of temporal properties," *Formal Methods in System Design*, vol. 1, no. 2, pp. 275–288, Oct 1992. [Online]. Available: https://doi.org/10.1007/BF00121128

[11] R. Pelánek, "Beem: Benchmarks for explicit model checkers," in *Model Checking Software*, D. Bošnački and S. Edelkamp, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 263–267.

[12] V. Štill, P. Ročkai, and J. Barnat, "Divine: Explicit-state ltl model checker," in *Tools and Algorithms for the Construction and Analysis of Systems*, M. Chechik and J.-F. Raskin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 920–922.

[13] C. Baier and J.-P. Katoen, *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[14] 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.

[15] D. L. Dill, "The murϕ verification system," in *Computer Aided Verification*, R. Alur and T. A. Henzinger, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 390–393.

[16] E. Fournier, C. Teodorov, and L. Lagadec, "Menhir: Generic High-Speed FPGA Model-Checker," in *2020 23rd Euromicro Conference on Digital System Design (DSD)*, 2020, pp. 65–72.

[17] G. Holzmann, R. Joshi, and A. Groce, "Swarm verification techniques." *IEEE Trans. Software Eng.*, vol. 37, pp. 845–857, 01 2011.