Verifying an L2 Cache

Chae Jubb
ecj2122@columbia.edu

1. INTRODUCTION

Verification is hard. Especially for complex pieces of hardware. We can cover common use cases of that hardware with simple directed simulation. More robust is the notion of a fully randomized test bench. Using this technique we can see a wide variety of test inputs, covering and testing more edge cases. However, one of the most powerful tools is formal verification. Formal verification techniques allow us to *prove* the correctness of a design. While a randomized test bench may give us a fairly high degree of confidence our design is correct, formally verifying a property ensures it must always be correct. However, properly formulating properties—and corresponding assumptions about their use—can be time-consuming and difficult. Taking a combination of these approaches can allow us to have the best of both worlds: common case behavior easily validated throughout development and edge case behavior and bounding properties verified at particular development milestones.

We now show the application of a combination of these techniques to the development of a hardware system: a cache.

2. CACHE OVERVIEW

Constructing a new computer architecture often involves creating and implementing a memory hierarchy. This hierarchy should be tuned for the end application of that new architecture. In order to boot and run Linux on the new Embedded Scalable Platform architecture [2, 1], we must implement a coherent memory hierarchy. Before this implementation began, each LEON 3 processor only had a private split L1 cache. In order to implement a coherency protocol, we chose to implement a private unified L2 cache as well as a shared L3 cache (see Figure 1). Both levels of cache are multi-way associative.

For this paper, we focus on the implementation and verification of the L2 cache. To properly motivate this, we give a short overview of the microarchitecture of the L2. As shown in Figure 2, we see the L2 cache consists of three major components: the coherency FSM, the data bank, and the tag bank.

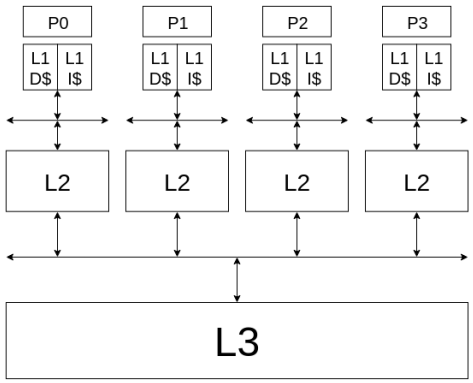

Figure 1: We see here the basic architecture of the memory hierarchy of our multi-core system. We note the private L1 and L2s as well as the shared L3 and main memory.

3. VERIFICATION PLAN

Each of the three major components, the coherency FSM, data banks, and tag banks, has separate verification and validation plans. Each of these plans has a combination of simulation and directed test benches as well as formal verification. These two types of validation have different sets of advantages and disadvantages. In conjunction, these two approaches can create a robust validation plan. We now break down and discuss the verification and validation plan for each of these subcomponents before discussing the results and outcomes in Section 4.

3.1 Data Bank Verification Plan

The data bank of the cache is quite straightforward architecturally (See Figure 3). It simply has one set of address, data, and write enable inputs. The module also has a data output. Internally, there is very little complexity in the internal microarchitecture: either a simple lookup or write.

We begin with a SystemC HLS specification for the

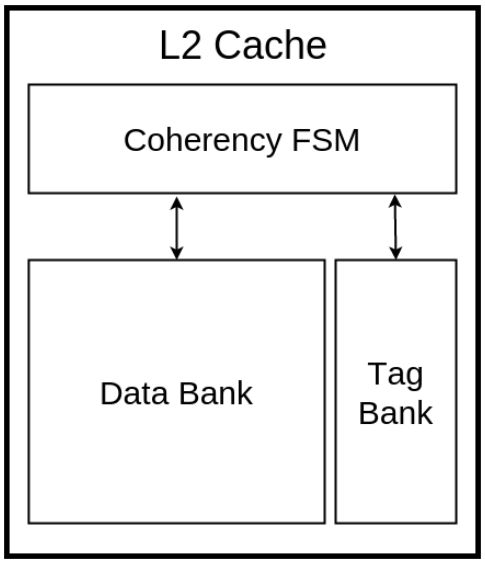

Figure 2: We see here the more detailed microarchitecture of the Level 2 cache. We note the three major components: coherency FSM, data bank, and tag bank.

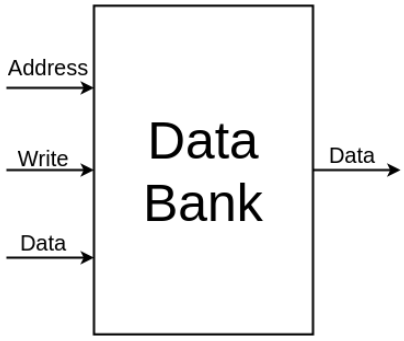

Figure 3: We see here the interface for the data bank of the L2 cache. We note it has a standard memory interface.

data bank. We accompany this with an HLS specification of a directed test bench. Because of the simplicity of the module, we expect to have full coverage with the test bench: both line and branch coverage.

After we've completed the test bench portion of validation, we use commercial tools to generate a Verilog implementation of the HLS specification. We then complete a simple check on this compilation by running the generated Verilog with the previous HLS test bench. The test bench is not re-implemented at the RTL level. We do not generate coverage metrics on this generated RTL as we've already (hopefully) covered enough of the HLS design. We do, however, run our formal verification on the generated RTL.

Because we expect to fully cover the data banks, our formal verification process will center around functional properties of the module. Again due to the simplicity of the design, we have only a few choices here. The primary focus will be determining a latency bound. The latency bound determination must be done at the RTL level. This is because the HLS specification does not fully specify timing of the design. The design is not fully scheduled until we generate the RTL implementation.

3.2 Tag Bank Verification Plan

Like the case of the data bank, the verification of the tag bank consists of two components: directed test benches and formal verification. The validation of this module is more difficult than the validation of the data banks as the internal microarchitecture of a tag bank is much more complicated.

3.2.1 Tag Bank Architecture

The duty of the tag bank is to track metadata-type information about each cache line stored in the data bank. This includes data such as validity of the data, tag bits for each (set, way) pair, and the line's current coherency state. It also must contain logic to evict lines from the cache as well as coordinate flushing of the entire cache.

In order to accomplish its responsibilities, we architect the tag bank as described in Figure 4. The module consists of an interface with the primary control of the cache. This interface has three planes. The first is the usual interface for normal lookups. The second is used to provide information concerning tags to be evicted. The final is used when flushing the cache.

3.2.2 Validation and Verification

Due to the complexity of the tag bank, we'll approach its test bench verification in two ways. The first will be a simple, functional check of the basic operation of the HLS specification using an HLS harness directly connected to the tag bank. The second, more complete method will be using the tag bank embedded within the

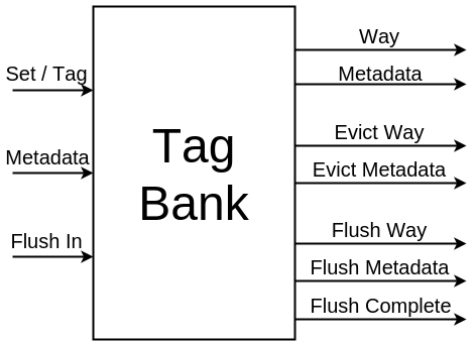

Figure 4: We see here the interface for the tag bank of the L2 cache. Note the three output plans: nominal, eviction, and flushing.

entire L2 cache. In this second case, the entire L2 (as an HLS specification) will be connected to an HLS test harness.

We repeat a process analogous to the data bank validation with the generated RTL implementation of the tag bank's HLS specification.

After the directed validation is complete, some properties of the tag bank will be formally verified. We focus on liveness properties as well as attempting to put a latency bound on requests.

Determining a latency bound on requests very nearly requires using formal verification techniques. A directed test bench would attempt to bound the latency for a certain subset of possible requests. Formal verification techniques allow us to explore the entire state space of the internal logic to prove an upper bound on latency.

3.2.3 Importance of Latency Bound

Fixing a latency bound for the tag bank is especially important. This submodule will serve at least one request for each request to the whole L2. Thus, minimizing and finding a latency bound is critical to system performance. This is especially important for a L2 cache, which we aim to have a read hit latency of approximately 10 cycles.

Because of the importance of this case, we make sure to write special properties for the read hit case in hopes of determining a tighter latency bound than the general case.

Knowing a proven upper bound for latency allows us to focus optimization efforts appropriately as we can now upper bound the request latency of certain sections of code.

3.3 Coherency FSM Verification and Validation

The main control of the coherence protocol exists in a Finite State Machine. This FSM takes as input the request received by the cache (either from the L1 or from the L3) as well as the current state of the affected line. From this, message(s) are sent and the coherency state updated if necessary. The FSM process communicates with both the data and tag banks, ensuring that the tag bank always contains the updated metadata for corresponding entries in the data banks.

For this module of the cache, we solely use a test bench-based approach, so we will focus less on this module in this report.

We begin with a HLS specification and a directed test bench. We use the HLS test bench in conjunction with copious assert statements in the HLS specification. When generating an RTL specification, we lose the asserts in the HLS specification[1] but reuse the HLS test bench and its assertions.

Completing a formal verification on the entire cache would be difficult due to the state space explosion problem so for practical purposes, we focus on increasing coverage. Were there more time available, some uncovered or complex functional properties would be examined through the lens of formal verification.

4. VERIFICATION PROCESS

The verification process began with an initial implementation of the cache completed. The test benches were generally built up alongside the actual implementation. These directed test benches were used to find minor functional errors during the development of the cache. After this, formal verification was begun and used to further refine the design. The formal verification was generally used to fill gaps left by the test benches (where applicable). Because of the thoroughness of formal verification, sometimes approximations needed to be made. These are discussed in detail below.

4.1 Data Bank Verification

As the data bank module was quite simple, it was primarily verified with the goal of becoming acclimated to the tooling, including the major step of generating and refining formal assumptions used in the verification process.

The test bench initially uncovered some minor errors in the protocol used at the interface boundary. These errors were corrected easily. Again due to the simplicity of the module, we were able to fairly easily achieve full test coverage, determined using `gcov` and related tools. All lines were covered as well as all branches.

An RTL implementation was then generated using the HLS specification. This implementation also passed the constructed test bench. Because the HLS specifica-

[1]Creating assertions shared between the HLS specification and an RTL simulation would have required some re-architecting of the cache to have another module tracking a large amount of internal state.

tion had full coverage, the RTL implementation passing the test bench served as a moderately-useful check on the correctness of the RTL.

We then began the formal verification process. The first set of properties tested consisted of cover properties on all control signals of the module. The lack of coverage would give a clear, early indication that some signals were unnecessary or there was some issue with the implementation. These properties were easily verified.

4.1.1 Formal Assumptions

At this point, focus shifted to formulating proper formal assumptions. As discovered during early verification, formulating a proper set of formal assumptions proved crucial to properly verifying the module. While we had a simple interface and few assumptions were necessary, determining the proper assumptions proved time-consuming. The assumptions generated here fell into three primary categories:

1. Latency-Insensitive Interface
2. Readiness for Response
3. Validity of Input

We discuss these major categories in depth below, noting the bases for and consequences of the assumptions.

Latency-Insensitive Interface.

The data bank module, like the other modules, provided a latency-insensitive interface. This meant that no part of the external interface relied upon any protocol dependent on timing. Each step of the protocol had to happen in a specified order; but, the latency between steps is neither fixed nor bounded.

This was accomplished using a three wire interface: *ready*, *valid*, and *data*. The receiver asserts *ready* if and only if it is ready to receive data. The sender asserts *valid* if and only if *data* is a valid transmission. By convention, *ready* and *valid* are held high until the transmission is considered complete: when *ready* and *valid* are asserted during the same cycle.

These assumptions were not initially added to the verification process. Without them, verification of even simple properties failed because the interface didn't follow the expected protocol. Adding in these properties formally specified that interface protocol, limiting the search space to those sequences following the proper protocol. The following categories of properties were also important in properly determining the truth value of given properties, though implementing the latency-insensitive interface identified the importance of a proper set of formal assumptions.

Readiness for Response.

We note that when integrated into the cache, only one request will be in flight at a time. That is, a request will be initiated by the coherency FSM, processed by the data bank module, and then completed by the coherence FSM receiving a response from the data bank module.

Because of this one-request-in-flight assumption, we can assume that the output channel will always be ready for a response. Thus we are simply waiting for the data valid signal to be driven high.

Validity of Input.

This set of assumptions also relies upon the assumption that there is only one request in flight at a time. Additionally, we use the basis that all input signals will have the corresponding *valid* signals set high on the cycle cycle. This means the address (and data if a write request) will be set high together. More specifically *data valid* will not rise unless *address valid* also rises.

Initially, an improper assumption was made. It was originally specified that the input valid bits wouldn't be asserted until an output was given. With this invalid assumption, the formal verification did not complete. After some debugging, it was determined that this assumption was not true after a write request as a write request did not return any data. After removing this over-constraining assumption, we proceeded with the formal verification.

4.1.2 Property Verification

The primary property of the data bank verified was a latency bound on read requests of a cache line. That is, when a proper read request is made, data out becomes valid.

On our first iteration, the tightest upper bound we could produce was 9 cycles. For an L2 cache with a read hit goal of 10 cycles, this was not acceptable. After further examination, this occurs because each word of the cache line was read in its own cycle. As a result, HLS knobs were adjusted, giving the relevant memories enough read ports to support reading a whole cache line in one cycle. As a result, we were able to place a much tighter upper bound on the read latency of the module.

Because of the small number of states in the simple data bank design, the latency bound properties were easily proved or disproved. The timeout on the verification process never needed to be extended longer than 1 minute so no clever techniques needed to be used in order to see reasonable upper bounds.

Omissions.

Notably missing from the verified properties is any check for correctness of the data itself. This would require a fairly complex property. We would need to spec-

ify that a read returned the most recent value written to any given address. We would also have to build in that a read without a previous write can return any value.

However, this property can be easily validated (but not formally verified) using the test bench. A fully randomized test bench[2] would give sufficient confidence in the correctness of the data handling. As all directed test benches check for correctness of data, we omit a formal verification of data correctness and constrain ourselves to a formal verification of control correctness.

4.2 Tag Bank Verification

After completion of the data bank verification, we move to verification of the more complex tag bank. While the validation and verification process is more complex and nuanced, it follows the same basic structure.

We begin with an HLS specification of the tag bank, paired with a test bench that was developed alongside the tag bank itself. This test bench is used to verify the basic functioning of the tag bank (as well as time it in select common use cases). The more substantive test-bench–based validation is done with the tag bank embedded into the larger L2 cache. While the tag-bank–specific test bench only covers common cases, the larger L2 test bench covers less common cases, such as eviction and flushing.

4.2.1 Formal Assumptions

As with the data bank, we begin with a set of formal assumptions guiding the model checker's state space exploration. Here we have the following categories of assumptions formulated for verification of the tag bank (including some overlap with the data bank assumptions):

1. Latency-Insensitive Interface (as in Section 4.1.1)
2. Readiness for Response (as in Section 4.1.1)
3. Validity of Input (as in Section 4.1.1)
4. Flushing Protocol
5. Optional Request Inputs

Below we discuss the new categories of formal assumptions.

Flushing Protocol.

The flushing protocol has two main requirements. The first is that we only have one flush in progress at a time. This means we do not see a second flush request before the first flush request is processed and completed. This assumption is assured because the L2 cache is private and the associated processor stalls during a flush.

[2]That is, many reads and writes to randomized addresses, running for a sufficiently high number of iterations.

The second is a small departure from our previous assumption that the output channel is always ready to accept new data. This assumption must be slightly modified such that we can have a flush request being processed concurrently with a non-flush request. We do, however, retain the assumptions that any two non-flush requests will be non-overlapping and also that the coherency FSM will wait for a response to a non-flush request.

As a result of this relaxed assumption, we create a small, but bounded, maximum waiting period. Identifying the proper latency here would require performing a formal verification on the entire tag bank to get a proper upper bound. This is impractical (explained in more detail in Section 4.3) so we choose a number large enough that we are likely to see some sort of issue arise[3]; it should be small enough, though, to ensure that we don't suffer from the state space explosion problem.

Optional Request Inputs.

The tag bank internals rely upon the notion of optional inputs. That is, it determines the type of request, and thus expected action, based upon which inputs are supplied with a request. Like the data bank, we have a formal assumption that all parts of the request will be received at once. That is, optional inputs will not suddenly become valid while a request is being processed.

The tag bank has more optional inputs than the data bank so we must take extra care to ensure that they're being supplied to the tag bank properly as the internals of the tag bank use them in a more nuanced way.

4.2.2 Covering Branches

After we generate a set of valid formal assumptions, we can begin to design properties meant to verify the correctness of the tag bank. We begin by examining the usefulness and coverage of the test bench, using that as a starting point for our formal verification efforts.

With a combination of our two test benches, we are able to achieve quite high line and branch coverage of the HLS design. The coverage is more completely specified and recorded in Table 1. It is important to note that the branch coverage metrics includes assertions that did not fire. Many were inserted using the SystemC `sc_assert`; therefore, `gcov` and similar tools did not recognize them as assert statements to be ignored in coverage metrics.

Excluding the false positives caused by `sc_assert` statements, we have fairly high branch coverage. The branches not covered generally stem from early resolution of branches. That is, a branch will have a complex conditional that doesn't require computation of all sub-

[3]A latency of one or two cycles could be small enough that an issue hides, especially an issue that violates the assumption of a latency-insensitive interface.

	Tag TB	Full L2 TB	All TBs (excluding `sc_assert`)
Line	69%	100%	100%
Branch	52%	79%	88%

Table 1: Description of Tag Bank Coverage Metrics for multiple test bench sources. Excluding `sc_assert` statements means not counting branches not taken because the assert did not fire. We can see that the tag-bank–specific test bench does not test a large portion of the tag bank. This is because it was intended only as a check on common cases. When we integrate the tag bank into the full cache and run the corresponding test bench, we stress the tag bank and see much higher coverage.

expressions to determine its truth value.

We do note, however, one specific uncovered branch. This would derive from a certain set of inputs (never given by the L2 coherency FSM). We wish to eliminate this uncovered branch to increase our confidence in the tag module. It then seems we have two options: we could simply eliminate the branch as dead code (and simultaneously cover that input condition with an assert); or we could construct a property covering that input condition to verify with model checking.

Because a further refinement of the cache could potentially introduce the set of inputs in question, we do not want to eliminate its handling in the HLS specification. Therefore, we write a property covering that input set.

We saw here a good use case of formal verification: covering a hard-to-reach branch.

4.2.3 Liveness and Latency Bounds

Now that we have a higher confidence of the correctness of the design, we begin to verify the liveness and bound the latency of the module. Verifying the liveness simply consists of writing a property such that if we send any proper request, we will eventually see some sort of response. This type of property is verified fairly easily by the model checker.

General Latency Bound.

Now that we've easily shown that the tag bank is live, we can attempt to bound the latency of different classes of requests. (The results of this overall process are described in Table 2.) We begin with the most straightforward: attempting to bound the latency of any general request. To do this, we begin with a high bound and gradually reduce it until the model checking begins to take sufficiently long. We arrive at a loose upper bound of 7 cycles for the response to any arbitrary request.

Non-flushing Latency Bound.

More interesting latency properties involve bounding the response time of common requests. In the common case, the tag bank is not attempting to serve a flush nor does it receive a request for a flush. In this case, we are able to bound the response latency to 4 cycles.

Read Hit Latency Bound.

We further specify constraints so we can attempt to place a bound on the read latency of a request that hits the cache[4]. Doing this proved difficult for two primary reasons.

There is no external signal that indicates the request was a hit: a hit is simply recognized by taking a certain branch. In order to externally identify a read hit, we must create a debug build of the tag bank which exports some internal state. We do this by creating an interface signal which identifies which block of code the module is executing.

Once we have exported our internal state signal, we begin by making a simple coverage property stating that the cache is able to process a hit request in 3 cycles. This is our end goal for the most common case, we verify it is possible before continuing with our bounding process. This property is proved fairly quickly as we worked out most of the formal assumptions while proving previous properties (including liveness).

Finally, we begin the process of bounding a read hit. This proves difficult. We are able to formulate some property that gives us a bound, but the formulation causes the verification process to be long. We can easily find a loose upper bound of 6 cycles. Allowing a longer proof time, we are able to reduce the bound to a tighter upper bound of 5 cycles.

Improving Read Hit Bound.

This bound of 5 cycles seems incongruous. Previously, we saw that without flushing (and no constraint on the hit), we found a bound of 4 cycles. Why can we only find a bound of 5 cycles here? Even more odd is that it takes so long to find.

We had not correctly specified the assertion so that flushes could not interrupt our request. Without that proper specification, we inadvertently raised the upper bound. We correct this mistake in the property and are able to easily find an upper bound of 3 cycles for a read hit.

4.2.4 State Space Explosion

As mentioned above, we bound the latency by bringing it lower and lower until either the property fires or

[4]We remind the reader that here the latency bound is simply on the tag bank access, not a bound on the entire request.

	Latency	Proof Time
General	7	seconds
General (exhaustive)	6	hours
No Flushing	4	seconds
Read Hit	3	seconds
Read Hit (Cover)	2	seconds

Table 2: Overview of latency bounds (cycles) for specific types of requests.

	Full L2 TB	Full L2 TB (excluding explicitly untested code)
Line	88%	94%
Branch	62%	63%

Table 3: Description of L2 Coverage Metrics. We define explicitly untested code to be (control input, coherency state) pairs that have been partially implemented but never used and never tested. We also note that most (but not all) of the uncovered branches are uncovered because an assert does *not* fire. Similarly, line coverage is lacking mostly (but not completely) because certain (control input, coherency state) pairs are designed to fall through to a `default` clause because they are unexpected.

we exhaust the time limit verifying. When we bring the latency low enough that the property first fires for latency n but not $n+1$, we have easily found a tight upper bound for the property.

However, when we simply run out of time, we must be slightly more thorough in our bounding. We begin with a loose upper bound that is proved in a reasonable amount of time. We additionally have a loose lower bound, disproved in a reasonable amount of time. For this specific module and properties, we do not see any separation in the loose upper bound and the loose lower bound of more than 2 cycles.

In order to allow ourselves to verify the design more or less strictly, we include properties with multiple latency bounds. Then, depending how much time we allocate for verification, we can see the results of more properties. That is, we include properties that we know for most verification runs will end inconclusively as well as looser versions of those same properties that will be proved or disproved very quickly.

Based on the incremental changes from version to version, the verifier can decide upon an appropriate time limit for the verification and derive a upper bound on latency appropriately.

4.2.5 *Flushing Properties*

One of the most unexpectedly difficult properties for the model checker to prove was coverage on the flush complete signal being asserted. Proving this coverage property takes a couple hours. We see this as another example of the state space explosion (and challenge of formal verification). This particular occurrence is derived from the size of the cache. The implementation of the cache tested is an 8-way associative cache with 128 sets. Thus, it takes a fairly long sequence to move through flushing every possible entry. We could reduce the time taken to verify this property by verifying an equivalent, but smaller, cache.

4.2.6 *Limits of Formal Verification*

One change from the basic verification outline of the data banks was the constraining of the reset input on the tag bank to inactive. The original verification took place without this constraint. However, this caused all properties to fail. After further investigation, this is because the tag bank takes a long time to fully reset after the input reset signal goes inactive. That is, no matter how long reset is held active, it takes a few thousand cycles after the reset is brought inactive before the module is fully reset and usable. This is an artifact of the HLS toolchain used.

To incorporate this into the model, we'd have to modify all properties to ensure they waited sufficiently long after the reset signal becomes inactive. Alternatively, the module could have been modified to output some ready signal, but that was unnecessary when building the module for integration with the entire cache, so we avoid adding it since we don't want our verification model to diverge too much from the model actually used in the cache. We instead slightly constrain the model. This seems acceptable because in practice we do not reset except for start-up and all of our properties are disabled if the input reset signal ever goes active.

4.3 Coherency FSM Verification

As previously mentioned, the validation and verification of the coherency FSM consisted solely of a directed test bench. It was developed alongside the FSM itself. This test bench is the same that is used to more fully exercise the tag bank. Its thoroughness provides a high level of both line and branch coverage of the source files, described more fully in Table 3.

Due to the complexity of the state machine (both a high number of possible input control messages and high number of possible coherency states), we do not attempt to bound the latency or perform other formal verification on this portion of the cache[5]. This decision was primarily based on experiences with flushing the

[5]Were there more time, some basic model checking would be done on the entire cache.

tag bank (see Section 4.2.6 for more detail).

5. FUTURE WORK

The most naturally flowing future work would be extending the formal verification techniques to the entire cache, including the coherency FSM. In order to do this effectively and meaningfully, we would have to be clever about the properties we come up with in order to avoid a state space explosion. In fact, in might be more plausible to further subdivide the cache into smaller, more easily verifiable components. We could formally verify individual portions and take a simulation-based approach to validate the interconnections between those modules.

Were more time available, we could also complete some sort of verification on the HLS specification. As previously mentioned, all formal verification was completed on the Verilog implementation generated from the HLS specification. Additionally, the only source of validation of the generated Verilog was running the same HLS test bench with the HLS specification and the Verilog implementation.

6. CONCLUSION

The validation and verification process of a moderately complex piece of hardware proves and interesting challenge. Because formal verification at a large scale is not always time-practical, we must approach validation with a combination of simulation and formal verification. Each has their own set of advantages and disadvantages.

Simulation can provide quick feedback as to the correct operation of critical scenarios. It can ensure that common cases act as intended. However, it has trouble with the rare cases. It is very difficult to, for example, stimulate an interface with the correct sequence of inputs in order to hit every corner case or cover every branch. A simulation-based approach would make it very difficult to create a bound on behavior. We could generate a bound on common behavior, but including the uncommon behavior becomes a monumental task. This is where we turn to formal verification.

Using formal verification, we can be sure that certain properties always hold for our design, no matter the input sequence. We can often cover uncommon edge cases with ease. We can determine bounds on behavior (e.g. latency) much more easily. However, we see a certain set of drawbacks. The language to describe formal properties is limited; this makes certain, potentially vital, properties difficult to enumerate. Complex designs are quite difficult to verify because they suffer from the so-called "state space explosion". In order to efficiently verify a complex design, we must derive a meaningful and useful set of formal assumptions. Formulating these assumptions can be difficult due to the nuanced behavior of a large piece of hardware.

These two approaches can be used in conjunction to drastically increase confidence in the correctness of a design. Simulation and directed testing may get us most of the way there, but formally verifying portions of the design gives that last bit of push that simulation is unable to provide.

7. ACKNOWLEDGMENTS

The coherent cache hierarchy mentioned here was designed and implementing while working in conjunction with Prof. Luca Carloni's System-Level Design group, especially Paolo Montovani. The approach to the verification process was formulated with the input of both Paolo Montovani and Giuseppe Di Guglielmo, both of whom work with the SLD group.

8. REFERENCES

[1] E. G. Cota, P. Mantovani, G. Di Guglielmo, and L. P. Carloni. An analysis of accelerator coupling in heterogeneous architectures. In *Proceedings of the 52nd Annual Design Automation Conference*, page 202. ACM, 2015.

[2] C. Pilato, P. Mantovani, G. Di Guglielmo, and L. P. Carloni. System-level memory optimization for high-level synthesis of component-based socs. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2014 International Conference on*, pages 1–10. IEEE, 2014.