NORTHEASTERN UNIVERSITY

Graduate School of Engineering

Thesis Title: Accelerating Explicit State Model Checking on an FPGA: PHAST

Author: Mary Ellen Tie

Department: Electrical and Computer Engineering

Approved for Thesis Requirements of the Master of Science Degree

Thesis Advisor: Prof. Miriam Leeser	Date
Thesis Reader: Prof. Stefano Basagni	Date
Thesis Reader: Prof. Laurie Smith King	Date
Department Chair: Prof. Ali Abur	Date

Graduate School Notified of Acceptance:

Dean: Prof. Sara Wadia-Fawcetti	Date

Accelerating Explicit State Model Checking on an FPGA: PHAST

A Thesis Presented

by

Mary Ellen Tie

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Master of Science

in

Electrical Engineering

in the field of

Computer Engineering

Northeastern University
Boston, Massachusetts

February 2012

© Copyright 2011 by Mary Ellen Tie
All Rights Reserved

Abstract

Verification has become an increasingly important part of the hardware design process. One technique used to verify digital circuits is explicit state model checking. PHAST, a Pipelined Hardware Accelerated STate Checker, achieves a 30x speedup for explicit state model checking of Murφ models over software implementations used in industry. PHAST is a reimplementation, to accommodate FPGA hardware and to utilize SDRAM, of the Murφ verifier developed at Stanford University. Murφ has been used to verify hardware and protocols, cache coherency protocols particularly. Murφ is used in industry due to its success in finding real design errors. PHAST takes advantage of the flexible memory architecture and inherent concurrency provided by an FPGA to accelerate model checking. In designs such as PHAST, where the data is created and managed locally and the connection is not the bottleneck, FPGAs can yield very good acceleration. The PHAST architecture can handle hundreds of transition relations and tens of thousands of states. Using PHAST, we achieved a thirty times application speedup in actual running hardware compared to Murφ on an example of a counter. The same structure developed for this simple example was also reused for a model of the DASH multiprocessor. The model of the DASH protocol is similar in size and complexity to models Intel uses to validate features of processors. Preliminary analysis of the DASH model as verified by PHAST indicates the speedup will stay constant across a large number of models. PHAST is the first complete implementation of model checking in FPGA hardware.

Acknowledgements

First I would specially like to thank my advisor, Professor Miriam Leeser. This work would not have been possible without her guidance, editing, and high standards. She has provided me with the oppurtunites and support necessary for my personal and professional growth.

Many thanks to Tim Leonard who provided the impetus for this work and spent many long hours providing much of the background, ideas, and name for this work.

I would also like to thank my colleagues in the Reconfigurable Computing Laboratory at Northeastern University for creating an interesting and engaging work environment and my family for always being there. My gratitude goes to Kevin Tie and Linda Nguyen for providing me with much needed daily sanity, love, encouragement and support.

Finally, I would like to acknowledge Intel, without whose funding this would not have been possible.

Contents

1 Introduction **1**

2 Background **4**

2.1 Model Checking 4

2.1.1 DASH Cache Coherency Protocol 8

2.2 Field Programmable Gate Arrays 9

2.3 Related Work 11

2.4 Conclusion 12

3 PHAST **13**

3.1 Model Checking on an FPGA 13

3.2 PHAST Architecture 15

3.2.1 Next State Generator and Invariant Checker 19

3.2.2 Hash Compaction 20

3.2.3 Hash Table Lookup 22

3.3 Conclusions 29

4 Murφ Models **30**

4.1 DOWN 30

4.2 DASH 32

4.2.1 Implementation on PHAST 36

4.3 Experimental Results 38

4.3.1 DOWN 39

4.3.2 DASH 40

4.4 Conclusion 41

5 Future Work and Conclusions **42**

5.1 Automatic Translation from Murφ to VHDL 42

5.2 Future Work 44

5.3 Conclusion 44

Glossary **46**

List of Figures

2.1 Xilinx FPGA chip (from Xilinx) 10

3.1 Algorithm Pseudo-code 14

3.2 Hardware Block Diagram 15

3.3 Parallel Processes Implemented in PHAST 17

3.4 Hash Compaction 21

3.5 XOR Tree Hash Compaction 21

3.6 CAM waveform 25

3.7 Hash Table Lookup and Collision Detection 28

3.8 Collision Reintroduction 28

4.1 Murφ Rule 33

4.2 Murφ Rule translated to VHDL 34

Chapter 1

Introduction

As verification and validation become larger portions of the design process, finding methods and tools to speed that process become more important. The 2006 update of the International Technology Roadmap for Semiconductors states that verification is the dominant cost in the design process and, "without major breakthroughs, verification will be a non-scalable, show-stopping barrier to further progress in the semiconductor industry" [1]. Industry currently finds that verification consumes as much as 70% of the engineering effort required to develop new products. Even with this effort, bugs still make it into the final design. For example, the Pentium FDIV (Floating point DIVision) bug led to a $475 million write-off by Intel. Intel estimates a similar bug in modern Intel processors would cost about $12 billion [2].

Verification methodologies are needed that do not solely rely on simulation of the design. Intel runs simulations involving thousands of machines running for several years to simulate less than a minute of actual running time on the processors they test [2]. Formal verification differs from simulation in that the goal is to prove or

disprove the validity of a design with respect to its specification. Formal verification tools are static in nature; given an input they verify the entire design without testing individual combinations of inputs. A promising formal verification technique is model checking.

Model checking is an emerging method for demonstrating the correctness of hardware and protocols. A designer models a hardware design as a finite state machine, states the properties the design implements, and invokes a model checker to verify that the properties are indeed true in every reachable state of the design. As explicit state model checkers utilize a breadth first search of a very large state space with large states, these model checkers are slow; sometimes extremely slow. An explicit-state model checker enumerates the reachable states of the model being verified, so the runtime can increase exponentially with the size of the model.

FPGAs hold the potential for performance improvement when the application can be reconstructed to take advantage of concurrency, which is the case in explicit-state model checking. This has led many to consider multi-threaded or distributed implementations of Murφ. An FPGA implementation, however, has two advantages: (1) the ability to pipeline steps like next state generation and (2) fast, high-bandwidth access to local memory. Fast access to memory is particularly important, since access to data structures such as the hash table is one of the main bottlenecks. The hash table contains the states reached thus far in the graph traversal and accessing it is difficult to distribute among concurrent threads.

In this thesis, we present an FPGA implementation of Murφ with the goal of accelerating the verification of specific hardware designs. We developed our approach on the example model of a counter, DOWN, which is provided with the Murφ verifier. We illustrate PHAST's capabilities with DASH, a cache coherency protocol, which is representative of protocols that are of interest to today's researchers. The implementation details of PHAST have been presented in [3].

This thesis is organized as follows. Chapter 2 provides the background and motivation behind accelerating model checking on an FPGA as well as introducing the DASH cache coherency protocol. Chapter 2 also explores related work on accelerating both model checking and breadth first searches on FPGAs and GPUs. Chapter 3 presents the PHAST architecture and implementation details. Chapter 4 provides details of both models used to test PHAST and the results from verifying these models. Finally, Chapter 5 contains a discussion on automatic translation of Murφ models to VHDL and other future work.

Chapter 2

Background

As this work presented in this thesis touches on both a formal verification method and the use of programmable hardware, this chapter provides background on model checking, the DASH cache coherency protocol, and FPGAs. Section 2.1 explores the different classes of model checking and explains why explicit state model checking and specifically the Murφ verifier was chosen for use in PHAST. Section 2.1.1 provides the history and introduction for the verified protocol used to test PHAST. In Section 2.3 of this chapter related work in the area of hardware accelerated formal verification and hardware accelerated breadth first search is explored.

2.1 Model Checking

Model checking [4] refers to several methods that automatically check if systems satisfy their specifications. Specifications can include both safety and liveness properties [5]. A designer models a hardware design as a finite state machine, states the properties the design implements, and invokes a model checker to verify that the properties are indeed true in every reachable state of the design. A challenge in model checking

is dealing with the extremely large state spaces that are generated. Despite many efforts to make model checking methods efficient, managing these large state spaces slows down model checking more than practitioners would like.

There are several classes of model checking tools currently used to verify hardware designs. Most of the different classes of model checkers came into use to deal with the state space explosion problem that happens with most models. These model checkers explore the graph that corresponds to all the states that any specific model can exhibit in different ways. Murφ and PHAST are explicit-state model checkers. For all classes of model checking, the model needs to be created by either a verification specialist or the hardware designer, but acceleration of the model checker works best when the verification is automatic.

The Murφ verifier and language have been in use for more than ten years. The Murφ language easily describes hardware and protocol models. These models consist of a description of the state of the hardware, a transition relation that defines how the state can change, and an initial state of the model. The language further provides the ability to specify symmetry in the states allowing the use of symmetry reduction to improve verification [6]. The Murφ verifier has been used in industry to model and verify cache coherency protocols and other hardware designs. Murφ supports both breadth first and depth first search, and checks both liveness and safety properties.

In most model checkers, including Murφ, probabilistic model checking is used to manage the state space. Model checking is probabilistic when instead of creating,

storing and examining every possible state a model can exhibit, the checker generates a state, examines it, performs a lossy hashing method, and stores the hashed data. The use of hashing does not change the verification from explicit to probabilistic state model checking. The verification becomes probabilistic when instead of storing a full state in a hash table, a portion of the hash data is stored as a tag. To do this Murφ uses a method called hash compaction, which was developed for Murφ at Stanford as an improvement over the "hashcompact" method of Wolper and Leroy [7]. As software and hardware designs get larger and more complex, the use of probabilistic model checking becomes more important since the state spaces created by their models grow exponentially. Storing hashed states instead of full states in the hash table saves a tremendous amount of memory at a small but measurable risk of possibly pruning problematic areas of the state space. This pruning happens when two states are hashed to the same hash value. Murφ calculates the probability of this happening for each model verified by Murφ. The probability of omitting even one state is reported during verification.

Symbolic model checking uses binary decision diagrams (BDDs) to represent the state transition graph, without building it, which for small state sizes can make symbolic model checking more efficient. However, when the size of the state is very large, and for many protocols, explicit state model checking can be more effective in finding bugs [8]. Unfortunately, even high-performance explicit state model checkers are slow; sometimes extremely slow. An explicit-state model checker, unlike symbolic

model checkers, enumerates the reachable states of the model being verified, so the runtime can increase exponentially with the size of the model.

Bounded model checking [9] expands the state transition graph for some number of iterations in order to limit the growth of the state space. Bounded model checkers only do full verification for safety properties when induction techniques are used. Bounded model checking needs further user input to fully verify the model. Explicit-state model checking is a fully automatic technique that has been used to great effect in verifying protocols. Explicit-state model checking tools like Murφ have been used to verify key design components that are high risk and have small models, *i.e.*, complex protocols. Even in these cases, the models are often restricted to keep the runtime acceptable. Verifying the least-restrictive model of a design means that the runtime of model checking may take days or weeks or even longer.

For the purposes of this project, we are accelerating a simplified version of the Murφ verifier. PHAST uses breadth-first search to explore the state space and checks only for violations of safety properties, called invariants. Symmetry reduction has not yet been implemented in hardware, but will fit into the PHAST architecture by adapting the Next State Generator. In this thesis we present an FPGA implementation of Murφ with the goal of accelerating the verification of specific hardware designs. We illustrate our approach on an example model of a counter, DOWN, and an example of a cache coherency protocol, DASH, which are provided with the Murφ verifier.

2.1.1 DASH Cache Coherency Protocol

We are currently working on the verification of the DASH protocol [10], a directory based cache coherency protocol developed for the DASH multiprocessor. The multiprocessor is made up of multiple nodes. Each node contains a small number of processors, each with a private cache, a portion of the shared memory, a shared cache, and a directory controller which interfaces the node to the rest of the network.

The distributed directory-based coherence protocol is provided as several examples with Murφ. An elementary, abstract version of DASH is one of the smallest protocol examples distributed with Murφ, and takes Murφ several minutes to verify. We chose to verify this model with PHAST for two reasons. First, we were unsure of PHAST's ultimate requirements in terms of area on the FPGA. It turns out that there are plenty of FPGA resources available. Second, the elementary, abstract version of DASH had the same size state, but much fewer transition relations. Since DASH was handcoded, this meant less time could be spent translation those relations. Once automated tools are in place to translate Murφ models to PHAST, larger protocol examples will be verified. Because of multicore architectures, verifying cache coherency protocols has increased in interest and importance. Our intent is to reduce the execution time for applying model checking to protocols that currently take weeks to verify.

2.2 Field Programmable Gate Arrays

FPGAs have long been used in the development of hardware designs. Traditionally FPGAs were used to prototype hardware designs before deployment with an ASIC device. While not a formal verification method, this allowed designers to run and test their design before committing the design to an ASIC. More recently, FPGAs have gained popularity accelerating formal verification methods including satisfiability [11] [12]. This work represents the first effort we are aware of in accelerating model checking.

FPGAs are chips made up of look-up tables, or LUTs, programmable switches, and wires. This arrangement allows custom circuits to be built in software and then downloaded to the FPGA. While FPGAs are mostly made out of small bits of RAM for logic and programmable interconnect, most FPGAs also include larger blocks of RAM for cache purposes as well as special purpose pieces of logic, such as multipliers and DSP blocks. One of the advantages of FPGAs is the lack of forced architecture or hierarchy inherent in the chip. With FPGAs, a designer can test different circuit architectures, memory hierarchies and caching mechanisms until desired goals are reached.

Figure 2.1 shows a Xilinx Virtex FPGA with the logic and RAM marked. This is a conceptualized diagram of how a FPGA looks. New FPGAs are mostly made up of interconnect that is necessary to support the sizable circuits that can fit on the chips. Typical usage of an FPGA makes the connection between the FPGA

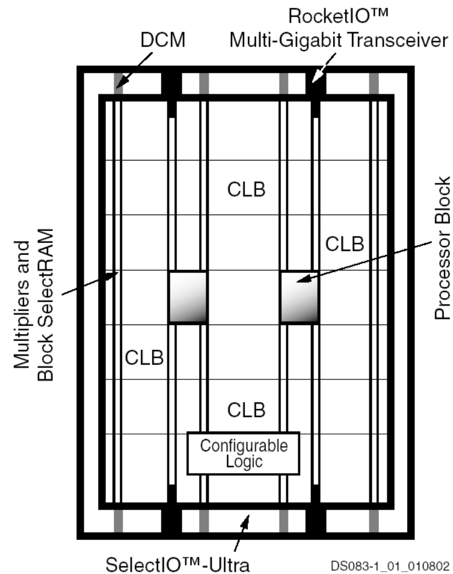

Figure 2.1: Xilinx FPGA chip (from Xilinx)

and the host computer the bottleneck, but in designs where the data is created and managed locally, FPGAs can yield very high speedups over equivalent software implementations.

In this thesis, we present an FPGA implementation of Murφ with the goal of accelerating the verification of specific hardware designs. We developed our approach on the example model of a counter, DOWN, which is provided with the Murφ verifier. Finally, we illustrate PHAST's capabilities with DASH, a cache coherency protocol, which is representative of protocols that are of interest to today's multicore processor designers.

2.3 Related Work

FPGAs have been used to accelerate formal verification as well as state space exploration. Graph traversal, which shares algorithm similarities with explicit state model checking, has seen great speedups when ported to an FPGA. An approach using A* search [13] achieves a 50x speedup over software. Earlier efforts to accelerate SAT solvers have been moderately successful, with speedups ranging from zero to one order of magnitude [11]. More recently, a SAT solver, implemented on an FPGA, saw a 70x speedup over state of the art software SAT solvers [14]. Efforts using hardware to accelerate pieces of the model checking algorithm and other formal verification techniques saw modest results. A 7x speedup has been achieved for a bounded model checker using SAT-based verification techniques [15]. Another approach uses Warshall's algorithm to create a model checking coprocessor [16].

State space exploration applications developed on GPUs have good results or small states, but none have managed to exhibit both characteristics. FPGAs are better suited to state space exploration with large state spaces and large states than GPUs for the following reasons. First, GPUs have a strict hierarchical memory organization that hinders the designers ability to manipulate state storage. Second, lack of communication across threads leads to GPUs frequently re-exploring pieces of the search space. Due to memory bandwidth and memory latency bottlenecks, state of the art model checking on GPUs currently see a 18x or less speedup [17]. An effort using GPUs and reversible hash functions with very small states achieves a

27x speedup over software [18]. This research used much smaller states that we are targetting with PHAST. Finally, an LTL model checker [19] cites data structures as the main bottleneck to achieving more than the 14x speedup seen on the GPU.

Efforts have been made to translate Murφ into HDLs automatically [20]. These efforts begin with useable Murφ and attempt to generate synthesizable VHDL. The resulting circuits are unlikely to fit onto FPGAs or generate speedup because of this top down approach, which results in very large HDL descriptions. The research presented in this thesis is unique because it uses a hardware accelerator to improve the end to end runtime of Murφ, an explicit state model checker, using models that exhibit real-world characteristics.

2.4 Conclusion

In this chapter, we presented some background and context for explicit state model checking and Murφ. We also introduced DASH and the use of FPGAs to accelerate breadth first search. Finally, we presented related work on accelerating model checking and breadth first search applications on both FPGAs and GPUs. In the next chapter, we cover the PHAST architecture and implementation details.

Chapter 3

PHAST

In this chapter, we present the PHAST implementation and architecture. PHAST is the FPGA implementation that accelerates Murφ, introduced in Section 2.1. To ease the understanding of PHAST, this chapter begins with a section that bridges the gap between the Murφ explicit state model checking software and the hardware implementation. The chapter continues by briefly touching on the modules that vary with the model under verification. Finally, the focus of this chapter is on those modules that provide the infrastructure of PHAST. The specifics of how models are handled and translated from the Murφ language are further addressed in Chapter 4.

3.1 Model Checking on an FPGA

As previously presented [3], using an FPGA to accelerate explicit state model checking provides two advantages. The first is the flexible memory hierarchy offered by the FPGA. This permits the intense memory management required for explicit state model checking. The next is the lack of communication required between the host and the FPGA. Communication is the most frequent and difficult bottleneck for ap-

plication acceleration on an FPGA and the reason other approaches to accelerating formal verification achieve smaller speedup. While models verified with PHAST are currently hand-coded, we plan to implement a generator in the future that will allow models specified in Murφ to be verified with PHAST.

Murφ starts with a start state, transition relations, and safety properties, provided by the user, and checks that the properties hold in all possible states. Several terms used in this thesis are specific to model checking or the Murφ verifier. The *start state* of a model is the first state or set of states that passes through the verifier. Murφ uses *rules* to specify the transition relations that generate a new state. *Unvisited states* are those that have been generated and are legal, but have not yet had the rules applied to them. *Legal states* are those that have not violated any invariants, or safety properties.

```
Add start state(s) to unvisited queue
While queue contains unvisited states
    remove unvisited state from top of queue
    generate all new states from current
    for each new state
        lookup in hash table
            if state has been visited before
                discard state
        check state against invariants
            if state violates a safety property
                stop application
                print trace
        add state to bottom of unvisited queue
Model is verified successfully
```

Figure 3.1: Algorithm Pseudo-code

The pseudo-code, shown in Figure 3.1, is a simplified version of the Murφ algorithm

using breadth-first search. In PHAST, the entire process is mapped to the FPGA, and both data structures, the hash table and the unvisited queue, are mapped to RAM. PHAST implements the checking of a particular model in hardware by directly implementing the rules. The rules, safety properties and the state state(s) are encoded in the model. To begin verification, only a signal to begin is necessary.

3.2 PHAST Architecture

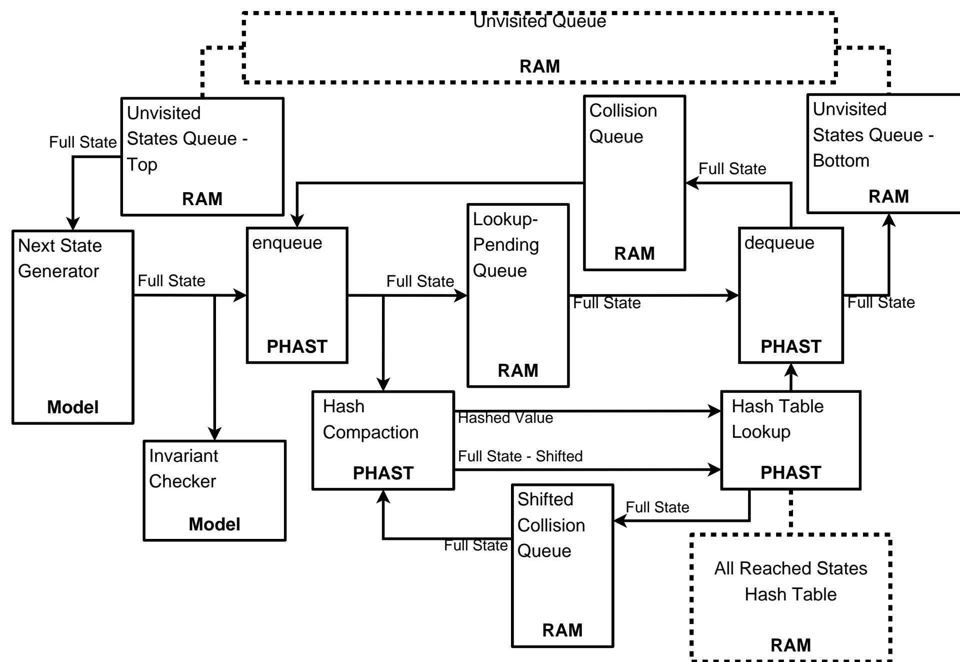

Figure 3.2: Hardware Block Diagram

Figure 3.2 shows a block diagram of PHAST. The boxes represent modules in the design. There are three types of modules in the PHAST design: Model, PHAST, and RAM. The modules labelled Model (Next State Generator and Invariant Checker) are derived from the model needing verification. The rest of the design, labelled PHAST

and RAM, implements the Murφ verifier independent of the model being verified. The Hash Table is labelled as RAM, but the location, on-board memory or on-chip memory, depends on the model. In PHAST, states are tracked through the use of the Lookup Pending Queue, the Collision Queue and the Shifted Collision Queue. A state exists in either the Next State Generator, the Lookup Pending Queue, the Collision Queue, or the Unvisited Queue and only in one of these locations. Hashed versions of the states exist in the Shifted Collision Queue or the Hash Table. Each module is described in more detail later in the chapter.

Figure 3.3, which describes concurrent processes implemented in hardware, shows the steps that PHAST takes to validate a model. A state is created in Step 1, the Next State Generator, and expires in Step 6, the Dequeue module. It is possible for a state to visit some modules multiple times in the case that a state's hash value collides in the Hash Table; however, the Unvisited Queue and the Next State Generator handle only unvisited states, assuming that the Hash Table Lookup Module successfully removes all duplicate states. In the case that a duplicate is not caught by the Hash Table, PHAST will revisit states. Duplicate states can arise because PHAST revisits parts of the graph or because an unvisited state generates a previously visited state. Collision and duplicate states are discussed in Section 3.2.3.

The RAM modules in the diagram represent queues. On-chip RAM buffers states, hiding latencies in reading states from the board's RAM. In Figure 3.1, we refer to a queue with unvisited states, and in the block diagram, there is a top and bottom for

```
Process 1: Next State Generator
    read Current State from Unvisited Queue
    generate Next State
        write Next State to Enqueue if valid
        else discard Next State
Process 2: Enqueue
    if Next State available
        read State
    else read State from Collision Queue
    write State to Lookup Pending Queue
    write State to Invariant Checker
    write State to Hash Compaction
        if State is Collision State
            mark State as Collision
Process 3: Invariant Checker
    check State against safety properties
    if State violates properties
        stop verification
Process 4: Hash Compaction
    read State from Enqueue
    if State is Collision
        read State from Shifted Queue
        compute Hash Value from Shifted State
    else generate Hash Value
    write Hash Value to Hash Table Lookup
Process 5: Hash Table Lookup
    read Hash Value from Hash Compaction
    check address from Hash Value
    if address is empty
        write Hash Value to Hash Table
        mark State as new for Dequeue
    if data at address matches Hash Value
        discard Hash Value
        mark State as duplicate for Dequeue
    if data does not match Hash Value
        write Shifted State to Shifted Queue
        mark State as collision for Dequeue
Process 6: Dequeue
    read State from Lookup Pending Queue
    wait for status from Hash Table Lookup
    if new, write to Unvisited Queue
    if duplicate, discard
    if collision, write to Collision Queue
```

Figure 3.3: Parallel Processes Implemented in PHAST

that unvisited queue. This top and bottom are buffers for the whole unvisited queue that resides in the RAM on the board. All new states, including start states, enter the design from the Next State Generator and are stored in the Lookup-Pending Queue while states are hashed and the hashed values are looked up in the Hash Table. When the results of the lookup are known, the state is moved out of the look-up pending queue into either the collision queue, the unvisited states queue or deleted if it has already been visited. Collision handling is described in Section 3.2.3.

Several differences exist between the Murφ and PHAST implementations of explicit state model checking. Because PHAST is a hardware interpretation of Murφ, PHAST takes advantage of concurrency in many places. The most obvious difference is when the state is checked against the invariants. In Murφ a state is checked against the invariants after it has been found to be unique, so that the fewest number of states need be examined. In PHAST, states are checked when they are generated. As PHAST uses concurrency to get the best hardware performance, a state is checked for all invariants in the same cycle. Thus, each cycle a new state can be checked, and moving the invariant checker earlier in the verification process does not slow PHAST down and allows verification to halt immediately after finding a state that violates a safety property. Another difference is in the manner that Hash Compaction is implemented, for efficiency, in hardware. Where Murφ performs a read and xor for each row of the hash matrix, PHAST computes the bits of the hash value concurrently. A final difference between Murφ and PHAST is the ordering of states. In Murφ states

are kept in a strict order, whereas PHAST reorders states in order to process a state every cycle. States may be processed much later in PHAST than they would be in Murφ because of collisions in the Hash Table.

3.2.1 Next State Generator and Invariant Checker

The two modules that change completely with each model that PHAST verifies are the Next State Generator and the Invariant Checker. The Next State Generator is the implementation of the transition relations for the model. All of the new states, including the start states, begin in this module. The only input to the Next State Generator is a start or reset signal. All transition relations, including the start state, are encoded in the module. The Next State Generator applies one rule to a Current State every cycle, and compares the generated state against the Current State. While the Next State Generator can generate a state nearly every clock cycle, some cycles are empty because the generated state is a replica of the Current State or because the Next State Generator is busy getting a Current State from the Unvisited Queue. The Next State Generator outputs new unvisited states.

The Invariant Checker encodes the safety properties or invariants for the model. This module checks every new state immediately after it is generated. The Invariant Checker implements each invariant independently, so a state can be checked for all properties concurrently. This module can process a state every cycle. It also stops verification as soon as a violation is discovered. The only output from this module is the stop signal. That signal is only asserted in the case that a state violates one

of the safety properties.

In Murφ a state is checked against the invariants after the hash table lookup. In PHAST the state is checked against the invariants immediately after creation, in parallel with hash compaction. The invariant checker processes more states than in the Murφ verifier, but since the checker can process states at the same rate that they are generated, this does not introduce any delay.

3.2.2 Hash Compaction

The Hash Compaction module takes a full sized state as an input and outputs a hash value. The module is based on the hash compaction used in Murφ [7] that was developed at Stanford as an improvement over the "hashcompact" method of Wolper and Leroy [21]. The hash compaction method required changes from Murφ when implemented in hardware. When performing hash compaction, each bit that is set in the state necessitates a read of a hash matrix. The hash matrix is a list of random values that is as long as the number of bits in the state and as wide as the width of the hash values. The bits of the matrix are randomly set when a new verification model is created for Murφ. Data is read out of the hash matrix at an address that corresponds to the one bits that are set in the state. These values are XORed together to generate the final hash value. Figure 3.4 demonstrates hash compaction with an 8 bit state and a 5 bit hash value. The values in the hash matrix and state are randomly set for the purpose of demonstration.

In order to achieve the best possible performance for PHAST, we did not imple-

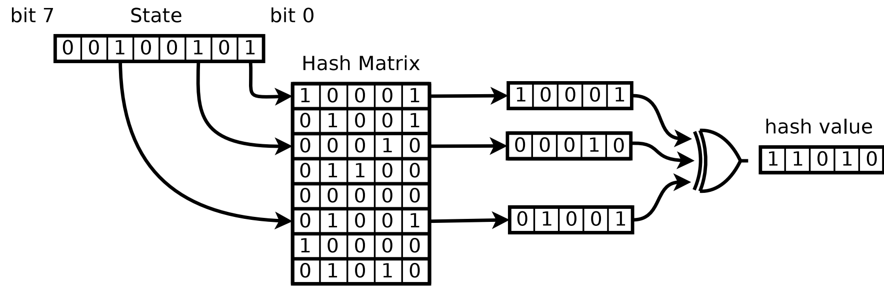

Figure 3.4: Hash Compaction

ment the hash compaction as shown in Figure 3.4. The number of cycles needed to compute the hash value would depend on how many bits were set in the state. For long states, this meant the hash compaction could possibly take hundreds of cycles. Not knowing exactly how many cycles it would take to compute a hash value also means the hash compaction could not easily be pipelined.

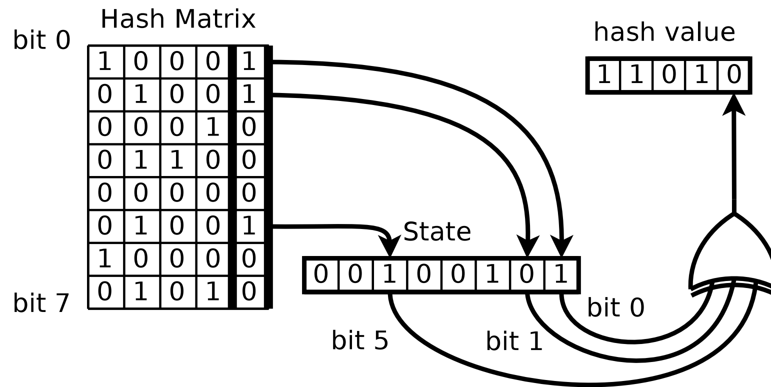

Figure 3.5: XOR Tree Hash Compaction

In PHAST, we implemented the hash compaction as an XOR tree for each bit of the output hash value. As with the Murφ hash matrix, the bits of state that are XORed together are chosen randomly for each new model verified by PHAST. Figure 3.5 has the same hash matrix and state as Figure 3.4. In Murφ, the hash compaction

is computed from sequential reads of the hash matrix, performing the running xor computation after ever read. PHAST computes all bits of the hash value in parallel. For bit 0 of the hash matrix, the xor tree is derived from column 0 in the hash matrix. The bits in the column that are set represent the bits in the state that would affect that bit of the hash value. Those bits in the state that are selected by the hash matrix column 0 are bitwise XORed together to form bit 0 in the hash value. The columns in the hash matrix have a one to one correspondence with the bits in the hash value. The computation can be pipelined by inserting registers between the levels of the xor trees. For small states with only one level of xor computation, as in Figure 3.5, the hash value can be computed in one clock cycle by computing all the bits in parallel.

3.2.3 Hash Table Lookup

The purpose of this module is to determine whether a state should be written to the working queue, removed as a duplicate, or reintroduced as a collided state. To that end, the Hash Table Lookup Module takes in a hash value, splits it into an address and tag, then asserts one of these signals: duplicate, unique or collision. Both PHAST and Murφ use a large table to store generated hash tags. In Murφ, the table is stored in main memory and lookup involves a simple comparison between the Hash Table data structure and the newly created hash value. For PHAST, the table and the lookup present some design challenges. For anything but very small models, the Hash Table can not fit in on-chip memory. Using external memory requires each

lookup to generate a read and possibly a follow-up write to the board memory. The addresses generated in the Hash Compaction Module, discussed in Section 3.2.2, was calculated by Murφ to have less than 0.000001% probability of omitting even one state. For a state size of 1500 bits, this module, and the accompaning probability of an omission, remains the same no matter which model is verified by PHAST. These probabilities are reported by Murφ, when verifying a model.

Since the latency of a read or write to SDRAM is long, the Hash Table Lookup hardware implementation uses a few techniques to overlap the memory transactions as well as remove duplicates to reduce the need to access memory. Since many of the same states are generated close together in the exploration, a Content Addressable Memory (CAM) is used to eliminate duplicates.

Duplicate Removal

The implementation of the Hash Table Lookup Module is partially dependent upon the specifications of the memory interface for the FPGA board and the characteristics of the state generation. The memory interface queues a number of read and write accesses to the memory banks. As the addresses for the transactions to memory are not sequential, the amount of time for a read or write to complete is variable from one transaction to the next. Naive implementation relied upon transactions being initiated on an as-need basis. The variable latency of a transaction and the lack of regulation for initiating transactions resulted in unpredictable performance that altered with small changes in details of the memory interface to the Hash Table

Lookup. Part of this stems from the state generation characteristics. Frequently, duplicate states are generated within a few cycles of each other, resulting in the Hash Table reporting that all states are not present in the Table. The use of the CAM allows the consistent removal of duplicate states, resulting in both better and more predictable performance.

In contrast to standard memory which returns content at a certain address, a CAM takes content and returns whether the content already exists in the memory as well as the address or addresses where that content is located [22]. Depending upon the implementation, if a duplicate is present, a CAM might only return the address of the first or last location where the content is stored, or the information that the data is in memory. Our implementation of the Hash Table Lookup Module needs to know of the presence of a state in the CAM and does not care about the possible location. To this end, the CAM used in PHAST only report if a match exists.

The number of states stored in the CAM is a balance between the need to cover the memory latency and to budget the FPGA chip resources. Duplicate states can result in thousands of extra duplicate states being created and can diminish any speedup that would be gained from the use of PHAST instead of Murφ. However, a very large CAM would essentially be a Hash Table stored on the FPGA chip. To this end, the size of the CAM was tuned until it eliminated duplicates that were incurred during a memory transaction without becoming so large as to consume too many FPGA resources. After investigating CAM sizes ranging from 4 entries to 128

entries, we chose a 32 entry CAM.

Content Addressable Memory improves state lookup in 2 ways: it removes duplicates as they arrive in the module, and it spaces out the transactions to memory by a consistent number of cycles. As the Hash Compaction Module computes hashes, they are queued up in the Hash Table Lookup. During the CAM phase, each state is checked against the contents of the CAM. The CAM begins empty, but each state that does not hit in the CAM is immediately written to it. States not in the CAM initiate a read request to memory. If a read request is needed, additional states are checked against the CAM while the Hash Table Lookup Module waits on the results of the read for the first state. The CAM and memory reads continue until the read result for the first state comes back. However, an unknown number of duplicate states may be processed during this CAM phase before that first memory transaction returns.

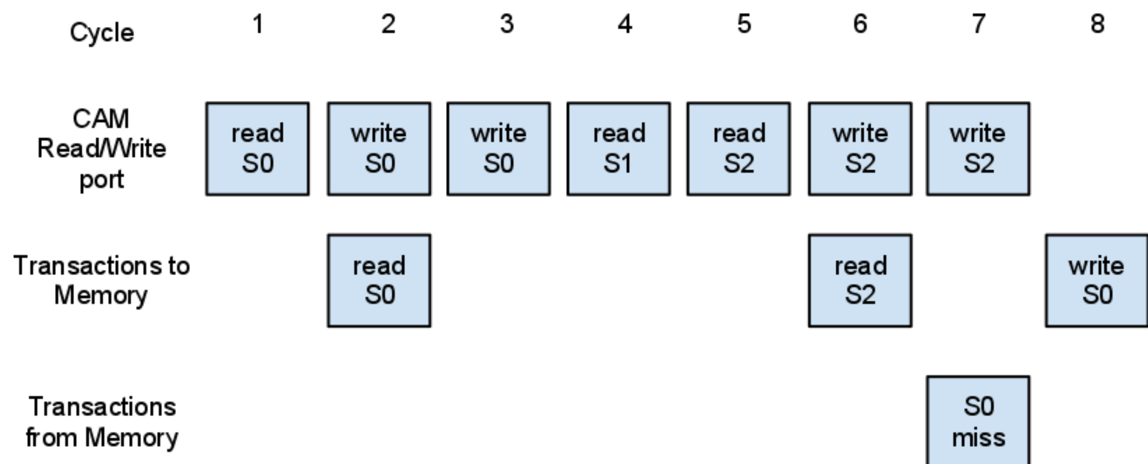

Figure 3.6: CAM waveform

Figure 3.6 demonstrates the CAM phase in more detail. In this example, only three states are shown for illustrative purposes. The first state, State 0 (S0), demonstrates a new unvisited state. Thus the state does not exist in the CAM or in the Hash Table, and so S0 misses in the CAM, which causes a write to the CAM. This write takes 2 cycles. S0 also misses in the Hash Table, which originates a write to the Hash Table. State 1 (S1) demonstrates a duplicate state that was originally visited sometime in the previous 32 unvisited states seen by the CAM. S1 hits in the CAM and was marked for deletion without necessitating a lookup to the Hash Table. By hitting in the CAM, the CAM is ready for a new state in the next cycle. In that next cycle S1 is also marked for deletion without impeding either the CAM or the Hash Table lookups. S2 has not been seen in the previous 32 unvisited states and thus does not exist in the CAM and is not known to be either a new or duplicate state without results from a Hash Table lookup. S2 misses in the CAM in cycle 5. In Figure 3.6, the result from memory for S0 occurs after the lookup for S2 is sent. In this example, no other lookup request will be sent to the Hash Table until the results for S2 come back. This demarcation between the sending of the read requests and the waiting for read results is an arbitrary transition point between reading the memory and writing to the memory that allows a controlled number of transactions to occur in sequence that limits memory thrashing while maximizing work to cover the memory latency. At that time the writes for S0 and possibly S2 will be written to the Hash Table and more states will be processed by the CAM.

The Hash Table Lookup Module is never idle as results are constantly incoming from the Hash Compaction Module and from the Hash Table. The CAM always has states available for processing. The write portion of the Hash Table Lookup Module, and the commit, delete or collision results for each state are sent to manage the working queue once the first state's lookup result comes back.

Collisions

Collisions occur when two different states hash to the same location in the hash table. Collisions are a common problem when hashing. When trying to compress the state space, we exacerbate the problem of collisions because we are compressing a very large state description into a relatively small location in the hash table. We can detect when a collision has occurred by checking the tag at the address specified in the hashed state. The tag consists of a valid bit and state bits. If the valid bit is set, and the tag from the table does not match the tag from the state we are looking up, then a collision has occurred. We try to reduce the number of collisions by using an effective hash function and keeping the number of entries in the hash table large.

In the Murφ algorithm, collision handling is part of the lookup. In PHAST, collisions had to be integrated into the pipeline. Collisions are detected in the Hash Table Lookup, as they are in Murφ, but are not resolved immediately. Resolution happens when the address in the Hash Table for a collision state is empty and the Hash Value can be stored at that address. To handle modified collisions in PHAST we created two Collision Queues, one for original states and one which stores modified

versions of the full states. These two queues are important because hashed values for collision states are generated from a shifted version of the original state instead of the original itself. The original state is kept to reinsert into the Lookup Pending Queue. When a new lookup into the Hash Table is performed on the rehashed state, the results will match the state in the Lookup Pending Queue that had the original collision.

```
Hash Table Lookup reads Hash Value address
if Hash Table contains no data at address
    Hash Value data written to Hash Table
    Dequeue reads State from Lookup Pending Queue
    Dequeue writes State to Unvisited Queue
if Hash Table has data
    if data matches Hash Value data
        Dequeue reads State
    if data does not match Hash Value data
        Hash Table Lookup shifts State
        Hash Table Lookup writes Shifted State
        Dequeue reads State
        Dequeue writes State to Collision Queue
```

Figure 3.7: Hash Table Lookup and Collision Detection

```
Hash Compaction recieves State from Enqueue
if Collision bit is set
    Shifted State read from Queue
    Hash Compaction performed on Shifted State
    Shifted State sent to Hash Table Lookup
else
    Hash Compaction performed on State
    State sent to Hash Table Lookup
```

Figure 3.8: Collision Reintroduction

The Hash Table Lookup module handles collision states by shifting the state by

one bit. Then the Hash Table Lookup module and the dequeue module write the shifted state and full state into the appropriate collision queues. Figure 3.7 outlines this detection. To see how the state gets inserted back into the design, we have to go back to the enqueue module, which handles writing new states into the Lookup-Pending Queue and sending those states to the Hash Compaction module. The enqueue module waits for a new state from either the Next State Generator or the Collision Queue. When the state is from the Collision Queue, the only difference from a new state is a signal sent to the Hash Compaction that the state is a collision. When this signal is set, the Hash Compaction module reads the shifted state out of the Shifted Collision Queue and performs the same hash function on the shifted version of the state. The original state is sent to the Lookup-Pending Queue just as new states are. Figure 3.8 provides pseudo-code for the reintroduction of the Collision State in PHAST.

3.3 Conclusions

This chapter discussed the model checking algorithm modified from Murφ and implemented in the architecture of PHAST. The model and infrastructure specific modules are separated in order to accommodate different models easily. We also presented implementation details that were both changed or ported from Murφ. In the next chapter, we discuss in more detail the models used to test and explore PHAST's architecture.

Chapter 4

Murφ Models

For the purposes of displaying PHASTs abilities, we chose two Murφ models, DOWN and DASH. PHAST was developed using the smaller DOWN model, then expanded to fit DASH, which was based upon a real cache coherency protocol as discussed in Section 2.1.1. All references to DASH in this chapter refer to the elementary abstracted model of DASH provided in Murφ and not the actual DASH multiprocessor.

4.1 DOWN

For this project, we chose DOWN as an example for verification. DOWN is an array of six counters that are decremented in a semi-connected manner. Counters one and two are decremented together, as are two and three, three and four, and four and five. Counters five and six are decremented independently. When all six variables have reached zero, the safety property that the sum of all counters should be greater than zero is violated.

In Murφ, the DOWN state is represented as an array of six variables. The ruleset checks if the variable is greater than zero and if so, decrements the variable and

possibly decrements the next higher adjacent variable, if that portion of the rule applies. Murφ uses rulesets when the same rule can be applied with different inputs. The ruleset for DOWN is the same for all six variables, so the rule does not have to be replicated six times. Rulesets can be thought of as loops. In the case of DOWN, the ruleset is a for loop, where the rule is applied to all six variables in sequence. The representation of DOWN in Murφ includes the array of variables, the range of values those variables can take, the ruleset and the invariant for the model. The Murφ model for DOWN can be downloaded with the Murφ verifier [23].

In PHAST, the DOWN state is built into the verifier. Because the variables are initialized to five and are decremented to zero, the state is eighteen bits wide, three bits for each variable. The Next State Generator has the same rule replicated four times with slight changes for rules five and six. The Invariant Checker only has one property to check against the states: that the sum of the variables is greater than zero. The largest number of bits set in any state is twelve. The longest number of cycles it would take a state to be compressed and looked up in the hash table is five, ignoring extra cycles between modules and due to control signals.

DOWN is a relatively small example and fits completely on the FPGA we are targeting. The final number of states generated for DOWN by Murφ is over ten thousand. The published results in [3] did not use actual hardware and the Hash Table and the Unvisited Queue were implemented using block RAM. The second implementation of PHAST ran on our Virtex 5 chip. The latest implementation of

PHAST uses the on-board memory for the Hash Table and Unvisited Queue to match the implementation of large models. For the Hash Table Lookup, the change from one cycle reads into the block RAM on the chip to the many cycle reads to the RAM on the board resulted in two changes in the number of duplicates and collisions. The use of the on-board RAM necessitated the implementation of the CAM and other changes to the architecture to reduce the large increase of duplicates. However, with the Hash Table spanning an entire 256 MB bank of the on-board RAM, the number of collisions dropped to 0 for DOWN.

4.2 DASH

DASH, Directory Architecture for Shared Memory, is a scalable machine that maintains a single address space with coherent caches. The model for this multiprocessor is a network of multiple nodes. Each node contains a small number of processors with private caches, a portion of the shared memory, a shared cache, and a directory controller which interfaces that node to the rest of the network. Because the memory is distributed over all the nodes, each node is responsible for maintaining only the portion of the address space that resides within its shared memory. The directory controller maintains pointers to every other node that has addresses from this node's shared memory in the other node's cache. The DASH multiprocessor model connects all nodes with a point to point interconnect so that every node can exchange messages directly with any other node. The interconnect has two dedicated fabrics, one

for requests and one for replies. The limiting of messages to only those nodes that have or maintain a specific address helps minimize the amount of communication necessary to maintain coherence.

```
Rule "handle read request to home"
  ReqChan.Count > 0
  & Request = RD_H
==>
Begin
Alias
  RAC : Procs[Dst].RAC[Dst][Addr];
  Cache : Procs[Dst].Cache[Dst][Addr];
  Dir : Homes[Dst].Dir[Addr];
  Mem : Homes[Dst].Mem[Addr]
Do
  Switch RAC.State
  Case WINV:
    -- cannot release copy
    Send_NAK(Src, Dst, Dst, Addr);
    Consume_Request(Src, Dst);
  Else
 [...] -- other cases in rule
  End; --switch;
End; -- alias: RAC, Cache, Dir, Mem
End; -- rule
```

Figure 4.1: Murφ Rule

The DASH state represents all processing nodes, each containing the private cache for each processor in the node and the remote access cache for each node, the reply and request fabrics for the interconnect, and the home node with the main memory and directory. The Murφ model contains safety properties and rules that describe how these caches can change, which fabrics get related messages, and whether other caches or cache states change as a consequence. In the model, there are three remote

```
read_req_to_home_01 : process(regen,procs_par,homes_par,rep_par,req_par,
                              rep01,req01,home01,proc01,aux0,aux1,regstop)
--reqnet[0][1].count > 0 && reqnet[0][1].mess[0].mtype = RD_H
begin  --src = 0 (Home), dst = 1 (Remote1)
    if regen = '1' then
        case procs_par(1)(6 downto 4) is
        when "100" => --case 4 is WINV, cannot release copy
            send_nak(rep_par(4),aux1,regstop,rep01(4));
            req01(1) <= consume_request(req_par(1));
            home01 <= homes_par;
           [...] -- assign all other state bits to parent state values
            same <= '0';
        when others =>
       [...] -- other cases in "Read Request to Home" Rule
        end case;
    else
        regstop <= '0';
       [...] --all bits in state are 0, not valid state
        same <= '1';
    end if;
end process read_req_to_home_01;
```

Figure 4.2: Murφ Rule translated to VHDL

processing nodes, one home node, and one home directory. Because the interconnect for DASH is point to point and this model has four nodes, there are 16 connections each for the reply and request fabric respectively. Most rulesets in the DASH model have guards that ignore connections from a node to itself; however, since not all rulesets have this guard, all 16 connections are counted.

The DASH model generates over 90,000 states at approximately 1500 bits per state and close to 100,000 state transitions. The unvisited queue, at its longest point, contains close to 8,000 states. The model contains 17 rulesets that result in 220 individual rules. Our model of the DASH protocol is similar in size and complexity to models Intel uses to validate features of modern processors: state sizes between 1200 and 1800 bits and a transition relation with more than 100 rules [24].

One of the smaller and simpler examples of a DASH ruleset, as encoded in Murφ, is provided in Figure 4.1. This rule is replicated for every combination of processing node. As an example, this rule is created for the connection from Remote Node 0 to Remote Node 1 as well as Remote Node 0 to Remote Node 3. In this example, a RAC is the acronym for Remote Access Cache, a NAK is a negative acknowledgement, and WINV is a RAC state that is waiting for an invalidate acknowledgement. The code shows that the Remote Access Cache, in the case that it is waiting for an invalidate acknowledgement, sends a NAK to Remote Node 1 and consumes, or deletes, the request from Remote Node 1 that originated the NAK.

4.2.1 Implementation on PHAST

The DASH model was chosen because the state size and number of rules is large enough to demonstrate PHAST's capabilities. The framework has been tested with DASH's transition relations.

Figure 4.2 shows a simplified example of one of the DASH rules. The VHDL for a ruleset is, on average, 40x more code than the Murφ model. The large increase in code does not result in more complicated logic on the FPGA. Murφ takes advantage of looping the rules to reduce the 220 rules to 17 rulesets, but these 220 rules are actually completely independent of one another. While DASH's transition relations and safety properties are implemented and DASH is performing similarly to DOWN, some techniques for reducing and managing the increase might be necessary for the automatic translation of Murφ models to PHAST.

The rule in Figure 4.2, which handles read requests to the home node, uses supporting procedures that have been translated to VHDL from Murφ. The rule is typical of many of the other 219 rule implementations in a few ways. First, most of the circuitry on the FPGA chip is used for simple logic circuits. The DASH rules create a next state by checking a value in the parent state and assigning a new value. Most DASH rules change only a small number of bits in the entire state; usually at least one branch of the code results in the next state being exactly the same as the parent state. PHAST generates an extra bit with the state that designates whether the state has had bits changed from the parent. In Figure 4.2, this is the 'same'

bit that is set to ‘0’ when a bit has been changed in the next state and ‘1’ when the parent state is copied to the next state. This bit is used to prevent a parent state from generating multiple next states that are duplicates. Another way in which Figure 4.2 is typical of the other 219 is that many of the rules are almost identical to each other. This particular rule is identical in behaviour to 15 other rules that vary only in the source and destination designations.

Nearly all of the rules rely upon a small set of auxiliary procedures to reduce the amount of work performed within the rule itself. In PHAST, the Next State Generator has all 220 rules explicitly defined along with all the auxiliary procedures, which are coded as VHDL functions or procedures. In addition, although not shown in Figure 4.2, many of the the rules for the DASH model have some assertions encoded directly into the rules, so the Next State Generator as well as The Invariant Checker for the DASH verifier has the ability to stop the verification.

In PHAST, the DASH state is built into the verifier. Because both the reply and request messages fabrics are point to point and can store up to four messages from one node to any other node, the reply and request fabrics need a large number of bits. The data representing the four processing nodes is 48 bits and the home node is an additional 15 bits. State management for DASH creates its own challenges, independent of the rest of the circuit. These challenges stem from the size of the state. In the Next State generator, PHAST initially used comparators to detect unique newly generated states. The DASH state, at 1500 bits, required an alternative

solution to avoid using that much hardware. Any implementations, such as the comparators, that required an operation be applied to an entire state were avoided. Other examples of challenges arising from the size of the state include storing the state in registers and fanout from the parent state to all 220 rules in the Next State Generator. To reduce fanout, PHAST replicates the parent state. Replication of the state even once consumes over 3000 registers, from the parent and next state, in the Next State Generator alone. Tradeoffs like these arise in almost every module in PHAST and have resulted in modules that are capable of handling the verification of real protocols with large states.

4.3 Experimental Results

This work compares two implementations of explicit state model checking: Murφ and PHAST. Murφ was run on the host machine: an x86 machine with two Clovertown processors running at 1.86 GHz and 8 GB of RAM. PHAST targets the ADM-XRC-5LX, an FPGA board from Alpha Data Inc. The board contains a Xilinx Virtex chip, 1 GB of DDR-II SDRAM and 16MB of Flash RAM, and connects to the host through a 64-bit 133MHz PCI-X interface connection. The Virtex chip on the board is a Xilinx Virtex 5 LX110. This chip contains 128 32Kb block RAMs and 17,280 logic slices. Two models were used for the comparison. Both ship with Murφ. The first is DOWN, and the second is DASH. DOWN is used to demonstrate the speedup that PHAST can provide while DASH demonstrates the complexity that PHAST can

handle.

4.3.1 DOWN

DOWN, a small example of a counter, was run with Murφ and PHAST. Murφ verifies DOWN in 100 ms using one core. The final number of states generated for DOWN by Murφ is over ten thousand. In [3], PHAST achieved a 200x speedup with a faster clock in simulation. This implementation did not use the memory interface as the Hash Table and the Unvisited Queue were implemented in block RAM on the Virtex chip. The one cycle reads to the block RAM also resulted in no duplicates. This PHAST implementation of the DOWN verification required less than 1% of the slices available on the Virtex chip and only 6 of the block RAMs. On the actual Virtex chip, the design ran at 50x faster than Murφ.

For the third implementation, the Hash Table and the Unvisited Queue were moved to the on-board RAM. While the first two implementations reported no duplicates and only a slight increase over the number of states generated by Murφ, the best case implementation using the on-board RAM generated forty seven thousand states and thirty six thousand duplicate states. The current implementation of PHAST uses a CAM to decrease the number of generated states to twenty six thousand and the number of duplicates to twenty thousand. PHAST now runs at 158 MHz and achieves a 30x speedup over Murφ.

While the use of the board's RAM did impact performance, the optimistic results originally given by simulation are typical of FPGA implementations. Simulation

results are typically faster because in real implementations, clock speed is limited by memory interfaces and real world hardware concerns.

4.3.2 DASH

Murφ verifies DASH in 15s using one core. Murφ stores nearly 7800 states in the Unvisited Queue at its deepest point. For the hardware implementation, this translates into less than 2 MB of space for the Unvisited Queue, which, while too big for on-chip block RAM, fits easily into one of the four banks of on-board SDRAM. Each bank is 256 MB. The Hash Table benefits from a large size, and thus was designed to span an entire bank of SDRAM, but only contains 0.5 MB of data when the entire state space is represented. The DASH model, with close to one hundred thousand states and 220 rules, takes up less than forty percent on the Virtex chip and less than thirty percent of the block RAMs. Hash Compaction, for example, when converted for DASH, results in xor trees with up to 4 levels and hundreds of xor operations. The block RAMs are consumed with caches to cover the latency from reading and writing to the on-board RAM, the Lookup Pending Queue and Collisions Queues.

The DASH transition relations and invariants have been translated into VHDL and implemented in the Next State Generator and Invariant Checker. Synthesis of the DASH model, alongside PHAST, in VHDL reports 172 MHz clock speed. Because the clock speed for PHAST is determined by the Hash Table Lookup Module, and not the Next State Generator, this increase in the clock speed does not indicate any additional improvements over the speedup demonstrated by the verification of

DOWN with PHAST. The speedup given by the verification of DOWN with PHAST holds true for DASH, despite the greater utilization of area on the Virtex 5 chip from the increase in transition relations, safety properties and block RAM. Currently, DASH has also not been tested in simulation or on real hardware.

4.4 Conclusion

In this chapter, we detailed the models used as proof of concept for PHAST. The first, DOWN, was used to develop the PHAST infrastructure. The second, DASH, showed that PHAST could handle models with real state and state space sizes. Both models were translated by hand from Murφ to VHDL and integrated into PHAST. In the next chapter, we conclude this thesis and present future work.

Chapter 5

Future Work and Conclusions

The largest need for this work going forward would be to automate the translation of the transition relations and invariants from Murφ to VHDL. In this chapter, we discuss this effort based upon the previous work with DASH and present other future work.

5.1 Automatic Translation from Murφ to VHDL

Currently any model we implement in FPGA hardware is hand-coded in VHDL, which limits the usability of the PHAST verifier. Thus, after PHAST has been tested on several examples, it will be expanded with a language compiler that will translate models in the Murφ language to the VHDL for that model. To assist in the effort, we have kept the hardware as generic as possible. PHAST as implemented for DOWN has some areas that would benefit from parallelization, but this parallelization would be applied to this model only. The model specific components, The Next State Generator and The Invariant Checker, have been kept separate from the constructs needed to implement the Murφ algorithm. The language parser may require

revisiting the hardware design, code parameterization, and some redesign. Hand-coding the Murφ models can take months. Once the PHAST language compiler has been created, the speedup compared to Murφ will take into consideration the amount of time required for synthesis and implementation of a model with PHAST.

The most important part of translating from Murφ to VHDL is in the design of the state. Each model is based around a state which encodes the design under verification. Poorly designed states can introduce difficulty into the transition relations. With Murφ, these state are described as objects. In VHDL these objects need to be further encoded into a string of bits. With DASH, the state was changed as the transition relations were translated. The first direct implementation had to be changed to account for the use of '0' values and undefined values. These settings in Murφ have different meaning than in VHDL. The state was again changed to have certain bit ranges aligned instead of striped across the state. These values were used together over and over in the transition relations, but the relationship was not inherent in the state description given by the Murφ model. While not all models might run into these difficulties, checking for certain characteristics in the model might greatly benefit the VHDL design.

After the design of the state is set, the transition relations themselves are fairly straightforward. Each one needs to be checked for assertions that might cause verification to quit. Each relation otherwise checks certain bits in the parent state, then depending on the values present, might reassign those or other bits to some new

value. The most difficulty with regards to transition relations came about because of the sheer number of cases present in any single rule.

5.2 Future Work

PHAST is based on the Murφ verifier which has been successfully used in industry for over ten years. During that time, much research has been done in the field of model checking. We intend to incorporate the current state of the art, such as symmetry reduction into the PHAST verifier. The advantages of symmetry reduction should be the same in software and hardware implementations of Murφ.

A possible expansion of PHAST's capabilities is to incorporate run-time reconfiguration into the verifier. Since PHAST is partitioned to separate the model from the verifier, we intend to investigate using run-time reconfiguration to change the hardware from checking one model to checking another, making PHAST more flexible.

While the implementation of PHAST focuses on DASH and DOWN, which are Murφ models, PHAST's structure is not limited to explicit state verification or a reimplementation of Murφ. PHAST explores a large state space to check for the existence of specific properties. PHAST uses transition relations to generate the states in this space. Any application that fits this structure can use PHAST's framework.

5.3 Conclusion

PHAST, a pipelined hardware accelerated state checker, succeeded in accelerating the execution time of Murφ, a software model checker. PHAST achieved over 30x

speedup in running hardware through a custom design and by taking advantage of FPGA characteristics. We expect to continue to see consistent speedups over the software implementations of explicit-state model checkers based on the characteristics of circuitry created for the DASH model.

The DASH model has a relatively small number of rules and invariants, yet takes up a fair amount of logic on the FPGA. The final amount of logic used with the DASH model utilizes nearly fifty percent of the Virtex 5's LUTs. Considering the capabilities of newer Virtex chips, models containing more transition relations might need to be ported to a newer chip.

Glossary

ASIC Application-Specific Integrated Circuit.

DASH Directory Architecture for Shared Memory; A multiprocessor, cache coherency protocol, and a model verified by Murφ and PHAST.

DSP Digital Signal Processing.

FPGA Field Programmable Gate Array.

GPU Graphics Processing Unit.

Invariants see Safety Properties.

Legal States States that have not violated any safety properties.

LTL Linear Temporal Logic; also a class of model checker that uses temporal logic to specify safety and liveness properties.

LUT LookUp Table.

Murφ an explicit-state model checker developed at Stanford University.

PHAST Pipelined Hardware Accelerated STate checker.

Reachable States All states that a model can transition to from a given start state or set of start states.

Rules see Transition Relations.

Safety Properties Referred to in Murφ as invariants. These properties specify what should never happen in any state in a model's reachability graph.

SAT SATisfiability. The problem of determining if a given Boolean formula can be assigned in a way to make teh formula evaluate to true. Also a class of algorithms that can solve large subsets of SAT instances for the purpose of verification.

Start State The state or set of states that is encoded with a model for verification.

Transition Relations The set of functions that defines the connections between the states in the reachable graph.

Unvisited States States that have been generated, and are legal, but have not yet had the rules applied.

Bibliography

[1] "International Technology Roadmap for Semiconductors," http://www.itrs.net/, 2006.

[2] B. Bentley, "Validating a modern microprocessor," 2005, see URL `http://-www.cav2005.inf.ed.ac.uk/bentley_CAV_07_08_2005.ppt`.

[3] M. E. Fuess, M. Leeser, and T. Leonard, "An FPGA Implementation of Explicit-State Model Checking," in *FCCM '08: Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2008, pp. 119–126.

[4] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.

[5] P. Manolios and R. Trefler, "A lattice-theoretic approach to safety and liveness," in *Twenty-Second ACM Symposium on Principles of Distributed Computing (PODC 2003)*. ACM, 2003.

[6] C. N. Ip and D. L. Dill, "Better Verification Through Symmetry," *Formal Methods System Design*, vol. 9, no. 1-2, pp. 41–75, 1996.

[7] U. Stern and D.L. Dill, "Improved Probabilistic Verification by Hash Compaction," in *Correct Hardware Design and Verification Methods*, P.E. Camurati and H. Eveking, Eds., vol. 987. Stanford University, USA: Springer-Verlag, 1995, pp. 206–224.

[8] U. Costa, S. Campos, N. Vieira, and D. Deharbe, "Explicit-Symbolic Modelling for Formal Verification," *Electronic Notes in Theoretical Computer Science*, vol. 130, pp. 301–321, May 2005.

[9] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu, "Bounded Model Checking," 2003. [Online]. Available: http://www.cs.cmu.edu/~emc/papers.htm

[10] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor," in

ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture, 1990, pp. 148–159.

[11] I. Skliarova and A. de Brito Ferrari, "Reconfigurable Hardware SAT Solvers: A Survey of Systems," *IEEE Trans. Comput.*, vol. 53, no. 11, pp. 1449–1461, 2004.

[12] J. Davis, Z. Tan, F. Yu, and L. Zhang, "A Practical Reconfigurable Hardware Accelerator for Boolean Satisfiability Solvers," in *45th ACM/IEEE Design Automation Conference*, June 2008, pp. 780 –785.

[13] Z. K. Baker and M. Gokhale, "On the Acceleration of Shortest Path Calculations in Transportation Networks," in *FCCM '07: Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 23–34.

[14] M. Safar, M. El-Kharashi, M. Shalan, and A. Salem, "A reconfigurable, pipelined, conflict directed jumping search sat solver," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2011, pp. 1 –6.

[15] H. Yoshida, S. Morishita, and M. Fujita, "Hardware-Accelerated Formal Verification," in *IWLS '08: 17th International Workshop on Logic and Synthesis*. Lake Tahoe, California, USA: IEEE Computer Society, 2008, pp. 247–252.

[16] J. Brandt, K. Schneider, and A. Willenbcher, "Hardware Acceleration for Model Checking," in *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, C. Scholl and S. Disch, Eds. Freiburg, Germany: Shaker, 2008, pp. 179–187.

[17] D. Bona?ki, S. Edelkamp, and D. Sulewski, "Efficient probabilistic model checking on general purpose graphics processors," in *Model Checking Software*, ser. Lecture Notes in Computer Science, C. Pasareanu, Ed. Springer Berlin / Heidelberg, 2009, vol. 5578, pp. 32–49. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-02652-2_7

[18] S. Edelkamp, D. Sulewski, and C. Ycel, "Perfect hashing for state space exploration on the gpu," 2010, see URL `http://www.aaai.org/ocs/index.php/-ICAPS/ICAPS10/paper/view/1439`.

[19] J. Barnat, L. Brim, M. Ceska, and T. Lamr, "Cuda accelerated ltl model checking," in *Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on*, Dec. 2009, pp. 34 –41.

[20] S. German and G. Janssen, "A tutorial example of a cache memory protocol and rtl implementation," in *IBM Research Report*, ser. Technical Report, vol. RC23598, 2006.

[21] P. Wolper and D. Leroy, *Reliable Hashing Without Collision Detection*, ser. Lecture Notes in Computer Science. Springer Berlin, 1993, vol. 697, pp. 59–70.

[22] K. Pagiamtzis and A. Sheikholeslami, “Content-addressable Memory (CAM) Circuits and Architectures: A Tutorial and Survey,” *IEEE Journal of Solid-State Circuits*, vol. 41, no. 3, pp. 712 – 727, March 2006.

[23] “Murphi,” http://verify.stanford.edu/dill/murphi.html, 1996.

[24] T. Leonard, 2009, private communication.