Glue: A Scalable Memory Hierarchy with HLS

ESP Cache Coherence

Chae Jubb
ecj2122@columbia.edu

1. INTRODUCTION

As we construct more complex heterogeneous architectures, we must ensure that our development of the memory hierarchy scales with the computational subsystem. The effect of additional processors and hardware accelerators will see greatly diminishing returns unless we also develop and implement efficient memory subsystems.

One of the largest memory performance bottlenecks is the coherence protocol. Typically we use a snooping protocol on a shared bus. However, as we increase the number of listeners on the bus, performance greatly degrades. We turn to the usual solution of using a more complex directory-based protocol.

Because of this complexity, we turn to high level synthesis (HLS) tools. Using HLS tools allows us to prototype and explore the design space much more quickly than we would be able to with an RTL design process. Additionally, we are able to work at a higher level of abstraction, allowing us to more easily implement latency-insensitive interfaces. Interfaces of this type allow much more flexibility in the use of the resulting design.

Implementing a cache hierarchy from a high level specification proves difficult, however, as we constantly strive for latency-optimization. Usually, we would create a hyper-optimized design at the RTL level[1]; however, we attempt to stress the available HLS knobs and produce a highly optimized design.

2. CURRENT ARCHITECTURE

We add this memory coherence protocol to an already-existing heterogeneous platform. The basic architecture of this existing Embedded Scalable Platform is a so-called "tiling" of computational components (See Figure 1). Our specific application consists of multiple general purpose CPUs (LEON 3) as well as hardware accelerators and memory controllers.

The construction of this architecture focuses on scalability and flexibility. We should be able to add as many tiles as we necessary for a given application without seeing a large degradation in performance. We also intend to build the architectural framework so that we can flexibly interchange these tiles as application requirements change. A large part of creating a flexible system is designing the tiles with a certain amount of regularity.

[1]Compare to writing critical code in C rather than Python, or even in x86 rather than C.

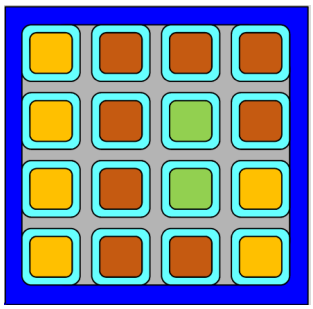

Figure 1: We see here a high-level architectural view of an Embedded Scalable Platform instance. We see here a 4x4 tiling of general purpose CPUs (brown), accelerators (yellow), and memory controllers (green). All tiles are superimposed on a NoC (gray), which is interfaced via the light blue wrappers.

When constructing our coherence subsystem, we must keep these goals of ESP in mind. Above all is the need for scalability. Design decisions must not arbitrarily and unnecessarily limit the performance of the system.

Our end goal for the system, after implementing the coherence protocol, is be able to boot Linux on multiple processors. Linux requires cache coherence when running on multiple cores. Many interesting use cases of the ESP architecture involve utilizing multiple cores[2]. Running a multi-core enabled Linux would allow us to accomplish this goal.

3. COHERENCE BACKGROUND

The primary goal of coherence protocols is to ensure that all processors see the proper value on every load.

[2]It really is a shame to have an architecture with so many cores that can't be used to run a multi-core program with Linux.

This potential problem is introduced by the presence of private caches. When a hierarchy contains private caches, multiple processors can have their own copies of the same data. When this occurs, we must ensure that the system as a whole has some sort of *coherence* such that stores are visible to all processors in a valid ordering of the instructions.

All but the lowest-level programmers should not need to consider these coherence effects from multiple processors sharing a set of data. This could occur in such instances as a multi-threaded program using a shared buffer or a process migrating across cores.

When constructing our new cache coherent memory hierarchy, we chose the standard MESI protocol. In this protocol, a given cache line can be in 4 primary states:

Modified The cache is the only writer; no other caches are readers.

Exclusive The cache is the only reader; no other caches are writers. This state exists to allow efficient upgrades to the modified state.

Shared The cache is a reader; other caches may be readers, but no other caches are writers.

Invalid The cache is neither a reader nor a writer.

Typically the *snooping* variant of this protocol is implemented. In this situation, all cores listen on a shared bus and update their states accordingly. It it typically chosen because it is relatively simple and sometimes lower latency compared to the *directory-based* variant—though less scalable.

The directory-based variant, however, involves a separate directory controller (usually a lower level cache). Coherence requests are made to that controller, which then sends responses and requests to the lower level caches.

We choose the directory-based MESI protocol for the ESP application because of its scalability. It seems quite implausible to implement a scalable coherence protocol that relies on all processors share a single bus. In the next section, we discuss the architecture of this cache-coherent hierarchy.

4. HIERARCHY ARCHITECTURE

We begin by visualizing an ESP hierarchy similar to Figure 1 that contains four LEON 3 processors and a shared DRAM controller. Each of the processors occupies its own tile; the DRAM controller also occupies a tile. From this point, we introduce a cache coherent hierarchy.

4.1 Shared LLC

Our first, most obvious, addition to the existing architecture is a shared Last Level Cache (LLC). This provides us with a relatively large on-chip cache that will help us avoid many requests to the much slower DRAM. We choose this cache to have a write-back policy. Because it is a shared, last-level cache, this policy can help avoid many accesses to memory, necessary on an eviction or a flush. Further, to simplify our later choices, we choose the cache to have a high enough associativity such that we never need to perform a recall of data from a lower level cache. Accordingly, the L3 cache is inclusive of the L2 cache.

4.2 Unified Private L2

We now recall that the LEON 3 has a split L1 cache. Both the L1D and the L1I have a write-through policy. Were we to connect the L1 directly to the LLC and implement the coherence protocol directly in the L1, we would see an inordinate amount of traffic. That result stems from two major issues.

Firstly, we'd see a lot of traffic because each processor would have two caches talking to the directory. Secondly, because the cache is write-through, we'd see a write request to the directory on every single processor write. In this situation, the cache loses many of the advantages given by spatial locality.

We can reduce the frequency of these requests (thus allowing for better scaling and lower average latency) by inserting a private L2 between the private L1 and the LLC. The mere existence of this cache lowers the complexity of the bus, given that we halve the number of connections to a shared bus. Additionally, we can give this cache a write-back policy. This will greatly reduce the number of requests to the directory. With less congestion, we can hope to see a decrease in the average latency of a response.

Like the LLC, we make this L2 cache as associative as necessary in order to prevent recalls from the L1. Given the specifications of the LEON 3, we find that an 8-way associative L2 cache is sufficient for this purpose.

4.3 Summary

We begin with a very simple memory hierarchy, consisting only of a split L1 and main memory. We add a private L2 per core as well as a shared LLC. As a result, we must design an L2 cache, an L3 cache, and a protocol for the two to communicate.

The LEON 3 is outfitted to have the L1 caches communicate with an AMBA bus for memory requests. Thus, we place our L2 on an AMBA bus shared with the L1s so we do not require modification of the provided processor package.

Due to the expected size of the L3, we plan to allocate it in its own tile in the eventual ESP architecture. For this reason, it will need to be connected by the NoC to all the relevant L2 caches. For this reason, we design our hierarchy with a NoC between the L2 and L3. We

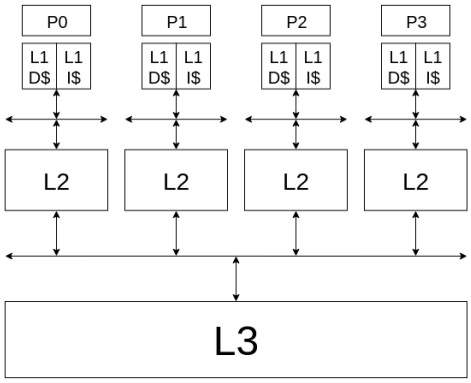

Figure 2: We see here the basic architecture of the memory hierarchy of our multi-core system. We note the private L1s and L2s as well as the shared L3, which interfaces with main memory. The L2s connect to the L1 via an AMBA bus; the L2s connect to the L3 via the ESP NoC.

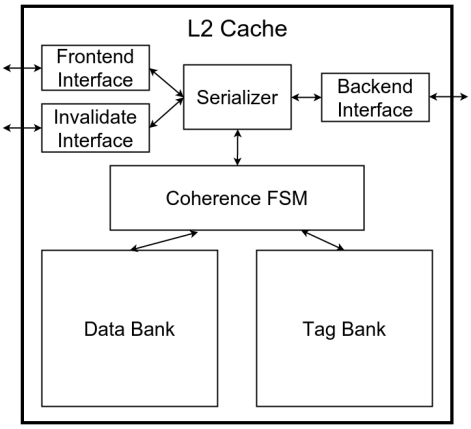

Figure 3: Basic architecture of L2 cache. We make note of the most important components: the coherence FSM, the tag bank, and the data bank. Other components are simply responsible for routing messages appropriately.

see this final hierarchy described in Figure 2.

5. L2 CACHE ARCHITECTURE

We first begin the design of the high level specification of the L2 cache. The L2 caches will all communicate with the L3 directory to ensure coherence of the containing data. These connections are made via an RTL wrapper to an AMBA bus (on the frontend L1 side) and NoC (on the backend L3 side), as described in Figure 2. We save more detailed discussion of the HLS process for Section 8.

We create a modular architecture for the cache, surrounding an FSM-like implementation of the coherence protocol. To do this, we conceptually break the cache into the following submodules and subprocesses (shown in Figure 3).

Frontend Interface Receives requests from the frontend bus, putting them into a common form.

Backend Interface Receives requests from the backend NoC, putting them into a common form.

Serializer Serializes cacheable requests from the frontend and backend for processing by the FSM.

FSM Maintains coherence, processes incoming messages, sends outgoing messages, and communicates with the tag and data banks.

Tag Bank Holds coherence state and other metadata for all lines stored in the cache.

Data Bank Holds data stored in the cache.

The first three subprocesses are uninteresting: they simply serve as request flow controls. The other three modules, FSM (Section 5.1), Tag Bank (Section 5.2), and Data Bank (Section 5.3) are described in more detail below. Before this, we note the reasoning for a few design decisions.

Non-Cacheable Requests.

For simplicity of the RTL wrapper, non-cacheable requests are routed through the cache as cacheable requests are. However, the serializer process will simply route them from one interface to the other without involving the coherence FSM. In this way, with all requests flowing through the same pipeline, we are able to avoid race conditions and complexity that might occur were we to implement separate data and control paths.

Flushing Entire Cache.

A usual flush instruction, like in the `x86` or `ARM` architectures, will flush only a single address[3]. However, the LEON 3 does not have the capability to flush only a given address. Flushes in the provided L1 evacuate the entire cache rather than specifically listed blocks. Implementing a single-block flush in the L2 would require modifying the internals of the LEON 3 processor, which is undesirable. Thus, we implement our L2 flush as a cache-wide flush. That is, every valid entry in the cache is removed. The details of this implementation are given below in Section 5.1.5.

5.1 Coherence FSM

The bulk of the logic in the cache is contained in the coherence FSM. This module is responsible for ensuring that all cache lines contained in the cache are kept in a

[3]More correctly, a block containing a specific, given address

coherent state. It coordinates the tag and data banks, ensuring that after each transaction is complete those two modules are consistent with each other.

5.1.1 Basic Control Flow

The coherence FSM will first receive a cacheable request. It then immediately checks the tag bank for the state of the containing cache block. After creating this message-state pair, the FSM will perform an action. All actions are written such that none will block. That is, actions are completed quickly without waiting for any external response. We attempt to eliminate as much stalling as possible to allow us to concurrently processes as many requests as possible.

In order to accomplish this, we have many intermediary states. For example, when a read request arrives for a line that is in the invalid state, we send a request to the lower-level cache for the data and transition to a state waiting on data. We then eventually receive the data and only then do we move into the shared state.

5.1.2 Single Core Optimizations

The working specification for the MESI protocol[2] assumes a cache that serves multiple cores. However, as our L2 serves only a single core, we are able to make some optimizations. When serving multiple cores, it is possible to have multiple in-flight read or write requests for the same piece of data. If that data is not in the cache and we are waiting for it to be returned from a lower level in the memory hierarchy, we must stall. Implementing this would greatly complicate the architecture as we would need many buffers per core.

However, moving to a single-core architecture allows us to avoid this stalling. Because the core itself is stalling waiting for the response from a read, for example, to return, it will not issue a second concurrent request. As a result, we can simplify the architecture to avoid stalling in nearly all scenarios. The more we can reduce the occurrence of stalling, the closer we can stay to our intended model of control flow (Section 5.1.1).

5.1.3 Forward Plane Stalling

Unfortunately, we cannot eliminate all stalls in our FSM. We can, though, limit the necessary stalling to the forward plane of our backend input. The only stalling necessary occurs when the cache is in an intermediary state waiting for data from a lower level cache and receives a message on the forward plane, asking for data to be forwarded to another cache. Because the cache does not yet have the data, it must stall until it does receive the data. We still must hold this initial message in a buffer so that we can receive the data and serve the initial request from the processor that initiated that data fetch before finally serving that forward message.

In order to avoid the need to buffer multiple forward requests, we assume that the RTL wrapper will use its forward plane buffer to hold any additional forward plane messages. The coherence FSM can not hold further messages because all backend planes are coalesced into one at the RTL wrapper. Thus, the coherence FSM does not see messages on separate planes and we must stall certain types of messages internally.

5.1.4 Read and Write Buffers

Due to our non-blocking FSM model, we must temporarily store the values of reads and writes if we must fetch data from lower levels in the cache hierarchy. Storing temporary values for reads is fairly straightforward due to our single-processor assumption. The processor will be stalling waiting for the response from the read, so we will not see further frontend requests. In this case, we simply store metadata while the line is being fetched. The write buffer has a more complicated use case.

Write Buffer.

Like the read buffer, we store data in the write buffer if we see a write request yet do not have the data currently in cache. We hold the data in this buffer until we've received the cache line; at that point, we commit the write from the buffer. However, we have the additional constraint that the processor-L1 pair will continue after a write request is issued. This means we must recognize when valid data exists in our write buffer and handle subsequent read and write requests accordingly.

We keep a fixed size buffer—for simplicity, size 1. This means we are unable to handle a second concurrent write request. Additionally, because we do not yet have the data, we are also unable to serve a read request to that same cache line. In these two cases, we stall at the frontend interface until the data in the write buffer has been committed. We note, though, that we are able to handle read requests to different cache blocks while waiting for the value in the write buffer to be committed. This read request will proceed in normal fashion through the FSM.

5.1.5 Flushing

The flushing process in the coherence FSM works very closely with the tag bank. When a flush request is received from the processor, we initiate the flush sequence with the tag bank, which will continually provide entries to be flushed until it asserts a completion signal. The flushing will proceed as a sort of background job, running when there is no other work to be done. Explicitly, this means that incoming requests from the backend will have priority over flushed entries[4]. We must allow these

[4]We will not see any frontend requests during this time as

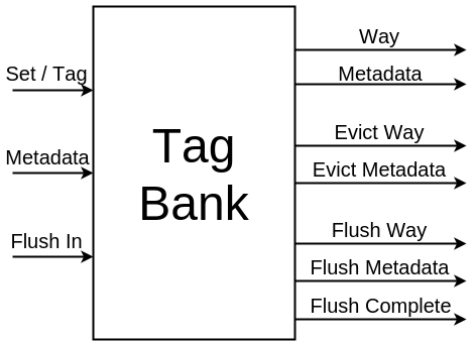

Figure 4: We see here the interface for the tag bank of the L2 cache. Note the three output plans: nominal, eviction, and flushing.

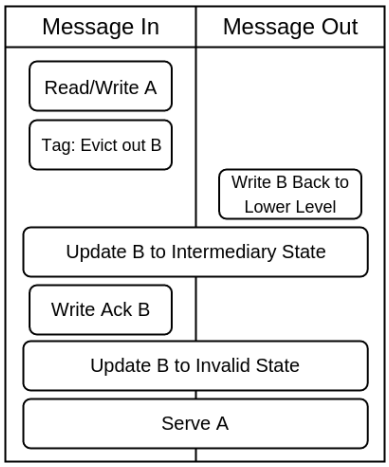

Figure 5: Flowchart of the eviction of a line from the cache. We note that we temporarily stall processing of the original request in order to handle and resolve the eviction. See Figure 6 for an annotated architecture diagram describing the same process.

backend requests to have priority over the flushing requests because in order to completely flush the entry (update its state to invalid), we must receive some sort of acknowledgment from the directory.

5.2 Tag Bank

The primary duty of the tag bank is to track metadata-type information about each cache line stored in the data bank (Section 5.3). This includes data such as validity of the data, tag bits for each (set, way) pair, and the line's current coherence 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.

5.2.1 *Nominal Transactions*

During a nominal request, the requestor (the FSM) will appropriately set the tag and set inputs (as well as any metadata updates). The tag bank will then process the request, either reading or updating the internal metadata. After that operation is complete, the tag bank will output the result on the nominal output plane.

5.2.2 *Eviction Process*

After using the cache for long enough, we will eventually have to evict an entry. This process begins with the tag bank. As described in Figures 5 and 6, once the tag bank recognizes that an eviction is necessary, it will indicate so by outputting a result on the eviction plane. This result simply identifies a line to be later removed from the cache. That is, the tag bank does not change the coherence state of any way-set pair it picks for eviction.

With this approach, as much logic as possible remains in the scope of the FSM, which can then more easily make decisions about the state of the cache as a whole. We retain the invariant, then, that only direct FSM actions lead to a change in coherence state of any given cache line.

We require a separate conceptual plane here because not every request ends in a eviction. This way, we can simply check if the eviction plane has a pending request to determine if we must handle an eviction.

5.2.3 *Flushing Protocol*

A flush of the entire cache moves through three phases in the tag bank. We begin through an initiation sequence led by the coherence FSM (Section 5.1.5). After this, we flush as continuously as possible, while simultaneously serving nominal requests. Finally, we conclude with a completion protocol initiated by the tag bank.

Initiation.

The tag bank begins a flush when requested to do so by the coherence FSM. The flush transaction is initiated by simply asserting an input signal. At this point, the tag bank enters a flushing state.

In Progress.

The basic mechanism of flushing is an iteration over all set-way pairs contained in the cache. If an entry

the single associated processor will be stalling waiting for an indication that the flush is complete.

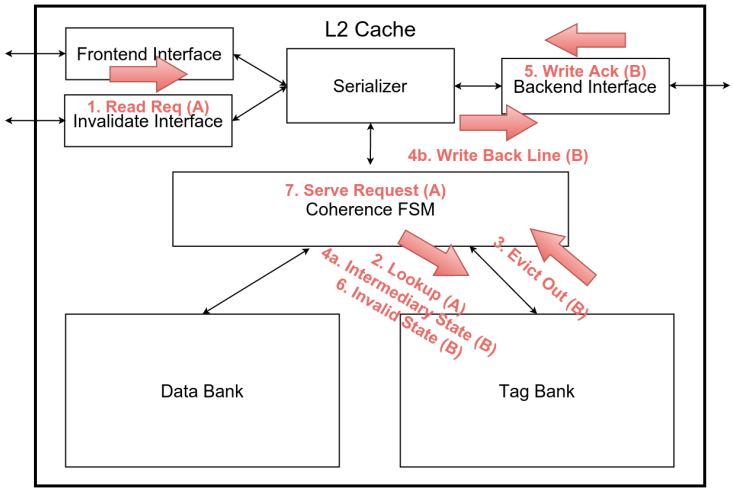

Figure 6: Annotated architecture diagram of the eviction of a line from the cache. We note that we temporarily stall processing of the original request in order to handle and resolve the eviction. See Figure 5 for a flowchart representation of the same process.

is not in the invalid state, we use the flushing plane to feed an entry to the coherence FSM. At this point, the coherence FSM will handle the entry appropriately and move to flush it to a lower-level cache. As with the eviction handling, the tag bank is simply recognizing an entry as eligible to be flushed; it does *not* update the stored state.

The overall architecture provides us with an interesting challenge here. Flushing (as eviction) does not happen in one step with one message. After moving to remove an entry from the cache, the coherence FSM will place the line in an intermediary state, waiting for a response before finally updating to the invalid state. As a result, the tag bank must continue to process nominal requests while flushing. This necessitates a tertiary flushing plane. Without a separate plane, we would have responses to nominal requests interspersed with flush requests.

During this process, however, the nominal requests continue to receive priority. In this sense, we can consider the flushing as a background job that the tag bank schedules only when there is no other, more important work to be done. This continues until the tag bank has no more entries to flush.

Completion.

Like the initiation process, the completion process is quite straightforward. The tag bank simply asserts an output signal read by the coherence FSM. Once that message is successfully passed, both the tag bank and coherence FSM recognize that flushing is complete.

5.3 Data Bank

While the data bank is by far the largest portion of the L2, its operation is quite straightforward. It is simply written to and read from by the coherence FSM in conjunction with the tag bank[5].

5.4 Invalidation

As we know from our examination of the MESI coherence protocol, we will occasionally see messages from the directory or other caches that instruct a particular cache to invalidate a particular block. Because the L2 serves as the coherence controller for the corresponding L1, we must pass this message to the L1, which may also be holding the data. For this, we use the frontend invalidation plane to pass addresses that need invalidation. If the L1 needs the data again, it will send a request to the L2, which will use the appropriate coherence protocol to ensure the data stays coherent.

6. L3 / DIRECTORY ARCHITECTURE

The overall structure of the L3 cache (Figure 7) is very similar to the structure of the L2 cache (Figure 3). We must simply replace the FSM with one appropriate to serve as the directory for our MESI coherence protocol. We also slightly alter the interface. We retain the notion of separate frontend and backend interfaces. Here the frontend will refer to the interface with the associated L2 caches. The backend will refer to the interface with main memory.

[5]In the initial design, there existed a separate data bank module through which the FSM made all data requests. However, this added an unnecessary cycle of latency to all requests that either read or wrote. (We needed a cycle to pass the request parameters to the separate module.) In order to reduce latency, we remove the separate data bank and integrate the data storage directly into the FSM.

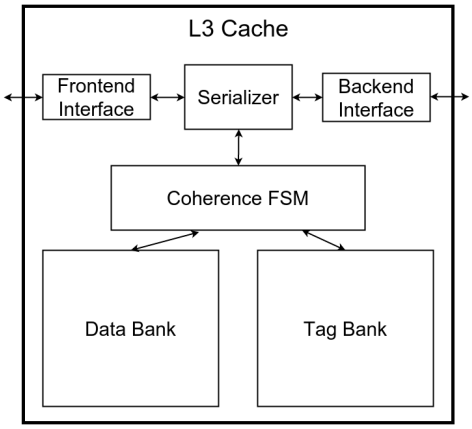

Figure 7: We see here the basic architecture of the L3 cache. It is extremely similar to the architecture of the L2 cache shown in Figure 3. We send an invalidate as a usual message through the NoC so we have no need for a separate process responsible for handling invalidations.

Flushing the Cache.

This feature has not yet been implemented for the L3. There has been difficulty surrounding devising an algorithm to do this correctly and efficiently.

6.1 Coherence FSM

As with the L2 cache, a bulk of the logic of the L3 exists in the coherence FSM. The L3 coherence FSM serves as the directory for the MESI Protocol[2]. The coherence FSM has the responsibility of ensuring the tag and data banks are consistent with each other at the conclusion of each transaction. The interesting edge case we see with the L3 directory but not the L2 cache is the case of multiple read or write requests while the line is in an intermediary state.

6.1.1 Intermediary States

Because, like the L2, we generate a set of states such that each action of the FSM (selected by the message–coherence-state) is non-blocking, we introduce multiple intermediary states to the basic MESI protocol. We then must handle the case where multiple L2 caches request a line in the intermediary state. We subdivide these cases into those that require resolving multiple reads, multiple writes, or both reads and writes. Each of these cases must be handled differently in order to retain coherence.

Only Reads.

The easiest case to resolve is the case where we've accumulated only readers. Because multiple readers of the same line are permitted, we simply share the line with those prospective readers.

Only Writes.

We then move to the case where multiple writers have requested access to a single cache line. When we see this, we simply allow one cache access to the line in the modified state. We then send a series of forward messages so that the line will be passed between caches until all writers have written. At this point, we return to normal operation with a single owner. This will ensure that all requested writers receive write access to the given line. This may be important in programs written using, for example, a multiple producer, single consumer model. We may want one producer to be disallowed from writing twice before another has written once.

Reads and Writes.

This case is the most complicated as we must allow all caches access, deciding an ordering for reads and writes. One possible ordering is allowing all reads access concurrently and then allowing writes to proceed as they would in the writes-only case. In this ordering, every cache is able to complete their operation, though the ordering may not be fair. With this ordering, it is possible that a single reader could request access just before the data returns, while many writers have been waiting, and then be the first to see the data.

It is important to note that no ordering can lead to starvation here. This is because we are only determining an initial ordering for each waiting core to make one operation each. After making one operation, a core will pass the line to the next core. If a given core needs to make multiple operations, it will simply send another coherence message asking for permission and be inserted at the back of the queue that we have created, where it will eventually be given access.

6.1.2 Dirty Bit

In order to implement a write-back L3 cache, we must store a dirty bit in the metadata. The L2 always writes back when a given line leaves the modified state[6], whereas the L3 need not always do so. A line in the modified state could be evicted from an L2 and written back to the L3, where it returns to a non-modified, valid state. Because we'd like to limit the traffic to main memory we will not write the line back immediately. We instead write the line back only when it is evicted or flushed from the L3.

6.2 Tag Bank

The architecture of the L3 tag bank is largely the same as the L2 tag bank. The primary difference is the metadata stored for each line. We must now store data such as an owner and a sharers list. The eviction

[6]In the sense that the line is sent to some other L2 or the L3 directory

	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.

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

Table 2: Description of L2 Coverage Metrics. We define explicitly untested code to be (control input, coherence 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, coherence state) pairs are designed to fall through to a `default` clause because they are unexpected.

process is identical; however, we do not yet implement flushing, though we keep the appropriate interface to allow for ease of future work.

6.3 Data Bank

The L3 data bank remains exactly the same as in the L2; the L3 is simply larger than the L2 as it has a greater number of ways.

7. VALIDATION AND VERIFICATION

An integral part of the development process was validation and verification. Each new module was developed alongside a test bench. As new features were added to a module (the L2 tag bank, the entire L2, the L3 tag bank, and the entire L3), so were those new features added to the corresponding test bench. The test benches served two purposes. Primarily, they served as tests on new features, allowing for discovery of bugs in the implementation of the module. Secondly, they served as regression tests. If the implementation of a new feature broke a previously-working feature, the test benches would raise an error.

Validating and verifying the design of the L2 cache served as a large portion of a project for Prof. Michael Theobald's course on Formal Verification. A brief summary of those results are included below.

7.1 Test Coverage

The first metric analyzed was test coverage. Developing the tests alongside the DUT itself led to very high levels of coverage. We see in Table 1 the ultimate test coverage (both line and branch) on the tag bank as exercised through two test benches.

We also analyze the coverage of the full L2 cache (excluding the tag bank module) when exercised with the associated test bench. We see high (though not as high as the tag bank) levels of coverage. This stems from the greater complexity of the entire cache compared to the tag bank alone. These results are presented in Table 2.

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

Table 3: Overview of latency bounds (cycles) for specific types of requests. The two General-type requests show the different bounds that are provable with differing levels of effort.

7.2 Model Checking of L2 Tag Bank

In addition to the directed test bench, we also apply model checking techniques to the RTL implementation of the tag bank. (The process of generating this implementation is given in more detail in Section 8.) We focus on latency-bounding properties. We begin with an attempt to bound the latency of any arbitrary request. After this, we attempt to create a tighter bound on requests that hit the cache. As we see in Table 3, as we make more constraints toward the common case of a hit, we see lower bounds. Bounding a read hit to 3 cycles is a great success for our tag bank as at least one request to the tag bank is required per request to the cache as a whole.

The validation plan of the entire L2 consisted only of the directed test benches. The coherence FSM was far too complex to complete a formal model-checking in a reasonable time. Even with the small tag bank, we started to see clear effects of the state space explosion problem.

8. HIGH LEVEL SYNTHESIS

Both levels of the cache hierarchy were designed using a high level specification language. This was chosen to allow faster design and prototyping of the architecture. The work proved to be an interesting challenge for HLS because the focus of the design was on creating a low latency solution. When hyper-optimizing a design, we might normally turn to directly creating an RTL implementation. In this case, however, we begin with an HLS specification and rely upon our compiler and HLS knobs to generate an optimized, low-latency RTL implementation.

In order to do this, we must identify bottlenecks in the design. The most obvious was the tag bank module. Each request to the cache required at least one tag lookup, with many requiring two or three. Thus optimizing this would allow us to reduce the latency of all requests to the cache.

Our ultimate goal is generating an implementation that serves a read hit in 10 cycles and a write hit in fewer.

8.1 Tag Bank

The initial RTL implementation of the tag bank took more than 10 cycles to process even a read hit. Clearly we would need to improve this metric were we to meet our 10 cycle read hit goal for the entire cache. We analyze the generated Control/Data Flow Graph (CDFG) for the source of our unnecessary latency (See Figure 8). Immediately we realize our mistake in only providing one read port for the tag bank memory.

8.1.1 Lookup CAM

By modifying this memory to contain eight[7] read ports, we are able to create the intended CAM structure. Shifting to a CAM structure allows us to have a one-cycle lookup to determine which way, if any, stores the given tag. This adjustment brought great improvement in overall latency. However, we still see a higher latency than expected. Again, because this module is on the critical path of every transaction, we dedicate time to optimizing it further.

8.1.2 Nuanced Optimizations

The subsequent iteration of tag bank includes a design that attempt to constrain specific functional blocks to a given latency. We see our HLS tools respecting the latency constrain of these blocks; yet, they are not chained together with the inter-block latency we would like. Finally, we realize that the easiest solution is to schedule the design by hand. Especially cognizant of the critical path, we make the read hit path as fast as possible and then afterwards make other cases quick.

[7]One for each way

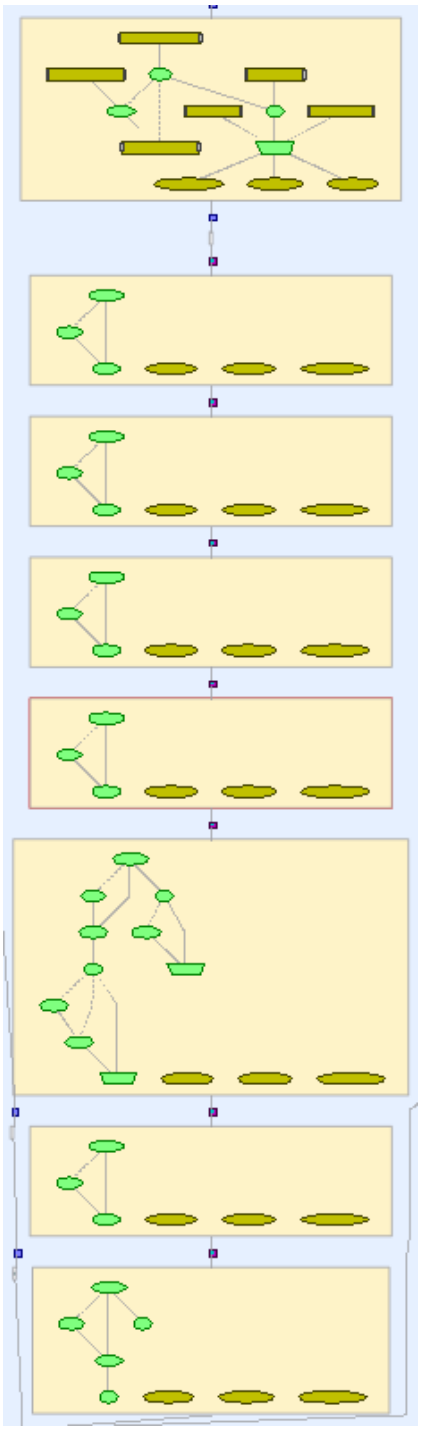

Figure 8: An eight-cycle excerpt from the initial CDFG of an 8-way associative implementation of the L2 tag bank. Each yellow oval represents a read from internal memory. We note the reads spread through 8 cycles (yellow boxes) because the internal memories have only one read port. This is not the CAM we were looking for.

We contrast an excerpt from the final CDFG (Figure 9) from the initial CDFG (Figure 8).

8.2 Data Bank

After we've optimized the tag bank to our satisfaction, we begin exploration around making the common case fast. That is, we want to make read and write hits as fast as possible, ignoring the area cost parameter[8]. As such, we want to make data accesses as quick as possible. We therefore choose to eliminate the separate data bank module present in the original architecture. We can save a cycle by accessing the data memory from the FSM instead of some intermediary. Ordinarily, we might choose to preserve this type of modularity at the cost of a cycle. However, the isolation of this small amount of logic is not worth using 10% of our latency budget.

8.3 FSM

After addressing our easy targets for latency optimization, we move to hyper-optimizing our common case: read and write hits. We begin by analyzing the cause of unnecessarily high latency. Analyzing the generated CDFG, we see that our common case critical paths share operations with less common cases[9]. Our attempts at optimization, then, will center around isolating our critical paths from the uncommon paths.

For this, we begin with similar approaches as optimizing the tag bank: selecting certain portions of the HLS specification and manually scheduling them. This eliminates some unnecessary latency but not all. We must resort to restructuring the code in order to conform to the tool's abilities to constrain latency of given blocks and sections of code. This restructuring knowingly degrades performance of the uncommon cases. We accept this as penalty for optimizing the common case.

After this constraining, we find the latency is still a few cycles higher than expected so we examine a cycle-by-cycle trace of the cache processing a read hit. Here we find that the interface and serializer processes each inexplicably took 2 cycles to process data[10] rather than the expected 1 cycle. We constrain these in the HLS process and finally see a latency in the expected range.

[8]We are fully on one end of the Pareto curve. The primary duty of a cache is to provide a fast lookup so latency is our primary parameter. Additionally, a large majority of the area of a cache is the data itself. Therefore, we can see a large reduction in latency in exchange for a small proportion of additional area.

[9]This is not too unexpected as a more moderate location on the area-latency Pareto curve is often an acceptable design. Additionally, current capabilities of the tools do not allow us to specify certain code *paths* as critical paths that ought to be optimized for area, speed, or power. We can only optimize *blocks* of code in this way.

[10]Maybe because we insulted them and called them boring back in Section 5.

	Eventual	Initial
Read Hit	12	41
Write Hit	4	28
Read Miss	46	82
Write Miss	40	72

Table 4: Latency (cycles) of given operations for both the initial and eventual RTL implementation. We note that reads are blocking and writes are non-blocking. Miss times are exclusive of processing time for any lower-level cache.

8.4 Summary

Through tuning the HLS knobs and restructuring the HLS specification, we are able to optimize the RTL implementation of the L2 as shown in Table 4. The initial implementation took 41 cycles for a read hit and 28 cycles for a write hit while the eventual implementation takes only 12 cycles to process a read hit and 4 cycles to process a write hit. In order to do this, we make full use of the HLS knobs provided, focusing on the performance of the critical paths, while allowing the tools to generate an implementation for the less-common paths. With more architectural optimizations, we could potentially see small decreases in latency, but we accept this as a reasonably optimized design.

8.5 Advantages of HLS

When developing with a high level specification language, we are able to make the code quite extensible and parametrized with ease. We have one header file that contains all parameters pertaining to the size of the cache, the size of a block, the size of control signals, etc. We can easily modify this header file and regenerate a differently sized design.

Making the architectural changes mentioned above also proved much easier due to the higher level of abstraction of implementation. Reworking interfaces happened with little effort. The control flow was reorganized with ease. While the higher level of abstraction allowed for easier bigger picture optimizations and refactoring, it also meant that specific optimizations were more difficult[11]. Overall, implementing the design in a high level specification language resulted in a much quicker development time[12].

9. MEMORY HIERARCHY TESTING

Once RTL implementations were generated for the L2 and L3 caches, work began on creating a test bench for an entire memory hierarchy. Previous testing had

[11]Compare this with it being much easier to generate intended machine code with C than Python.

[12]Also, the author is much more competent with SystemC than SystemVerilog.

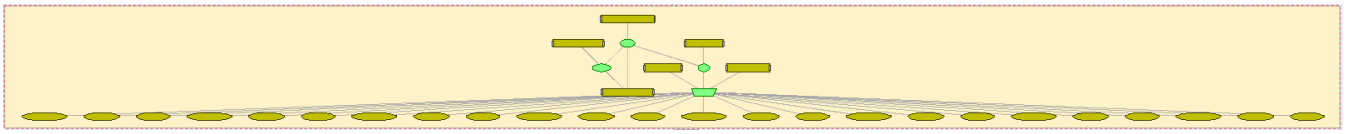

Figure 9: A one-cycle CAM lookup from the final CDFG of an 8-way associative implementation of the L2 tag bank. Each yellow oval represents a read from internal memory. We see 24 ovals because each CAM lookup reads three fields for each entry. These reads all happen in the same cycle.

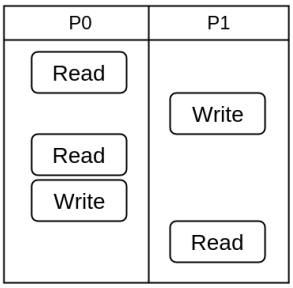

Figure 10: An example of a basic test used to test the proper implementation of the coherence protocol. We see an alternating series of reads and writes from two processors sharing the LLC.

shown that each individual module worked according to expectations. However, integrating the modules and introducing race conditions by having multiple L2 caches would test that the individual protocols worked well together.

9.1 NoC Interface

As the test benches were also written with a high level specification language, we needed to generate a basic high level specification for the NoC interconnect to allow the L2 caches to communicate with the single L3. The implementation of this interconnect included some features of the expected RTL wrapper.

9.2 Complexity of Test Bench

Given the current implementation state, we test only basic multi-processor scenarios. While we do not test the implementation of most intermediary states, we do verify that multiple L2 caches are able to share cache lines appropriately. The sequence shown in Figure 10 works as expected. Each cache sees the correct value of previous stores. Additionally, we see that this test bench passes using the high level test bench in conjunction with the RTL implementation of each cache.

10. FUTURE WORK

While a majority of the functionality of each cache has been implemented, there still remain some unimplemented edge cases, mostly concerning concurrency. We list future work and descriptions below.

10.1 Flushing

While flushing is a feature implemented in the L2, the L3 implementation does not currently support a full-cache flush. The main obstacle to this implementation is the transition from a private to a shared cache.

With a private cache, it was acceptable to stall the frontend while the operation completed. It is much less desirable to stall the entire memory hierarchy if a single processor requests a flush, especially since that flush may only have the intention of writing back a single buffer to main memory.

The second issue preventing trivial porting of the L2 algorithm to the L3 is the loss of the invariant that only those lines in the modified state need to be written back to the cache. We worry about the situation where processor A modifies a line and then processor B also modifies the same line. Without tracking a line's history we cannot know if the flush-requesting processor has modified the line.

Our safe option, then, is to flush the entire memory hierarchy to main memory, which is a very expensive operation. This operation becomes more expensive as the heterogeneous architecture scales, undercutting our attempt to implement a scalable cache coherence algorithm.

10.2 Concurrent Requests to Invalid Line

Because we attempt to avoid stalling the L3 whenever possible, we introduce intermediary states that the cache line enters while waiting for data to be fetched from main memory. Handling requests while in these intermediary states is not fully supported. We must decide on an ordering and priority for reads and writes. A possible implementation is given in Section 6.1.1. We handle the case of multiple readers as well as one writer with multiple readers. All other cases remain unhandled.

10.3 L2 Forward Plane Stalling

We must occasionally stall the forward plane. Causes of these stalls, and a suggested implementation are described in Section 5.1.3. This feature is implemented yet untested. The untested implementation limits our

ability to introduce concurrency into the test bench for the full memory hierarchy.

10.4 AMBA Wrapper

Each cache (L2 and L3) was designed with an interface providing the necessary information to complete a transaction with either an AMBA bus or the NoC (as applicable). However, the timing of the protocol at the edge of the caches does not match that expected at the AMBA or NoC interface. For that reason, we must develop a simple wrapper that converts between the cache interface and the expected interface. We expect to directly use an RTL implementation of this wrapper.

10.5 Integration with Accelerators

The approach thus far has been to coordinate and ensure coherence between the processors of the system. In order to use ESP fully, we must be able to guarantee the coherence of memory used by accelerators via DMA. This could be acomplished in two major ways. The first possibility is properly and fully implementing a flush capability for the LLC. The accelerators will then use DMA to access memory directly. We additionally consider allowing the accelerators DMA access to the LLC (as presented in [1]), at which point the accelerators will also interact with the already-existing coherence directory.

10.6 Race Conditions

We also have not yet handled a few very uncommon (yet possible) edge cases. Many of these result from a limited depth of buffers. That is, deeper buffers would allow more concurrency yet dramatically increase the complexity of the architecture.

11. CONCLUSION

We have described an architecture than can be used to implement a scalable cache coherence protocol on an Embedded Scalable Platform instance. We add to a current architecture a private, unified L2 as well as a shared L3-directory. We've used a high level specification to generate an RTL implementation of two different caches. We made heavy use of available HLS knobs in order to create a latency-optimized design. Our design achieves a desirable low latency, serving read hits in 12 cycles and write hits in 4 cycles. In order to fully move our implementation to that end of the Pareto curve, it was necessary to rearchitect portion of the design.

During active development, we constructed a directed test bench. Development of this test bench proceeded alongside development of the caches themselves. We examined various coverage metrics in order to begin to validate our design. Additionally, we utilized formal verification techniques in order to prove correctness of portions of the design. We were also able to produce latency bounds on certain subsets of the cache.

Finally, we created a test model of our additions to the existing cache hierarchy. We specify two processors for a single shared LLC and validate the operation of basic uses of the coherence protocol. This successful simulation was run using the generated RTL implementation of both caches.

Creating this scalable memory coherence protocol will allow further development of Embedded Scalable Platform instances as we will be able to more fully utilize the power of heterogeneous computing for quick and power-efficient computation.

12. 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] D. J. Sorin, M. D. Hill, and D. A. Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 6(3):1–212, 2011.