SoC-FPGA Accelerated BDD-Based Model Checking

Rúben Alexandre Pereira Teixeira
Instituto Superior Técnico,Universidade de Lisboa,Portugal
INESC-ID
ruben.teixeira@tecnico.ulisboa.pt

Abstract

Model Checking is considered one of the most important tools in formal verification, commonly used to verify hardware and software requirements. One of most important breakthroughs in Model Checking was the use of Binary Decision Diagrams (BDDs) to significantly improve the runtime and allowing to tackle larger problems. Because of the fundamental problem that plagues state space exploration systems, Model Checkers are considered too much time consuming in order to be widely used.

In this paper, we propose a HW/SW architecture that uses SoC-FPGA devices to improve Bdd based Model Checkers runtimes. We develop a base architecture and two modifications for a total of four different architectures. The end result comes close but does not improve in terms of performance with regards to a full software implementation.

Keywords

Model Checking, Binary Decision Diagram, SoC-FPGA, Hardware/Software System.

1 Introduction

A Model Checker is a verification tool that verifies if a given Model follows a set of properties. This is performed by state exploration: the state space of the model is explored until a state is found that does not follow the property, or every state has been explored. A type of Model Checker, called Symbolic Model Checkers use Binary Decision Diagram (BDD) to efficiently encode the state space and to efficiently explore it.

Due to the nature of state space exploration, the bigger the model the more time consuming a Model Checking becomes. This problem can be overcome by providing more efficient implementations of BDD operations.

2 Related Work

The usage of hardware controllers to accelerate the performance of Model Checkers is almost done to explicit Model Checkers. These types of Model Checkers represent the state space explicitly and share very little similarity with Symbolic ones.

Works exist that tried to implement simple Bdd operations in hardware [3] or using an accelerator like a GPU [2] or a vector machine[1]. These works do not implement the operations necessary for Model Checking, only a small subset of BDD operations.

3 Background

Symbolic Model Checking represents state space by using BDDs. The properties are specified using Temporal Logic. Temporal Logic is a Logic used to reason about events over time. Checking a Temporal Logic property is performed by a recursive step that computes a state transition at every iteration (among

other operations, depending on the type of property). This computations eventually reach a fixed point, at which point the Model Checking is over.

Model Checkers calculate state transitions by performing BDD operations. These operations are: And,Exist, Relational Product and Substitution. With these four operations it is possible to implement the verification of Temporal Logic.

A BDD is a binary tree like structure composed by terminal and non-terminal Nodes. A non-terminal Node is associated to a variable and has two children, Then and Else. Terminal nodes are two, that represent the constant Zero and the constant One. The Then children encodes the boolean function that arises when the variable is set to 1 and the Else children encodes the boolean function that arises when the variable is set to 0.

Operations over BDD are performed in a recursive manner. They take BDD nodes as input and output the resulting BDD node. They are usually implemented using a recursive approach and because they are non-destructible, its common to implement a Computation Cache to store the results.

The operations can be divided into two phases, Apply and Reduce. The Apply phase fetches the Nodes and the variable to which they are associated. Operations are performed when both Nodes have the same variable. If one Node has a smaller variable, the children of that Node are used as the arguments to the child operation, while the other Node is sent along. When both Nodes have the same variable, both their children are sent as the arguments to the recursive call. The Apply phase defines a Simple Case at which point the recursion ends. This simple case depends on the operation. For example, the And operation return the Constant Zero if one of its inputs is Zero, as the And of Zero is always Zero.

The Reduce phase creates and stores the result of the Apply phase into Nodes. It also performs a search over the Nodes that already exist in order to reuse them if possible. The common implementation of BDDs reuses Nodes that already exist to save space. This type of implementation is called Reduced Ordered Binary Decision Diagram (ROBDD) and is characterized by not only reusing existing Nodes but by also imposing a fixed variable order, such that the children of a variable cannot have a variable bigger than their parent.

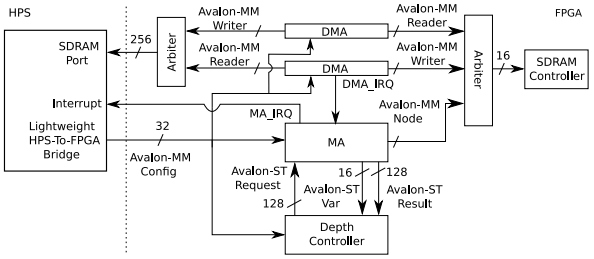

Figure 1: The schematic of the hardware controller implementation.

4 HW/SW Architecture

The proposed HW/SW architecture implements a BDD Controller that performs the four BDD operations mentioned earlier. A Model Checker configures the hardware and then awaits for it to end.

The proposed architecture is depicted in Figure 1. The Depth Controller and the Memory Access are the proposed modules to implement the BDD Operations. The Memory Access module provides an abstraction over the memory so that the Controller can perform BDD operations without having to worry about memory transfers and data coherence with the software system. The HPS system houses the CPU that runs the software.

The modules communicate through Avalon interfaces. The arrows move from the Master to the Slave interface. The Avalon Memory Mapped (Avalon-MM) interface provides a memory type of interface where the master can set an address and perform read and write operations. The Avalon Streaming (Avalon-ST) interface sends data from a Source interface into a Sink interface. Data only moves in one direction but more Avalon-ST interfaces can be used to provide a full-duplex type of communication.

Memory management is made by utilizing a Paging approach. Memory is divided into Pages where each page stores Nodes. In our implementation Pages are also associated to variables and as such every Node

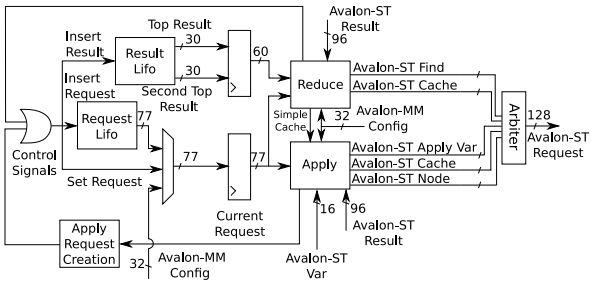

Figure 2: Depth Controller

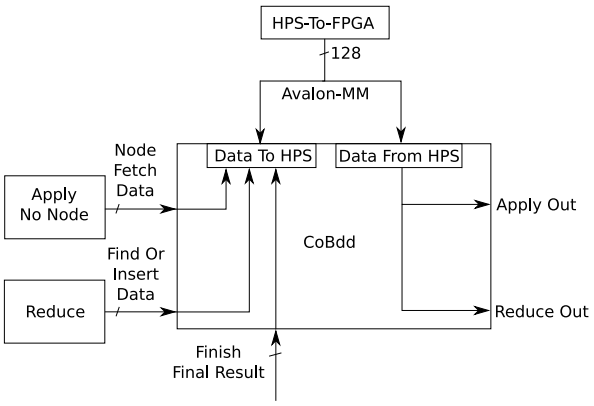

Figure 3: The CoBDD module

inside a Page has the same variable. A page transfer is performed by request from the Memory Access IP. When it detects it needs access to a given Node, it asserts an interruption line. A interrupt routine configures the DMA with the necessary data to transfer the page and acknowledges the Memory Access IP which then awaits until the transfer is complete.

The Depth Controller is the module that performs the BDD operations. The schematics for the Controller are show in Figure 2. The process of accessing memory is performed using a send/receive type of interface with the Memory Access IP. The Controller sends a request, containing information over the type of data it needs to access as well as the operation, write or read, and the Memory Access sends the result back, if any.

The implementation of the Depth Controller resembles an iterative implementation, that utilizes a LIFO to keep track of Requests and Results. During a BDD operation a current Request is set, the initial one being set by the software to start the operation. This Request is then processed either by the Apply or the Reduce modules shown. This depends on the state of the Request. The result of the Modules then creates new requests that are either stored in the LIFO or set to be processed, or pops Requests from the LIFO and sets them as current Requests, if the module produced a result from Reduce or Simple Cases. This process continues until eventually there is no more valid current Requests or Requests stored in the LIFO. This ends the operation and the HPS can fetch the final result.

4.1 CoBDD

The CoBDD modification is a proposed alteration to the previous architecture that does not store node data in the FPGA SDRAM. Instead the Depth Controller interfaces with the HPS to request it to perform any operation related to Nodes. The CoBDD modification is provided in a Module that hides the interfacing with the HPS from the Controller. The CoBDD module is thus responsible for performing the part of the Apply phase that fetches Nodes as well as the part of the Reduce phase that searches or inserts Nodes.

Figure 3 showcases the CoBDD module diagram. The module awaits for data from the Apply or the Reduce modules and sends it to the HPS. It then awaits for the HPS to insert the result which is used as the output of the stage that is being processed.

4.2 Bounded-Depth

The Bounded-Depth architecture augments the Depth architecture by storing and processing multiple requests at the same time. A Data structure called a Context stores various requests as well as their state, ValidApply and ValidReduce, in a bitfield. The Apply and Reduce phases are now performed over every Request inside the Context whose valid bit is set.

An iteration of the Bounded-Depth algorithm per-

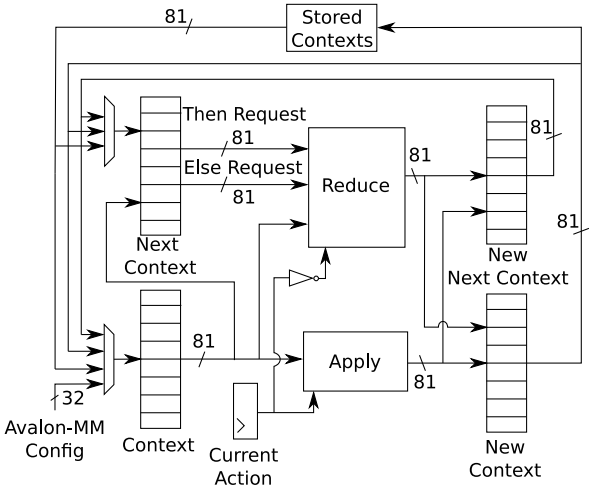

Figure 4: Bounded-Depth Controller

forms a phase, Apply or Reduce, from a Context and the Next Context and creates a New Context and a New Next Context. The Bounded-Depth Controller schematics is represented in Figure 4. Contexts are also stored in a LIFO type structure. After each phase, the result of the phase dictates how the contexts are moved and stored. For example, if the current phase was an Apply that only performed Simple Cases, then the previous Context must be Reduced. The previous Context is stored and as such a transfer from the Stored Context into the Current Context is made. The Context that had the results of the simple cases (New Context) is transfer to the Next Context in order for the Reduce phase to have access to the results computed.

Like the Depth Controller, this process is started by the HPS setting the first Request in the first Context. Afterwards the Controller iterates until it performs a Reduce operation on the first Context, at which point the final result is calculated and the HPS can then access it.

4.3 Bounded-Depth CoBDD

The CoBDD and the Bounded-Depth modifications can be implemented simultaneously. The resulting architecture thus utilzes the Bounded-Depth approach to store and process Requests and the CoBDD approach to perform operations that require Node data.

5 Results

The results where obtained by running a Model Checker, NuSMV, that utilizes a BDD package, CuDD, to perform the BDD operations. Since our data structures are different than the ones used by CuDD, we perform a conversion from the CuDD format into our format. We then use the data converted to run our implementation of the proposed architecture.

The results where collected from one example model that came with the Model Checker. We measured the times taken to perform the Relational Product operation, due to the fact that it is the most important and the most time consuming. The And and Exist operations are in a way also tested since the Relational Product operation simplifies into these operations for some cases.

The implementation of the architectures proposed has a frequency of 100 MHz. The critical path is present in our modules and as such there is room for improving the maximum frequency.

Table 1: Resource usage by implementation

Architecture	ALM	M10K Blocks needed	M10K Blocks used
Depth	7,562	281	368
Depth CoBDD	6,619	89	124
Bounded-Depth	9,144	312	394
Bounded-Depth CoBDD	12,652	124	165
Total	32,070	397	397

The resources utilized by each architecture are shown in Table 1. The Bounded-Depth CoBDD has an unusual higher amount of Adaptive Logic Module (ALM) usage because the Quartus Compiler implemented several optimizations in order to reach 100 MHz of frequency. Since the other implementation had more leeway, and the optimizations are typi-

Table 2: The time results for 8 Relational Product operations in the *msi_trans* example.

Operation Number	CuDD	Depth (no load)	Depth (load)	Depth (no CoBDD	Bounded-Depth (no load)	Bounded-Depth (load)	Bounded-Depth CoBdd
37	1.499	2.056	2.036	2.135	2.186	2.182	2.058
41	1.347	1.750	1.669	1.652	1.841	1.828	1.712
81	1.173	1.835	1.693	1.608	1.856	1.818	1.562
85	2.497	2.861	2.816	2.769	3.011	2.997	2.637
89	2.675	3.042	3.009	3.042	3.181	3.173	3.053
93	3.223	3.482	3.466	3.433	3.663	3.650	3.417
97	1.635	1.803	1.791	1.890	1.940	1.937	1.959
101	1.174	2.078	1.925	1.743	2.201	2.161	1.845

cally not done when the design is already within the frequency requested, they end up using less ALMs. Blocks needed are the minimum amount of memory needed. Because blocks might not be fully utilized or in order to meet certain timings, it is often the case that more blocks are used than the minimum.

Table 2 showcases the times taken for each Relational Product operation. We profiled 8 operations. The preload indicates whether Node data was already present in the FPGA SDRAM or not. All the architectures run at the same frequency and use the same parameters and as such the differences in time correspond to the differences in architecture.

As can be seen, none of our architecture reaches or surpasses the CuDD time. Still, because the implementations can reach higher frequencies with better optimizations, we expected the architectures to be able to at the very least, match the times of the CuDD software. In this case, the operation is not performed faster, but at least in the case of the Depth and Bounded-Depth approaches, it frees the HPS to perform other operations.

At the same time, in order to compare approaches, certain parameters where not fine tuned to the point where they could improve the times. As an example, the CoBDD implementations can use more RAM memory to store Cache. Therefore there are still potently improves to be made.

6 Conclusion

The implementations of the architectures did not improve the performance when compared to the software package. Still, due to how close the results where to match the time taken, and since the implementations can still be optimized further, it should be possible to provide an implementation that at the very least matches the software implementation.

Regardless, further work on this area should focus on maximizing memory throughput. BDD operations are a very memory heavy type of computation. Even though FPGAs are not recommended to deal with memory heavy algorithms, they offer a higher degree of control over how memory is accessed and as such there is potential to develop an architecture that significantly improves BDD operations.

Further work should therefore focus on SoC-FPGAs with high amounts of memory and high memory throughput, and the architectures should focus on trading logic for memory. As seen in our architectures, memory is more important than the amount of logic circuitry available.

References

[1] Ochi, H., Ishiura, N., Yajima, S.: Breadth-first manipulation of SBDD of boolean functions for vector processing. In: Proceedings of the 28th conference on ACM/IEEE design automation conference - DAC (91). ACM Press (1991)

[2] Velev, M.N., Gao, P.: Efficient parallel GPU algorithms for BDD manipulation. In: 2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC). IEEE (Jan 2014)

[3] Yoneda, T., Ishigaki, T.: Hardware acceleration for bdd manipulations (01 2000)