4 Consistency and Coherence

TODO: Chapter introduction.

4.1 Cache Effects

This section is a discussion on how the listed cache effects affect memory coherence and consistency properties. [This section might be moved towards the end of this chapter, since the coherence models have not yet been described.]

Memory Consistency: TODO

Inclusion Policy: This property is an attribute of a multi-tired memory hierarchy. The choice of an inclusion policy is design dependent, maximising total caching capacity necessitate an exclusive policy, a strictly inclusive policy requires lower level caches such as the L2 to hold all data present in the L1 caches. A non-inclusive policy permits intermediate behaviour between strictly inclusive and excursive policies. Shared memory based coherence protocols benefit from a strictly inclusive policy, as shared memory is always aware of any private cached data. The BERI Directory model relies on a strictly inclusive policy. The Time-Based model does not require any explicit coherence messaging, any policies is acceptable, however, since we do not enforce exclusive behaviour, caches follow the non-inclusive policy.

Associativity: The storage location of a memory entry is determined by the replacement policy, ranging from direct mapped to fully associative. Maintaining coherence is simpler is direct mapped caches as all addresses will have a fixed location. Set associative or fully associative designs require explicit cache lookups to find any

matching data. Directory coherence is not directly affected by private cache associativity, some design dependent address lookup overheads might be present. Similarly, Time-Based coherence is not affected by associativity as coherence is only enforced when data is loaded or stored.

Line Eviction: When a cache runs out of free/invalid memory locations in order to store new data, the replacement policy dictates where the new data will be stored. As already established, direct mapped caches have a fixed policy. Set associative caches can use a number of techniques to identify stale lines or pick random location to place new data. The old data is replaced or evicted. Directory coherence is affected by capacity misses in shared memory as it results in coherence messaging, Time-Based coherence is unaffected as no coherence messaging is used.

Virtual and Physical Addressing: Various addressing modes can be used for caches in order to improve performance, virtual addressing may not require TLB lookups where as physical addressing does. Coherence mechanisms benefit when shared caches and private caches use a common addressing mechanism, coherence messages are much simpler and cause fewer side-effects. All BERI caches are physically addressed, directory coherence messages carry the physical address and Time-Based coherence does not require messages.

Cache Block Granularity: BERI caches use 32 bytes per block/line. Each block shares a common valid bit and tag field. An access miss to any word will result in a full eviction. With the exception of aliasing which is discussed further, large blocks are beneficial to both Directory and Time-Based coherence. The memory metadata overheads are lower with a larger block size and coherence messages are more efficient as well. The tag-time-counter used in Time-Based coherence has a smaller overhead as compared to a cache with a smaller block size.

Line Aliasing: Software accessing more than one data location mapping to the same cache line results in aliasing. Aliasing is particularly hazardous in virtually mapped caches, since two virtual addresses may point to the same physical address. Physically mapped caches are accesses through unique physical addresses, and aliasing usually results in the eviction of current cached data. Contention for the same cache line results in a degradation of performance and potential side-effects due to coherence. In coherent systems such as ones based on a global directory, memory aliasing can present a performance disadvantage. The directory must maintain an

up to date record of any private caches sharing memory locations, when a memory line is evicted, the record must be updated as well. However, any sharers of the evicted line must be notified that the line is no longer in shared memory, failure to notify sharers will result in caching of stale data without the knowledge of the directory. This issue is also related to the inclusion policy as designs may follow different behaviours. Aliasing in an inclusive cache hierarchy, such as the one described, will result in unwanted but necessary coherence messages. Capacity misses in the shared cache will also result in evictions and coherence messages. This case has been observed in our Directory based BERI design. The Time-Based coherence design is not penalised by the side-effects of aliasing memory, as there is no explicit coherence messaging. Additionally this coherence model allows non-inclusive behaviour due to the absence of any sharers records. While Time-Based coherence has many drawbacks due to synchronisation and counter overflows, it does not suffer from aliasing overheads.

False Sharing: A subset of memory aliasing is false sharing of memory lines. It occurs when two memory words within a cache line are accessed independently, there is no direct sharing of data, however, the coherence mechanism must keep track of any updates, particularly when the two words are accessed by different private caches. False sharing mostly results in performance degradation in the case of a Directory design. as with aliasing, the Time-Based model is not affected by false sharing.

Memory Operation Reordering: Memory access reordering must behave according to a selected consistency scheme. TSO requires a strict store ordering, where as RMO permits interleaving of memory accesses. The TSO based directory coherence model must obey store ordering, necessitating precise and timely coherence messaging. The Time-Based coherence model uses RMO, requiring less intervention. Our cache design is fairly simple and does not offer memory access reordering within a core and its private caches, however, shared memory accesses may be out of order.

Prefetching: Compulsory cache misses can be reduces by fetching an entire block when a memory word is requested. Block prefetch is beneficial for both Directory and Time-Based coherence. The sharers list is only modified once when a cache accesses any data within a block in a Directory and the tag-time-counter is set once, when the line is loaded for the first time.

Alignment: Any memory access that requires data from more than one cache line is known as an unaligned access. Most modern systems permit unaligned accesses, the MIPS model used in BERI has a very limited support using special instructions. Since BERI does not permit accesses across cache line boundaries, we do not deal with this issue, however, as a general argument; aliment presents a challenge for both Directory and Time-Based coherence. The Directory will require multiple sharer list accesses and the Time-Based model will require updating more than one tag-time-counter. Additionally mismatched tag-time-counters may exist where one of the two lines was evicted and then reloaded. Coherence guarantees will drop unless the tag-time-counters are more accurately tracked.

Cache Instructions: Most processors support some form of cache instructions, designed to allow cache cleaning, usually by the OS. In MIPS R4000, the cache instructions can target the L1 or the L2 caches. However, the effects of these instructions on coherence are not specified. In order to maintain the strict inclusion policy, the Directory sends coherence messages to the L1's whenever a cache instruction evicts a shared line (applies to a single sharer scenario). Time-Based coherence does not require strict inclusion, so L2 cache instructions have no effect. Respective consistency policies of the coherence schemes must also hold for cache instructions, this behaviour has been tested for both coherence models.

Invalidate vs Update Coherence: This property only affects coherence models based on messaging. A simple way to notify sharing caches of data modifications is through invalidate messages, the data location in the private cache is marked invalid and updated when it is accessed next time (may not be accessed again). An alternative is to send a message containing updated data and then modifying the private cache line to match shared memory. The BERI Directory uses invalidate style messaging, allowing the short tags optimisation and future proofing any scalability issues [TODO: Reference]. An update mechanism has also been tested, it performs quite well but for simplicity it was not used in the coherence comparison tests (TODO: Is this item necessary at all??).

Write Buffers: Caches act as large write buffers but many modern systems have an additional layer of buffering for improved memory accesses. Memory consistency mostly dictates the behaviour of write buffers. A TSO model would require correct write propagation through the write buffers. An RMO model allows a more relaxed

behaviour, but the write buffers must propagate on synchronisation. BERI caches do not use write buffers, some buffering is present in the shared bus, but specified consistency models are guaranteed.

Page Boundary Crossing: As previously specified, BERI caches are physically addressed, so page boundary crossing is not an issue. If the page reference is absent in the TLB, a page fill is requested. Neither coherence mechanism is affected by this issue. (Is this required?)

Write Hit Policy: There are largely two write hit policies: write-through and write-back. A write-through policy updates the current cache and propagates the update to a lower level of memory. Write-back caches mark the line as dirty and hold the copy until the line is evicted or explicitly requested through coherence. BERI L1 caches are writethrough. Both coherence schemes benefit from this behaviour. (TODO: Further)

Write Miss Policy: On a write miss, the cache can either request the line from a lower tier of memory and update it with new data (write allocate), or simply pass the write to a lower level of memory and leave the current cache untouched (Non Write Allocate). BERI caches use the non write allocate policy. Directory and Time-Based coherence can accommodate both policies natively. A write allocate policy would perform a load from shared memory and a directory would register the requester as a sharer, Time-Based model would assign a tag-time-counter to the loaded and updated line.

4.2 BERI Time-based Coherence

Correctness of the Time-Based coherence model analysed in this thesis relies on three key implementation elements: cache self-invalidation, barrier instructions, and lock-free instructions. Unlike many other coherence schemes, coherence is controlled within the data caches. Most other hardware coherence implementations focus on a bottom up approach, where coherence is dictated by the last-level cache (LLC) or a dedicated memory controller.

In this implementation each data cache has a dedicated time-counter. This counter governs the cache self-invalidation policy (optimised versions of the protocol use multiple counters). The counter is incremented every cycle when the cache

is in operation, however, there are two exceptions: cache initialisation and a synchronisation (SYNC) instruction. A cache flush is triggered when the counter rolls over, this is done through the cache initialisation mechanism, all memory access to the cache are blocked during flush.

A data cache line contains Data words and Tag bits, these are stored in separate memory blocks. A counter value is added to the Tag bits, this tag-time-counter (TTC) dictates the lifespan of the cache line. The TTC value is assigned when a line is cached for the first time. The TTC value is assigned a value equivalent to the sum of: current time-counter value and a fixed offset. A range of offset values can be selected, in the default case of the protocol, the offset value is fixed at compile time. When a cache line is requested and the Tags are valid, the TTC is compared to the current time-counter value. The lifespan of the line is deemed expired when the time-counter is greater than the TTC. If the operation is a load then the line is re-fetched from a lower level of memory, a store operation causes a line invalidate (Note: this behaviour does not apply to Store Conditional instructions). Once a line is loaded into the cache and the TTC is set, its value is fixed until the cache line is either evicted or expired.

4.2.1 Optimal Time-Counter Size

Caches improve overall performance by exploiting spacial and temporal locality of data. Software data structures are often stored contiguously in the memory. Repeated operations on data can be significantly improved through caching. Caches are designed to hold valid data for as long as possible, and cache design improvements mostly focus on this aspect. Thus, choosing an appropriate cache line lifespan in a Time-Based coherence cache is non trivial. There is a constant tradeoff between miss rates due to: counter overflows and cache capacity overheads. In order to retain the benefits of data caching, the lifespan of each data line must be long enough to maintain a low miss rate, as well as short enough to allow stale data eviction. The Time-Based coherence model does not provide fixed line eviction guarantees, the operating system can not directly observe cache line time-outs, software must use correct locking structures and barriers to ensure data sharing.

TODO: Figure showing how counters work

In the evaluation of Time-Based coherence we will see how the choice of cache line time-outs affects system performance. Holding a line in the cache isn't always beneficial, partly due to the way the OS deals with spin locks and other locks (TODO: Explain FreeBSD locking mechanism). If a lock can not be acquired within

a set amount of time, the OS simply schedules another thread. This is mainly the reason why Time-Based coherence works comparably well.

Why not eliminate cache time-outs all together? While it is true that coherence is guaranteed only when correct software primitives are applied, cache self-invalidates allow some data propagation, thereby reducing the risk of deadlocks. I discuss the concept of SYNC-only coherence further (TODO: Section needs to be added), and deadlock avoidance is discussed in Section 4.2.9.

I have studies several different D-Cache time-counter sizes for Time-Based coherence. Larger counters are beneficial as they provide a finer timing granularity, at the cost greater logic overhead.

4.2.2 TTC Memory Overhead

Figures 4.1 and 4.2 show hardware cache line overheads between different TTC values. Figures 4.1 and 4.3 compare Tag overheads with increasing cache capacity. The counter size must be carefully considered when synthesising these caches on an FPGA, the block RAM (BRAM) design will greatly affect optimisation. Odd sizes could necessitate the use of multiple BRAM's for storing each Tag, this tradeoff must be considered. ASIC designs are more flexible, arbitrarily sized Tag's, Data lines, and other components can be produced (TODO: discussion and verification of these facts).

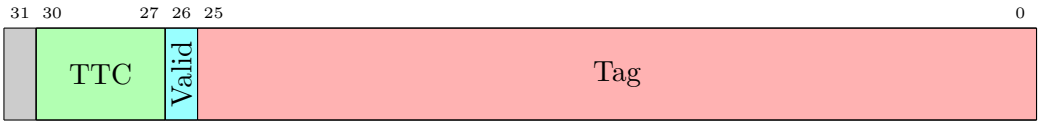

Figure 4.1: D-Cache Tag's, short TTC (16KB Size)

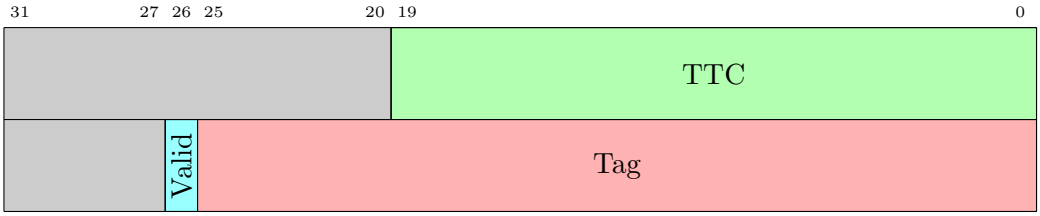

Figure 4.2: D-Cache Tag's, long TTC (16KB Size)

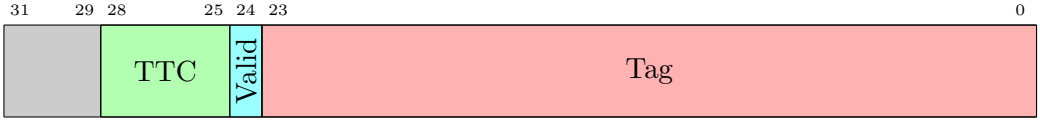

Figure 4.3: D-Cache Tag's, short TTC (64KB Size)

Short counter values (Figure 4.1) result in an overhead of 4 bits per line, however, much coarser time granularity is used for self-invalidation. Figure 4.2 shows a larger TTC overhead (20 bits), this design allows much finer time granularity and results in better benchmark performance (discussed in the evaluation chapter).

Our RISC processor requires only 40 bits of physical address space. When the cache size is increased, fewer Tag bits are required as fewer address bits are stored as Tags. Figure 4.3 shows this effect when the cache size is quadrupled. An additional 2 bits are used for cache indexing, the Tag is shrunk by 2 bits. This can be beneficial for Time-Based coherence, since a larger TTC can be used and relative storage overheads are lower.

4.2.3 Load Linked and Store Conditional

Our LL/SC model requires propagation of these instructions to the last level cache (LLC), L2 cache in this case. In order to achieve this, a load linked (LL) instruction is treated as an uncached access in data cache. This ensures that an updated LL value is always fetched.

Store conditional (SC) instructions check if the desired line is present in the data cache, the SC data is written through to the LLC. Additionally the cache line is invalidated on hit, this is necessary due to uncertainty in the outcome of SC. The LLC determines SC success or failure, the result is forwarded through the data cache to the Writeback stage. The LL/SC instructions use separate registers to check data validity, no additional Tag bits are required. L2 Tag overheads will be discussed further in Figure 4.17.

4.2.4 SYNC Instruction

This instruction performs two operations in the data cache. The time counter is incremented on SYNC, ensuring that all stale data will be treated as invalid, it can also be used as a single instruction full cache flush. Note that if the SYNC instruction causes the time counter to overflow, then the cache is reinitialised as specified earlier. The second property of this instruction is to ensure that all loads/stores have been propagated to the LLC. This is achieved by performing a special write to LLC, this memory access has no side-effects. The operation is expected to return a response back to the data cache. The response guarantees load/store propagation from the given core. The response is ignored by the pipeline as SYNC is not expected to respond.

Instruction caches do not self-invalidate, coherence is not necessary for instructions as long as self-modifying code is not executed.

4.2.5 Trace Format

The trace syntax has the following format [TODO: Reference - A checker for SPARC, Matt N.]

$$C_{id} : \; M[M_{addr}] \coloneqq \mathbb{N} \tag{4.1}$$

C_{id}:	Core/Thread ID
M[ ]	Memory operation (SYNC is a variant)
M_{addr}	Address of the memory operation
$\coloneqq$	Memory operation type Load (==), Store (:=)
$\mathbb{N}$	Natural number stored/loaded

A sequence of atomic instructions is encapsulated in angle brackets separated by a semicolon: $\langle$ operation 1; operation 2; ... $\rangle$. This representation is used for LL/SC operations where a load linked operation is followed by a store conditional operation. Each memory instruction executed by the checker follows this format. The checker evaluates the outcome of each instruction and verifies the behaviour. The behaviour must obey a selected memory consistency format, otherwise the test sequence fails.

4.2.6 AXE Litmus Tests

Appropriate references for the tests are required. Also references for the PPCMEM claims. Appropriate test labels should be used rather than the raw test names *.axe

Litmus tests are a common way for architects to evaluate the memory consistency model for a given design. The tests are typically very short parallel memory operations that identify subtle variations in memory consistency. The test evaluates all permutations for a given set of memory instructions.

Due to the unique behaviour of Time-Based coherence, the system behaviour is different to other relaxed models. While systems such as PowerPC state a number of memory consistency properties, not all of these are visible in hardware. This is due to the effects of cache design, memory latency, ordering, and many other

factors. Time-Based design demonstrates some of the memory orderings not visible on PowerPC, these are discussed below. A number of memory orderings are not observable on Time-Based BERI multi-core, these are also shown below.

4.2.6.1 Non-Observable Relaxed Behaviour

LB+addrs.axe In this example (Figure 4.4), both threads attempt to load values of (x) and (y) at steps (a:) and (c:) respectively. Both load operations are blocking, no memory operations will be submitted into the memory sub-system until the conditions are satisfied. Steps (b:) and (d:) perform stores to (y) and (x) respectively, these values satisfy the blocking conditions. The nature of blocking instructions will prevent this sequence of instructions and threads ever succeeding unless the memory sub-system ignores blocking operations and allows load/store reordering. The Time-Based coherence model does not prevent this scenario, however, the cache structure of BERI does not support out of order execution of memory instructions or memory reordering.

```
*  Memory Operations  *
0: M[0] == 1 (blocking)
0: M[1] := 1
1: M[1] == 1 (blocking)
1: M[0] := 1
```

po - program order	dep - dependency(blocking)	rf - read-from

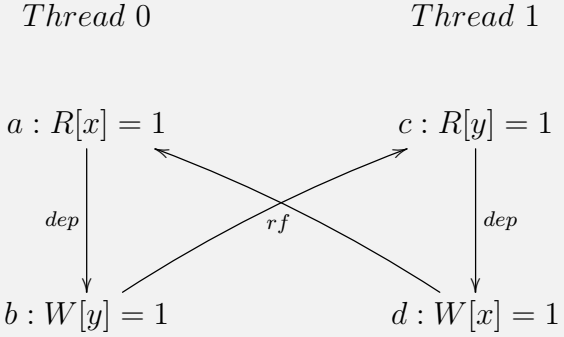

Figure 4.4: LB+addrs.axe

LB+sync+addr.axe The example shown in Figure 4.5 is similar to code sequence discussed above, however, the blocking instruction for Thread 0 has been replaced with a SYNC instruction. While blocking instructions are largely implementation dependent, SYNC instructions must provide stronger ordering guarantees. SYNC must guarantee that all preceding memory operations (specifically store's) have been completed. Note that SYNC instructions only apply to the executing thread and are not expected to propagate memory operations from other threads. In this example the update of (y) at (b:) occurs after a SYNC operation, step (c:) is a blocking operation that depends on the stored value. Hence, if step (c:) has succeeded then it would indicate that the operation at (b:) has propagated to shared memory and an updated value of (x) was observed at (a:).

```
*  Memory Operations  *
0: M[0] == 1
0: sync
0: M[1] := 1
1: M[1] == 1 (blocking)
1: M[0] := 1
```

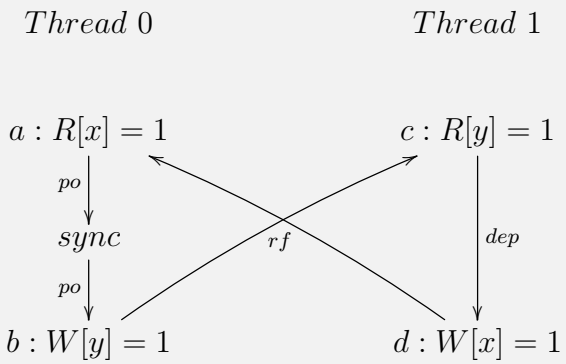

Figure 4.5: LB+addrs.axe

This example shows a cyclic dependency between all operations and barring a bug in the hardware implementation, this scenario should not be possible on any system (TODO: Too Strong?). This example is stronger than the one described above.

4.2.6.2 Observable Relaxed Behaviour

Testing of Time-Based BERI has revealed several relaxed consistency scenarios not observed on the PowerPC memory model. These are listed below.

MP+sync+addr.axe This scenario is observable on the Time-Based coherence model but not on PowerPC. Time-Based coherence allows stale data to reside in the cache and thus, loads to stale memory are allowed. The example shown in Figure 4.6 demonstrates the stated behaviour. Steps (a:) and (b:) on Thread 0 are both store operations, the former is propagated through the sync instruction. The stores update values of (x) and (y) respectively. Thread 1 at step (c:) depends on the store at (b:), the following operation at (d:) is a load of (x).

```
*  Memory Operations  *
0: M[0] := 1
0: sync
0: M[1] := 1
1: M[1] == 1 (blocking)
1: M[0] == 0
```

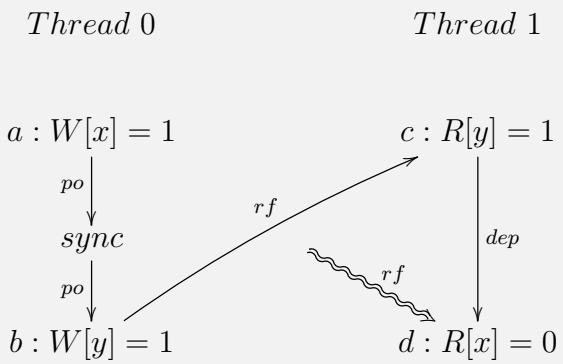

Figure 4.6: MP+sync+addr.axe

If we assumed a sequential behaviour of memory with no stale data caching, then the load of (x) at (d:) would always produce an updated value. On our Time-Based system, the load at (d:) can produce either the original or the updated value of (x), since the original value could have been in the private cache with a valid tag-time-counter. On the other hand, the load of (y) at (c:) could produce the updated value as the line could have expired in the private cache.

Systems that rely on coherence messages may not exhibit this behaviour as eager messaging will evict stale copies, however, lazy coherence messages along with reordering of messages might produce a similar outcome. This behaviour is certainly allowed by the PowerPC model but not observed in practice [TODO: References required, from Peter Sewell's papers].

WRC+sync+addr.axe Litmus tests are extendible to multiple threads. Figure 4.7 show a triple thread litmus test demonstrating coherence locality of threads/-cores. Thread 0 performs a store to (x) at (a:), which is observed by Thread 1 at (b:). Thread 1 proceeds to execute a SYNC instruction followed by a store to (y) at (c:). SYNC instructions only apply to the executing thread, if Threads 0 and 1 have observed a certain memory behaviour, the same can not be guaranteed for Thread 2 in this example. The conditions between steps (c:), (d:) and (e:) are identical to the previous example (MP+sync+addr.axe). As we have already observed, even if the load of (y) at (d:) yields the latest value, the load of (x) at (e:) may produce stale data.

```
*  Memory Operations  *
0: M[0] := 1
1: M[0] == 1
1: sync
1: M[1] := 1
2: M[1] == 1 (blocking)
2: M[0] == 0
```

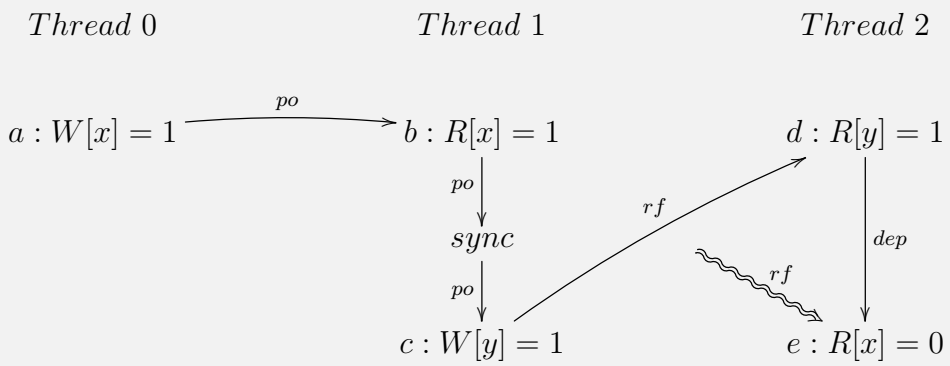

Figure 4.7: WRC+sync+addr.axe

W+RWC+sync+addr+sync.axe Figure 4.8, show another example where two threads can observe each others memory operations, while the other thread's may not. Thread 0 performs two store operations at (a:) and (b:), the former enforced through a SYNC. Thread 1 performs two loads at (c:) and (d:), the later exhibits a dependency on the former. The interaction between Threads 0 and 1 is allowed and expected under BERI Time-Based coherence. Thread 2 performs a store at (e:), enforced by a sync and followed by a load of (x) at (f:).

```
*  Memory Operations  *
0: M[0] := 1
0: sync
0: M[1] := 1
1: M[1] == 1 (blocking)
1: M[2] == 0
2: M[2] := 1
2: sync
2: M[0] == 0
```

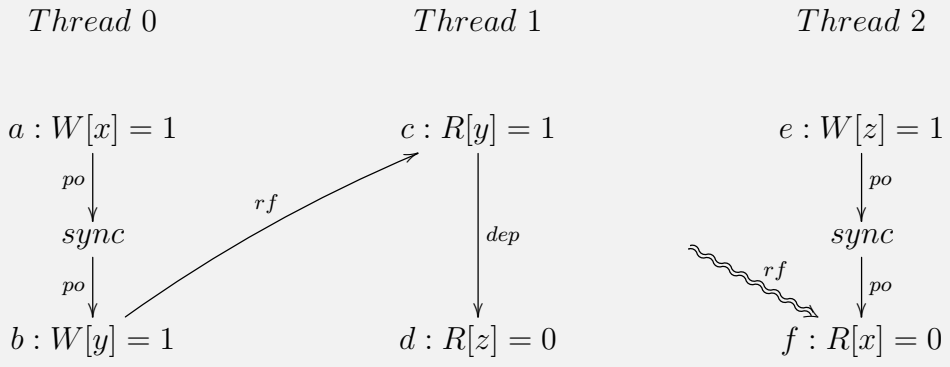

Figure 4.8: W+RWC+sync+addr+sync.axe

Since, Thread 2 is not explicitly dependant on any operations performed by either Thread 0 or 1, it can load the stale value of (x) from its private cache. Figure 4.8 makes the lack of dependency between Thread 2 and other particularly evident.

4.2.6.3 Forbidden Behaviour

So far I have shown evidence that the Time-Based memory model is relaxed and exhibits more relaxed behaviour than some commercial hardware [TODO: Reference]. The next example demonstrates that the Time-Based model can still provide programmer assurances.

MP+syncs.axe This example (Figure 4.9) shows how adequate use of SYNC instructions can provide strong programmer guarantees. Individual threads perform operations enforced through SYNC's. Thread 0 updates values of (x) and (y) at (a:) and (b:), and Thread 1 loads (x) and (y) at (c:) and (d:). If Thread 1 observes the updated value of (y) then it must not observe a stale value of (x), as the SYNC instruction will guarantee that all stale data has been evicted from threads private cache.

```
*  Memory Operations  *
     0: M[0] := 1
     0: sync
     0: M[1] := 1
     1: M[1] == 1
     1: sync
     1: M[0] == 0
```

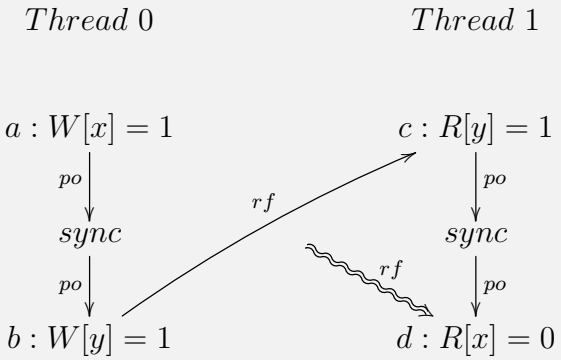

Figure 4.9: MP+syncs.axe

4.2.7 CHERI Litmus Tests

CHERI Litmus requires careful referencing as it is currently unpublished

The claims stated in the AXE litmus tests are backed up by CHERI litmus tests. Message passing (MP) with SYNC's has already been discussed in Figure 4.6, a similar scenario is demonstrated using CHERI litmus. We clearly observe different memory consistency behaviour depending on the use of SYNC's. The default litmus MP test does not use SYNC instructions, since Time-Based coherence allows caches to keep stale data until a cache line expires or a SYNC instruction, stores from another threads will not be observed. Two versions of MP1 test were created in order to observe all outcomes. The barrier implementation required by the test also plays a major role in the outcome. MP2 test uses a SYNC instruction between the two load operations of thread 1, this test mimics the example shown in Figure 4.9. Time-Based coherence guarantees the specified outcomes will not be observed.

Figure 4.10 shows the code sequence used in the MP1 test. The default test is executed without the SYNC instruction (line 4). In the modified test, the SYNC instruction in added into the sequence. Figure 4.11 shows the MP2 test. The two SYNC instructions (P0: line 3 and P1: line 2), ensure that this condition is never observed on Time-Based coherence.

The barrier loop implemented in CHERI litmus is shown in Figure 4.12. The SYNC instruction on line 6 is not present in the default case. The instruction has been added in order to observe the desired memory states as well as improve the execution time the test on Time-Based BERI (discussed in later sections). The barrier contains two loops: one is dependant on LL/SC instructions, and the second relies on shared memory updates. Since, lines will be locally cached until the timer expires, this loop could take a long time to execute, the SYNC instruction ensures that fresh data copies are fetched more frequently.

The test outcomes for MP1, MP1-mod and MP2 are documented in Table 4.1. The table shows the outcomes of a 1,000 iterations of the litmus tests in simulation. It is evident from the results that all test outcomes are only observable in MP1 and MP1-mod, and only in the case where the barrier implementation executes an extra SYNC in the second loop. This proves the relaxed nature of Time-Based coherence. The tests are highly timing sensitive and achieving all outcomes requires some test adjustments, however, no modifications were made to the hardware.

An interesting test outcome is when the default MP1 in executed together with the default barrier implementation, the desired registers both read initial values as

```
*       Initial Conditions      *
{0:r2=x; 0:r4=y; 1:r2=x; 1:r4=y;}

         P0      |       P1
1:  li r1,1      |  lb  r3,0(r4)
2:  sb r1,0(r2)  |  lb  r1,0(r2)
3:  sb r1,0(r4)  |
4:  sync*        |

    Exists (1:r3=1 /\ 1:r1=0)
```

Figure 4.10: Message Passing 1 and modified

```
*       Initial Conditions      *
{0:r2=x; 0:r4=y; 1:r2=x; 1:r4=y;}

         P0      |       P1
1:  li r1,1      |  lb  r3,0(r4)
2:  sb r1,0(r2)  |  sync
3:  sync         |  lb  r1,0(r2)
4:  sb r1,0(r4)  |

    Exists (1:r3=1 /\ 1:r1=0)
```

Figure 4.11: Message Passing 2

the cache line timer does not expire during the duration of the test. A cache line lifetime of 10,000 cache cycles is used to produce the results shown.

The limtus tests were also executed on the Directory version of BERI and the results are presented in the same table. Notably the Directory model demonstrates a stronger consistency model and the final outcome (r3=1, r1=0) is never observed. This demonstrates expected TSO behaviour.

A notable comparison between the Time-Based and Directory results are the relative observations. Majority of Time-Based coherence results show initial value loads, whereas the Directory demonstrates more fully updated values or transitional results.

```
1:  lld    $8, 0(%0)
2:  daddu  $8, $8, %1
3:  scd    $8, 0(%0)
4:  beqz   $8, 1b     {Branch to label 1}
5:  nop
6:  sync*
7:  ld     $8, 0(%0)
8:  bne    $8, %2, 6b {Branch to label 6}
9:  nop
10: sync
```

Figure 4.12: Barrier Implementation

The MP1-mod test with default barrier shows another interesting outcome for Time-Based coherence, no intermediate states are observable, only initial or updated values are seen. This is another indicator of the SYNC behaviour.

TODO: More analysis of table results.

	Modified Barrier	Coherence Model	Observed Outcomes			
			r3=1 r1=1	r3=0 r1=1	r3=0 r1=0	r3=1 r1=0
MP1 default		Time-Based	0	0	**1000**	0
	•	Time-Based	81	18	895	**6**
		Directory	517	107	376	0
	•	Directory	620	101	279	0
MP1 modified		Time-Based	271	0	729	0
	•	Time-Based	71	5	913	**11**
		Directory	543	72	385	0
	•	Directory	623	22	355	0
MP2 default		Time-Based	49	43	908	0
	•	Time-Based	95	43	862	0
		Directory	529	115	356	0
	•	Directory	627	124	249	0

Table 4.1: Litmus: Message Passing Observed Outcomes

4.2.8 AXE Trace Evaluation

So far we have discussed the behaviour of Time-Based coherence using different memory access patterns. A detailed evaluation of numerous litmus tests allows us to determine the memory consistency model of a given hardware memory architecture. AXE analysis has confirmed that out implementation of Time-Based coherence follows the RMO consistency model. A few examples of testing the Time-Based memory model shown why stronger consistency is not supported. The example traces follow the trace format shown in [Section 4.2.5].

Sequential Consistency (SC) Test The test sequence shown in [Table 4.2] is an extract of an AXE test failure running on a software simulation of Time-Based coherence.

SC requires the propagation of all loads and stores, every cycle. The test performs a store to (x) at time 0 and a store to (y) at time 1. At time 3, loads to both variables (x) and (y) produce initial values. This behaviour is not permitted by SC. A correct implementation would result in both (x) and (y) loading updated values.

Testing SC memory model	**Comments**
`Initial: x==0, y==0`	
`≫ 0: x == 0`	`core[0].op(LW,'h2)`
`≫ 1: x := 1`	`core[1].op(SW,'h2)`
`≫ time delay`	`other instructions`
`≫ 0: y := 1`	`core[0].op(SW,'h0)`
`≫ 1: y == 0`	`core[1].op(LW,'h0)`
`≫ time delay`	`other instructions`
`≫ 0: x == 0`	`core[0].op(LW,'h2)`
	`core[0].getResponse`
`≫ 1: y == 0`	`core[1].op(LW,'h0)`
`≫ Failed!`	

Table 4.2: AXE: SC Evaluation

Total Store Order (TSO) Consistency Test Similar to the SC evaluation, a TSO evaluation of Time-Based coherence is shown in [Table 4.3]. TSO enforces an ordering of stores to memory location, out of order loads are permitted. The test performs a store to (x) at time 0 and a store to (y) at time 1. At time 3, (x) loads the initial value and (y) loads the updated value. TSO does not permit this behaviour. A correct implementation would result in either: (x) and (y) both loading initial values, or (x) and (y) both loading updated values.

Testing TSO memory model	**Comments**
`Initial: x==0, y==0`	
≫ `0: y == 0`	`core[0].op(LW,'h2)`
≫ `1: x := 1`	`core[1].op(SW,'h0)`
≫ *`time delay`*	*`other instructions`*
≫ `0: y := 1`	`core[0].op(SW,'h2)`
≫ *`time delay`*	*`other instructions`*
≫ **`0: x == 0`**	`core[0].op(LW,'h0)`
	`core[0].getResponse`
≫ **`1: y == 1`**	`core[1].op(LW,'h2)`
	`core[1].getResponse`
≫ `Failed!`	

Table 4.3: AXE: TSO Evaluation

Partial Store Order (PSO) Consistency Test The PSO memory model allows caches to buffer writes. Batching minimises the number of write accesses to shared or lower levels of memory. Loading values from write buffers is also permitted. As a result of this design, store reordering is allowed by PSO consistency. In the example shown in Table 4.4 shows a PSO analysis on Time-Based coherence. The test sequence updates the value of (x) and thread 2 observes the initial update. Thread 1 updates the value of (x) again but thread 0 lodes the initial value of (x). Since thread 2 was able to observe an updated value, loading the initial value by thread 0 did not follow expected PSO behaviour.

Testing PSO memory model	**Comments**
Initial: x==0, y==0	
≫ 0: x == 0	core[0].op(LW,'h1)
≫ 1: x := 1	core[1].op(SW,'h1)
≫ 2: x == 1	core[2].op(LW,'h1)
≫ *time delay*	*other instructions*
≫ **1: x := 2**	core[1].op(SW,'h1)
	core[1].getResponse
≫ *time delay*	*other instructions*
≫ **0: x == 0**	core[0].op(LW,'h1)
≫ Failed!	

Table 4.4: AXE: PSO Evaluation

4.2.9 Performance Testing using CHERI Litmus

Litmus tests evaluate memory behaviour by executing instruction permutations. As a side effect, execution time varies depending on the memory model under test. Timing CHERI litmus tests on Time-Based and Directory models has highlighted some key performance differences. The Directory model exhibits strong memory consistency, run time is lower as barrier loops fewer cycles to complete. Observations of the Time-Based have shown that barriers implemented without synchronisation instructions work, however, loops spend a long time waiting for private caches to self-invalidate. As a result the execution time is very high. Appropriate SYNC instructions in the barrier greatly improve performance.

```
*   Initial Conditions   *
       P0    |      P1
1:  li r1,1  |  li  r3,1
2:  nop      |  nop
```

Figure 4.13: Litmus NOP Test

A special NOP instruction test was written to evaluate the performance (Figure 4.13). Litmus tests require some register evaluation, hence, two initialisation instructions were added. Note that none of these instructions interact with memory, so any slowdown will be due to the barrier implementation.

The execution time of a Time-Based model with a time-out value of 10,000 and no barrier SYNC shows a (>32×) slowdown as compared to a Directory model. Inserting a SYNC instruction (Figure 4.12) reduces the slowdown to slightly over (2×). The relative improvement in the execution time of Time-Based coherence is (>14×). At least 5 samples were taken for each test, there was no more than a 1% variation in the timing for all samples. The simulation produced identical results with each run, so the execution time variation can be attributed to OS behaviour. These results are shown in Table 4.5.

Comparing the default MP1 test run with and without the extra Barrier-SYNC, the SYNC instruction present in the branch delay slot is executed more than (>322×) as compared to the one without the barrier. This explains the huge performance gap observed when running the tests with and without additional SYNC's. The strength of the Time-Based mechanism lies in correctly crafted code. As we will see

in the Evaluation section, the coherence mechanism often performs equally or better than a directory when running FreeBSD (TODO: How FreeBSD helps with this?).

Additional barrier SYNC instructions have no significant effect on the Directory model due to an eager coherence behaviour. There is a small penalty for executing the SYNC instruction but it is negligible compared to other test overheads.

TODO: Describe system set up on Vaucher. The performance of the server is irrelevant as we are interested in a direct comparison of execution time rather than absolute values.

TODO: Test 1,000 & 100,000 time delays. The current results are based on 10,000 self-inv time-out.

Coherence & Barrier Type	Execution Time (sec)
Time-Based (No Barrier SYNC)	2575
Time-Based (With Barrier SYNC)	180
Directory (No Barrier SYNC)	80
Directory (With Barrier SYNC)	80

Table 4.5: Litmus NOP Test Performance

4.2.10 TODO: Beneficial BERI Memory Features

- Write-through L1 caches. BERI's write-back policy provides a guarantee that any data written to the L1 data cache will be immediately propagated to the L2 cache (LLC in case of BERI). Any modified data in the d-cache is also present in the L2. Write-through caches are generally undesirable as they impose high bandwidth requirements on the underlying memory sub system. However, the scheme reduces the complexity of the cache design. TODO: Elaborate further.

- LL/SC propagated to LLC. A correctly functioning LL/SC mechanism is critical for a relaxed memory architecture. Locks and other OS synchronisation primitives are based on LL/SC and an inconsistency will result in unexpected behaviour. TODO: More and better explanation.

Time-based coherence can be added to many existing coherence models in order to reduce spontaneous coherence communication. Examples of such designs have been demonstrated in [References].

SYNC: RMO schemes rely heavily on the appropriate use of SYNC instructions. TODO

4.2.11 TODO: Regression Testing

We have already discussed the AXE memory consistency evaluation tool. The tool has confirmed that BERI Time-Based coherence complies with the AXE RMO consistency model. The checker tool tested a total of 1,000,000 memory operation permutations on the coherence model in simulation. Table 4.6 shows memory executions with different parameters. The instruction depth indicates the number of instruction checked for correct consistency behaviour. Each instruction depth was tested 200 times.

Model Parameters	Instruction Depth	No. of Iterations	Outcome
RMO	5000	200	Pass
RMO +LLSC	5000	200	Pass
RMO +SYNC	5000	200	Pass
RMO +LLSC +SYNC	5000	200	Pass

Table 4.6: AXE Time-Based Consistency Results

Each model parameter specified in the table can affect the behaviour of the system. The presence of an LL/SC instruction could affect the time between other memory operations. If a SYNC check depends on this timing then an incorrect test pass can be detected. In order to avoid such a scenario, the time-based memory consistency scheme was tested with all combinations of test parameters.

TODO: Further Details

4.3 Comparison with Other RMO Systems

TODO

4.4 BERI Directory Coherence

I have tested several coherence mechanisms while developing BERI multi-core. The BERI Directory protocol is the result of a refined exploration of communication centric coherence protocols. One of the most basic coherence protocols for a shared memory system is: Invalidate on writes (IOW). Coherence is maintained by broadcasting invalidation messages when ever a store operation is performed in the shared cache. The protocol will operate correctly only if the data caches are write-through. A write-back cache will hold a dirty line until it is evicted. Without explicit barriers, coherence will fail (TODO: Confirm). The IOW coherence mechanism leads to large coherence communication overheads since, every store operation generates a broadcast message. Costs further increase if the data caches invalidate lines without Tag lookups. Even if the data caches check Tag's and only invalidate on Hit's (significantly reducing miss rates), the data cache will be blocked during invalidation. The combination of above factors necessitates cache inclusion (TODO: check property).

Constricting the properties of IOW and tracking sharers resulted in the first iteration of the directory protocol. The last level cache holds a single bit per core per cache line. A combination of all shared bits for a given LLC line is a sharers list. The list indicates whether that line is also present in one of the private caches. When the shared line receives a store operation, the sharers list identifies caches holding stale data copies. The LLC sends an invalidate message to a coherence controller. This device simply reads the sharers list in the message and then distributes the invalidate to all relevant private caches. The coherence network used by the LLC and the coherence controller is isolated from other memory communication. This allows low latency invalidate distribution. Invalidate messages take priority in private caches in order to enforce a TSO memory consistency model. We choose not to enforce coherence in the instruction caches since, self-modifying code is not used.

The directory is fully contained in the LLC, thus, data caches incur no coherence storage overheads. Memory overheads are dramatically different in the Time-Based design and the Directory design. The former incurs data cache storage overheads, where as the latter incurs LLC storage overheads.

Data Cache Short-Tag's (TODO: Restructure section) (TODO: Quick baremetal test to highlight improvements) A performance optimisation designed to reduce congestion due to invalidates in the data caches has resulted in some storage overheads, however, the directory scheme can be implemented without this feature. Figure 4.14 shows the Tag structure for a standard data cache with directory coherence. Figure

4.15 shows the Tag structure for an optimised directory coherence data cache. The Short-Tag optimisation allows parallel memory access Tag lookups and invalidate Tag lookups. The Short-Tags are a subset of the complete physical address Tag. Block RAM's require 2 cycles to respond, one cycle to submit a BRAM request and one cycle to respond. During this operation all I/O ports of the cache are blocked. A subset of the Tags is sufficient for maintaining a low invalidate false miss rate, but the scheme allows Short-Tag lookups while the regular Tags are also accessible. As a result, an invalidate operation only blocks the cache for 1 cycle instead of 2.

Figure 4.14: D-Cache Tag's, Dual-Core [Directory Coherence Default] (16KB Size)

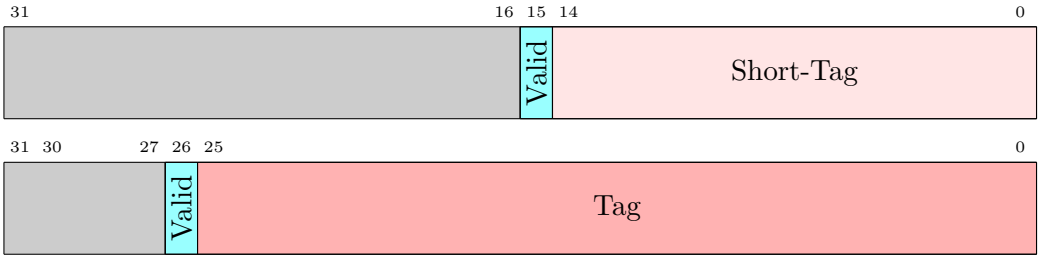

Figure 4.15: D-Cache Tag's, Dual-Core [Directory Coherence using Short-Tag optimization] (16KB Size)

An alternative to using Short-Tags would be either completing a full Tag lookup or blindly invalidating cache lines. The former adds a cycle of latency where as the latter will increase cache miss rates.

As mentioned earlier, the Short-Tag's are accessible independently of the cache line Tag's as they have been implemented in a separate BRAM. The size of the Short-Tag's is such that for a given data cache capacity, all bits fit into a single BRAM. For this reason a total of 16 bits are used in a 16KB data cache. One valid bit an 15 bottom bits of the physical address.

The Short-Tag's provide a sufficient number of bits to prevent most cache misses and save 1 cycle. If the Short-Tag's are valid and the select bits of the physical address match, then the line will be invalidated, otherwise no action is taken. This mechanism is proven correct through memory consistency verification, baremetal tests and correct FreeBSD operation.

The LLC must maintain a sharers list as specified above. We use a shared L2 cache and the overheads for the directory scheme are shown in Figures 4.16 and

4.18. For comparison, the L2 Tags for the Time-Based coherence model are shown in Figure 4.17. Since one bit per core is required to maintain directory coherence, the overhead is clear in Figure 4.18, where the overheads double for a quad-core BERI.

Figure 4.16: L2-Cache Tag's, Dual-Core [Directory Coherence for D-Caches] (64KB Size)

Figure 4.17: L2-Cache Tag's, Dual-Core [Time-Based Coherence] (64KB Size)

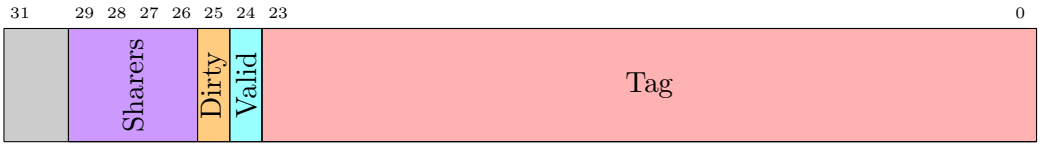

Figure 4.18: L2-Cache Tag's, Quad-Core [Directory Coherence for D-Caches] (64KB Size)

The shared L2 cache is write-back, one coherence property that emerged was the need for maintaining cache inclusion. The issue became apparent when I was attempting to run FreeBSD on dual-core BERI. If a line is evicted from the shared cache, an invalidate must be broadcast to all sharers caches. Otherwise stale data will remain in those caches unless the select line is replaced or re-fetched. The mechanism used in FreeBSD signalled core 1 through shared memory and then core 0 proceeded with the boot. Many cycles later core 0 would signal core 1 again to wake it up. Since many cycles had passed and only regular load/store operations were used, the line was likely evicted from the L2 and core 1 would never see the updated data value. This would lead to an unresolvable livelock. Due to this reason the coherence mechanism was improved to invalidate sharers on line eviction.

TODO: place appropriately - The coherence mechanism only deals with Data-Caches, we do not run JIT compiled code and Instruction-Cache coherence is not necessary.

4.4.1 Example Trace

The directory coherence model memory consistency was verified using the same evaluation techniques as the ones used for the Time-Based model. The BERI directory enforces TSO consistency. This memory model was selected partly due to relevant research in the field [TODO; References] and partly due to the inherent properties of coherence messaging. The coherence network allows fast and efficient distribution of messages, it seems unnecessary to constrict this behaviour in order to comply with PSO, RMO, or other weaker consistency models. Time-Based coherence research often focuses on modifying existing designs based on TSO, primarily X86 ISA variants. The BERI directory passes all but the SC consistency check in our framework (AXE).

The example shown in Table 4.7 demonstrates no SC behaviour. The two load operations of (x) by cores 1 and 2 would produce the most update value of the variable, had the model obeyed SC. However, one of the two cores loads a stale value. The stale value had been observed in the local cache of core 2 in a prior load operation, and this behaviour is acceptable in TSO.

Testing SC memory model	**Comments**
`Initial: x==0, y==0`	
`≫ 0: x := 1`	`core[0].op(SW,'h3)`
`≫ time delay`	
`≫ 1: x == 1`	`core[1].op(LW,'h3)`
`≫ 2: x == 1`	`core[2].op(LW,'h3)`
`≫ time delay`	
`≫ 0: x := 2`	`core[0].op(SW,'h3)`
`≫ time delay`	
`≫ 1: x == 2`	`core[1].op(LW,'h3)`
`≫ 2: x == 1`	`core[2].op(LW,'h3)`
	`core[2].getResponse`
`≫ Failed!`	

Table 4.7: TestMem: SC +no_conditions

This behaviour simply implies that core 2 has not yet received an invalidate message for the given memory location. All invalidates are delivered to all cores simultaneously, however, a data cache might be in the process of fetching a memory

line (possibly (x) in this scenario) or performing another blocking operation, this would result in an invalidation delay. Our caches require that all responses must be consumed, so cancelling a fetch midway is not possible. There has been some recent research on a blend of Time-Based and directory coherence following SC consistency [TODO: Reference], however, it stands alone in the field.

4.4.2 BERI Directory Coherence Tricks (TODO: Is this required?)

L2 Invalidate instruction If the OS or other software issues a specific L2 cache invalidate instruction, the desired line could be shared. If the line is shared then all the sharers must be invalidated to preserve inclusion. This may not be the case for a single core system. BERI singlecore does not invalidate the D-Cache, and there is no mechanism to do so.

Preserving Sharers List The L2 cache stores Tags on every memory operation. This is done to ensure that all cores accessing shared memory are included in the sharers list. Load's typically do not need to affect the Tag's, however if more than one core is loading a cache line then each core must be added to the sharers list. Failing to do so would result in inconsistencies and stale data residing in the D-Caches.

Load/Store Miss If a line needs to be fetched from main memory, the requesting core has exclusive access, and hence is the only one added to the list. The D-Caches follow the non-write allocate policy, thus an access to the L2 and/or an L2 store miss guarantees that the line is not present in the D-Cache. The sharers list is null on a store miss.

TODO: Update Directory Protocol * An update based version of the Directory protocol has also been tested. I ran some select Splash-2 benchmarks and determined that the protocol was less efficient and performed weaker than the equivalent invalidate protocol. This has also been identified by a number of papers [TODO: reference papers for this statement]. The graph shown [TODO] shows a performance comparison between the two versions. Much like the invalidate protocol, the update protocol sent copies of the new data to all sharer cores along with some meta-data (Tag, Byte-enables). The d-Cache then compares the Tag of the update with its local copy, the line is updated with correct byte-enables on Hit. One disadvantage

of this scheme is 1 cycle lost per Tag lookup for update comparison. Regardless of the Hit/Miss, 1 cycle is lost for the actual Tag comparison, data can we written or not in parallel so there is no additional penalty. This mechanism does reduce subsequent data miss rates, but at the cost of more coherence communication, d-Cache blocking, and data wires. As we will see in the Evaluation Chapter, the benchmarks share data but much of it is accessed only once, hence, the update protocol generally performs worse (TODO: check statement).

4.4.3 Regression Testing

The BERI Directory coherence model was tested using Bluecheck simulator. The final revision of the coherence scheme passed all the tests shown in Table 4.8. Each test simulation ran a 1,000,000 tests before declaring a pass.

Model Parameters	Instruction Depth	No. of Iterations	Outcome
TSO	5000	200	Pass
TSO +LLSC	5000	200	Pass
TSO +SYNC	5000	200	Pass
TSO +LLSC +SYNC	5000	200	Pass

Table 4.8: Bluecheck Directory Coherence Results

TODO: Details

4.5 Overheads Comparison

Coherence mechanisms typically add logic and memory overheads, the overheads for our Directory and Time-Based coherence models are highlighted in this section. The absence of a coherence network limits the overheads of the Time-Based model.

Time-Based Coherence

- Extra control logic in the D-Cache: Tag-time-counter and time-counter comparators.
- Time-Counter's. Size of the counter depends on the implementation but the likey range is between 16-64 bits.
- Tag-Time-Counter. Every D-Cache line requires this counter. The size of this counter will depend on the selection of the Cache Time Counter. Expected to be 4-20 bits. 4 bits per line for the optimised counter (4×512 bits for 16KB direct-mapped D-Cache, 64 bytes per line), 20 bits for non-optimised counter (20×512 bits for 16KB direct-mapped D-Cache, 64 bytes per line).

Directory Coherence

- Extra control logic in the D-Cache: Invalidation and tag lookup logic.
- Extra control logic in the L2 cache: Invalidation and sharer list evaluation logic.
- If Short-Tags are used then: 16×512 bits for 16KB direct-mapped D-Cache.
- Coherence network. Width: 48 bit physical address + sharer bits (core dependant) (TODO: the whole phy-address is not required, it can be trimmed). The coherence message is broadcast by the L2 cache and then distributed only to the sharers by the Multicore module (glue module between the L2 and other parts of the system).
- L2 cache sharers list required for each memory line. For a dual-core: 2×2048 bits, 64KB cache, 64 bytes per line.

FPGA Area Overheads (TODO: Refine section) BERI multi-core has been generated and tested on FPGA. The Altera Quartus tool is used in the synthesis process. Some of the key FPGA resource overheads are highlighted in Table 4.9 and Figure 4.19. FPGA register statistics for a dual-core build show that the Time-Based design requires ∼1% fewer registers than the Directory design.

The FPGA compiler outcome depends on a number of factors and each build will be different, however the resource usage between the two models is sufficiently large to make the overhead observations.

The data collected from the Quartus build is based on a Directory coherence dual-core BERI with the short-tags optimisation and a Time-Based coherence dual-core BERI using the short-tag-time-counter. The short-tags optimisation in the Directory version is a performance optimisation and may not be necessary in some systems. Some of the total Directory overheads will be due to this optimisation but a significant portion will be due to additional logic and wiring required for the coherence network.

Quartus II 64-Bit – Version 13.1.0 Family Stratix IV Device – EP4SGX230KF40C2			
Statistic	Directory	Time-Based	Capacity
Total Logic Utilization	106,457 (58.4%)	105,453 (57.8%)	182,400
Combinational ALUTs	72,262 (39.6%)	70,942 (38.9%)	182,400
Total Registers	65,902	65,188	14,625,792
Dedicated Registers	65,504	64,790	14,625,792
Total BRAM Bits	3,833,174	3,796,310	14,625,792
ALUT/Register Pairs	101,134	99,962	NA
Clustering Difficulty	Low	Low	NA

Table 4.9: Dual-core BERI FPGA Resource Overhead Comparison

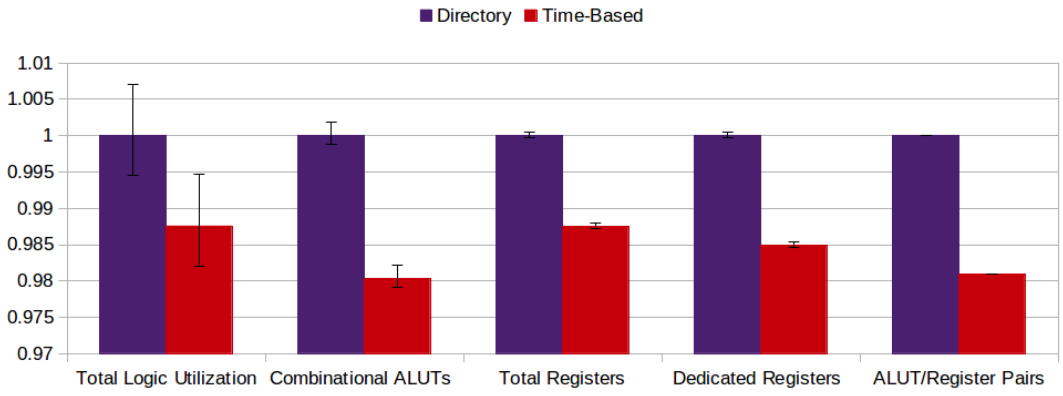

Figure 4.19: Quartus Overheads

4.6 Application of Time-Based Coherence

TODO