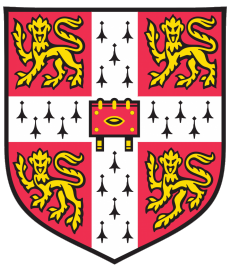
Time-Based Memory Coherence



Alan Mujumdar

Computer Laboratory

University of Cambridge
Christs College

A dissertation submitted for the degree of

Doctor of Philosophy

4 January 2016

Contents

List of Figures

List of Tables

Abstract

Time-Based Memory Coherence
Alan Mujumdar

Cache coherency is the dominant mechanism for data sharing in commercial multiprocessor systems. Such mechanisms are complex to implement and can become costly (both power and performance) for larger systems. Many cache coherency mechanisms, like directory-based approaches, aim to carefully coordinate data sharing, a distributed problem demanding a high volume of coherency messages to maintain order. This thesis explores an alternative approach focused on the lifespan of data in caches, which can be monitored locally. We demonstrate that this time-based coherency approach can be: simpler to implement, requires no coherency messages, performs surprisingly well, and can more efficient under appropriate circumstances.

The proposed time-based coherency approach takes inspiration from software oriented time-based coherency mechanisms and self-invalidating techniques used in some GPU caches. To thoroughly evaluate the approach I have designed a multi-core version of the BERI processor, implemented on FPGA and supporting the FreeBSD operating system. Thus, a full system evaluation was made possible. A directory-based coherency scheme was also implemented to provide a base-line comparable with commercial approaches.

Cache coherency mechanisms have also been exploited to break security, since a malicious thread can interfere with the temporal properties of a program under attack. We demonstrate that time-based coherence can be tuned to make side channel observations more challenging, which we believe can be used, together with other techniques, to mitigate side channel attacks.

Chapter 1

Introduction

1.1 Strict or Relaxed Consistency?

TODO

1.2 Avoiding Coherence Messaging

TODO

1.3 Reducing Side-Channel Leakage

TODO

1.4 Research directions

TODO

1.5 Contributions

TODO

1.6 Dissertation Overview

TODO

Chapter 2

Background

Protocols & Attacks

2.1 Memory Consistency

- (1988) THE DESIGN OF A LOCKUP-FREE CACHE FOR HIGH-PERFORMANCE MULTIPROCESSORS http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=44672 TODO
- (1992) An Effective Write Policy for Software Coherence Schemes http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=236636 TODO: Good argument to use when describing a write-back self-inv policy.
- (2006)TESTING MEMORY CONSISTENCY OF SHARED-MEMORY MULTIPROCESSORS http://xenon.stanford.edu/~hangal/manovit_thesis.pdf TODO

2.2 Cache Coherence

TODO

2.3 Time-based Coherence Protocols

Shared memory multi-core processors typically implement hardware cache coherence based on variants of MESI. Commercially available X86 based processors by Intel and ARM use extensions of MESI such as MOESI and MESIF. ARM designs are also largely based around MESI. The processors referenced above are not Chip Multiprocessors (CMP), they use a fixed memory hierarchy and share a common cache. The last level cache is typically shared, typically level 2 or 3.

MESI is a robust protocol that defines four states, each cache line can exist in one of these states. Coherence communication for this protocol is typically invalidate or update based. One of the major drawbacks of coherence communication is the added hardware complexity, power dissipation, latency, and other factors. As a result we observe slow access to global memory and the lack or absence of instantaneous broadcast mechanisms. It has been shown that broadcasts are infrequently used by typical user programs [REF] , there might be higher occurrences in HPC applications (SPLASH-2 OCEAN???). In an attempt to avoid or reduce coherence communication, a number of time based schemes have been proposed. In a time based scheme, each private cache has the capacity to self-invalidate a memory word with varying granularity. The notion of time can provide precise memory consistency guarantees even in the absence of explicit coherence messages.

The memory consistency model of cache coherence protocols rely heavily on barriers, fences, and synchronisation instructions. These instructions are actively used by the operating system and compilers. It is also the case for time based schemes. Time based coherence research has predominantly focused on enhancing the performance of MESI based protocols using timestamps. Compiler inserted invalidate instructions are also a popular choice in time based protocols. We observe the difference between the two types of time based coherence implementations below.

As applications continue to grow in parallelism many modern software languages adapt very strong techniques for locking and synchronisation schemes.

This is due to the fact that processor architectures are introducing more levels of parallelism as well. The boundaries between traditional processing elements (CPUs, GPUs, etc) is being erased as software is designed to exploit maximum parallelism. Common hardware consistency/coherence mechanisms have overheads, some more than others. The X86 model behaviour is very different from that of ARM or PowerPC. X86 enforces a very strong memory consistency, this in turn allows a much more relaxed software design as the hardware will take care of it. ARM or PowerPC use much more relaxed hardware schemes and expect the software to correctly handle parallel workloads. So, stronger software implementations are required, however, the model still provides programmer assurances, albeit more care needs to be taken.

Interesting things happen when a piece of software is designed to be cross platform. Often the compiler will insert architecture appropriate primitives. But it is also beneficial for the software developer to design the software accordingly. If the software is designed for a system with a weak memory model, it will likely work just fine on a stronger model.

A lot of software is already compatible with weaker consistency schemes. Hence, we ask the question What is the weakest memory consistency scheme that can still support commercial software without any modification of said software? This question has lead us to develop the mRMO consistency scheme. One of the versions of the BERI multicore has been built with the mRMO memory mode. This processor can run FreeBSD without any modifications to the OS. The consistency scheme is more relaxed that the PowerPC model, however it is still usable and provides a number of programmer guarantees.

The memory model relies on hardware time-stamps. Private caches keep lines with valid time stamps and self-invalidate otherwise. mRMO does not use any explicit inter-core communication messages. The scheme solely relies on private caches; self-invalidating, and correct SYNC and LL/SC mechanisms.

Time-stamp based coherence is not a new concept in itself, many have described this mechanism in conjunction with other memory models (quite

often based on X86). However, there is little evidence that anyone has attempted to use this as a standalone memory consistency scheme. We suspect that this is due to fact that most implementations attempt to maintain X86 compatibility. There has been related research where time-stamps are used for GPU coherence, but explicit coherence messages are still required.

We have compared a hardware implementation of mRMO with a Directory scheme on the same processor, mRMO achieves equivalent or better performance for many benchmarks/workloads. We are not enforcing the idea of using mRMO as a universal system, but there is lots of scope to use this scheme with unpredictably behaving software such as HPC applications. Given that architectures with weaker memory models are being more and more widely used, we believe that there is scope for simplifying the hardware design, at the cost of software of course and potentially some performance but a simpler chip can yield some benefits.

2.3.1 Time-based Coherence Related Research

Memory consistency and coherence are closely tied. Consistency establishes rules for coherence. Thus, coherence is largely a software-hardware implementation of consistency.

Multiprocessor systems benefit from coherent shared memory, data sharing between processing element (PE's) is simpler. Conceptually, one of the simplest possible coherent system would be one where all PE's share one common memory. While perfectly plausible, this system would be highly inefficient. In general PEs operate several magnitudes faster than memory. Significant improvements have been made in memory technologies, but the performance gap is still quite large. Hierarchical memory arrangement bridges the performance gap by masking memory latency.

Caches are typically situated on the same substrate as the PEs. Faster load/store rates are achieved through physical proximity and the construction of memory. Data spatial and temporal locality is another critical factor for exploiting memory performance. The number of memory layers tends to increase with an increase in processing elements. A multiprocessor system

will typically have private caches for each PE, and a larger shared cache. Current architectures continue relying on this model, however, a number of other more scalable architectures have been proposed.

Software parallelism is one way of improving overall performance, it has been the tendency over the past decade. Parallel software usually requires some communication between the distributed data. Memory consistency dictates this communication behaviour.

Typical coherence algorithms require explicit messaging between PEs. If PE 0 updates a memory location and PE 1 requires the same data, coherence hardware could propagate the updated value to PE 1. In practice things are not quite as simple. Most coherence schemes are highly sophisticated and require many resources.

TODO: Describe Directory, MESI, other coherence.

The idea of using timestamps for coherence purposes has been around since the early 90s. The main motivation for using timestamps is removing stale data in the cache and assisting an existing coherence mechanism. (NOTE: I have found no evidence of a standalone timestamp based coherence mechanism.)

Early work on time-based coherence was largely hindered by limited software support for relaxed memory. Stale cached data was a real problem for program correctness [x1988]. Explicit invalidate instructions inserted by the compiler were used to ensure appropriate eviction of stale data. These branched into two major categories: TLB-based invalidates and compiler inserted. The TLB approach was potentially wasteful as entire pages were deemed invalid. The compiler approach was finer grained but with overheads. A TLB self-invalidating scheme has been shown in [x2012b], aimed at JIT compiled self-modifying code.

Cache invalidation came in two flavours: indiscriminate and selective. The entire cache could be invalidated in as a single cycle operation, at the cost of higher miss-rates for still useful data [x1993][x1997]. Selective invalidation produces much lower miss-rates but requires sequential invalidation of cache lines which is expensive.

Timestamp-based coherence (TS) has been proposed by [x1992a] and improved in [x1996b]. This approach depends on compile-time analysis and additional hardware support. Caches have extra tag bits and counters that are used for coherence. Each shareable data structure is associated with a clock. This clock is incremented at the end of each epoch. A timestamp is associated with each memory word and compared to the clock, data is valid of the timestamp is greater or equal to the clock. There is a tradeoff between the number of bits used per cache word and the penalty of overflows. Write operations are analysed by the compiler and if they are shared then these will be marked. The mark allows the hardware to decide the clock value to assign. Tag overheads are reduced to 1 bit per line in [x1993][x1997] but clock overflow remains a constraint. Authors do not mention the consistency policy of TS or its variants but judging by their fear of stale data values, it is a strong model.

Improved compiler support and better exploitation of spacial and temporal locality can further simplify the hardware [x1994a][x1999a][x1999b]. The compiler directed scheme shown in [x1996a] has been implemented and tested on a Cray T3D processor. Compiler supported coherence form CMP's has been described in [x2008], using synchronisation points to maintain coherence is highly beneficial in large distributed systems.

In [x1994b] authors highlight major drawbacks of directory protocols: tracking ownership of every block, explicit individual block invalidation requests, blocking on release operations, requesting blocks requires coherence actions, and multi-level cache inclusion policy.

There have been a number of proposals to combine directory protocols with some form of time-based coherence [x1995], improved in [x2000] using a self-invalidation predictor table. Most of these observe a performance improvement. However, the protocols are still limited to a strong consistency model, in many cases sequential consistency [x2015b][x2015c][x2015d]. Further modification of directories using timestamps is shown in [x2011a][x2011b][x2014b][x2015a]. Most of this work is based on MESI style directories for X86, and strong consistency models. Overheads of directories such as sharers lists or write invalidation.

Another potential use for time-based coherence has been shown in [x2014a], aimed at GPU's. Traditionally GPU's support little or no coherence. The authors have shown that self-invalidating GPU caches can improve overall performance.

Work shown in [x2012a] demonstrates the need for coherence traffic reduction in large distributed systems.

Writeback policies are another critical factor when operating a time-based coherence mechanism [x1992b]. Writethrough caches are preferable [x2011b][x2014a][x2015a].

2.4 Cache Side-Channel Attacks

Chapter 3

BERI Multiprocessor Architecture

3.1 BERI Architecture

The Bluespec Extensible RISC Implementation (BERI) processor is based on a 64-bit MIPS instruction set architecture (ISA). The processor is implemented in Bluespec SystemVerilog. The initial implementation was done by Gregory Chadwick [TODO: Reference], the processor was extended to support MIPS R4000 [TODO: Reference] by Jonathan Woodruff [TODO: Reference]. BERI is a single issue, in-order processor with 6 pipeline stages. BERI has a branch predictor and register renaming. The memory structure consists of: instruction cache, data cache, and a shared level 2 cache. The processor can be synthesised to run on Field Programmable Gate Array (FPGA). Figure 3.1 shows a logical layout of the processor pipeline, control logic, and memory sub-system.

The BERI processor is further enhanced by adding capability support. This version of the processor is known as Capability Hardware Enhanced RISC Instructions (CHERI).

TODO: Further BERI details.

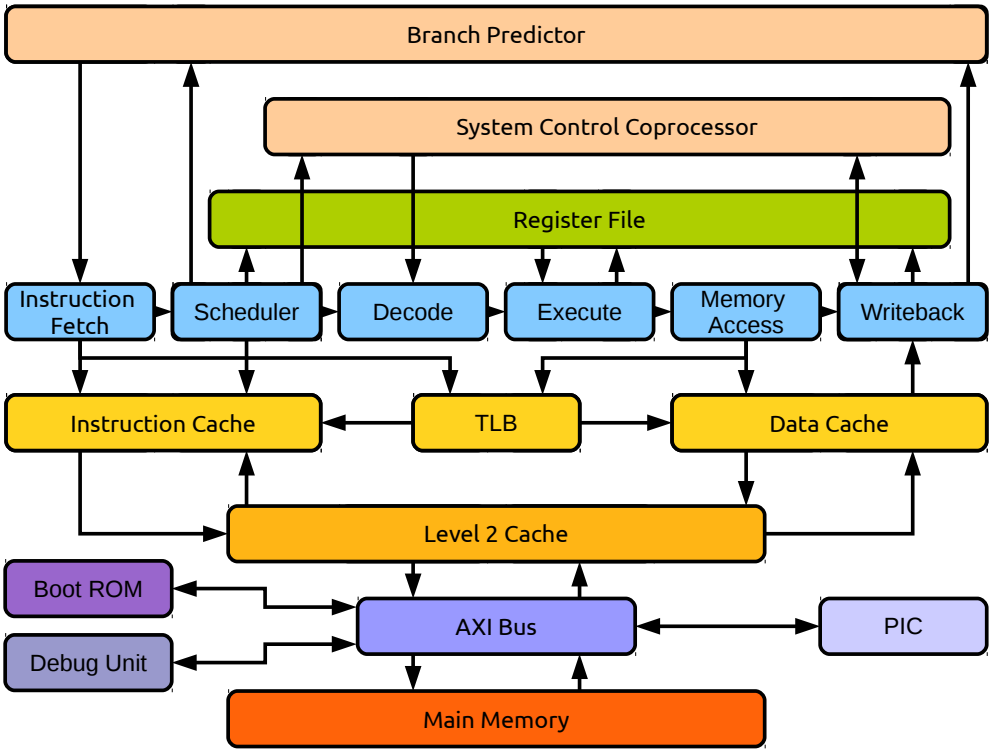


Figure 3.1: BERI Pipeline

3.2 Bluespec Language

TODO

3.3 Multi-core BERI Design

I have used the BERI processor core to produce a multi-core BERI design. Multi-core BERI supports a large number of cores, the design complexity and size is only limited by the Bluespec compiler. Most of the work described in this thesis is mostly based on a dual-core BERI, however multi-core version with up to 12 cores have been tested in hardware or simulation. Due to FPGA size limitations, the hardware tests have been limited to 1-4 cores on Altera DE-4 Stratix-4, larger FPGA chips should be able to support more cores.

BERI multi-core design is based on a shared memory multiprocessor. The private caches of each BERI core (I-Cache and D-Cache) communicate with a singe shared Level 2 cache. The L2 cache communicates with peripherals and main memory through the AXI bus [TODO: Reference AXI]. A number of pipeline modifications were made in order to support multiple cores. Multiple memory coherence models were tested as a part of this research and I have selected two to discuss in this thesis.

Memory Interface Modification

1. Directory Coherence: The directory is contained within the L2 cache (Figure 3.2). If the directory chooses to invalidate a shared memory line, it send an invalidate message through a separate invalidation interface. The message is processed by a coherence module, recipients of this message are selected based on the sharers list. This ensures no coherence congestion on the shared bus and fast invalidate delivery.

2. Time-Based Coherence: This coherence scheme operates through private cache self-invalidation (Figure 3.3). As a result, the L2 cache does not require any modification and the coherence module used in directory coherence is not required. There are no coherence messages and a coherence network is not required.

Core Identification Software often requires a method to identify individual cores/threads, the operating system learns the hardware set up and schedules software threads to run accordingly. Core identification is accomplished though a special coprocessor 0 (CP0) instruction. The mechanism used in BERI is similar to MIPS processors. The special instruction, accessible in kernel-space, returns a 32 bit value.

The bottom 16 bits hold the core id and the top 16 bits contain the total core count $\{0 = \text{One Core}\}$ (Figure 3.4). The distribution of core identifiers occurs at Bluespec compile time. Once built, the identifiers are fixed and can not be changed or overwritten.

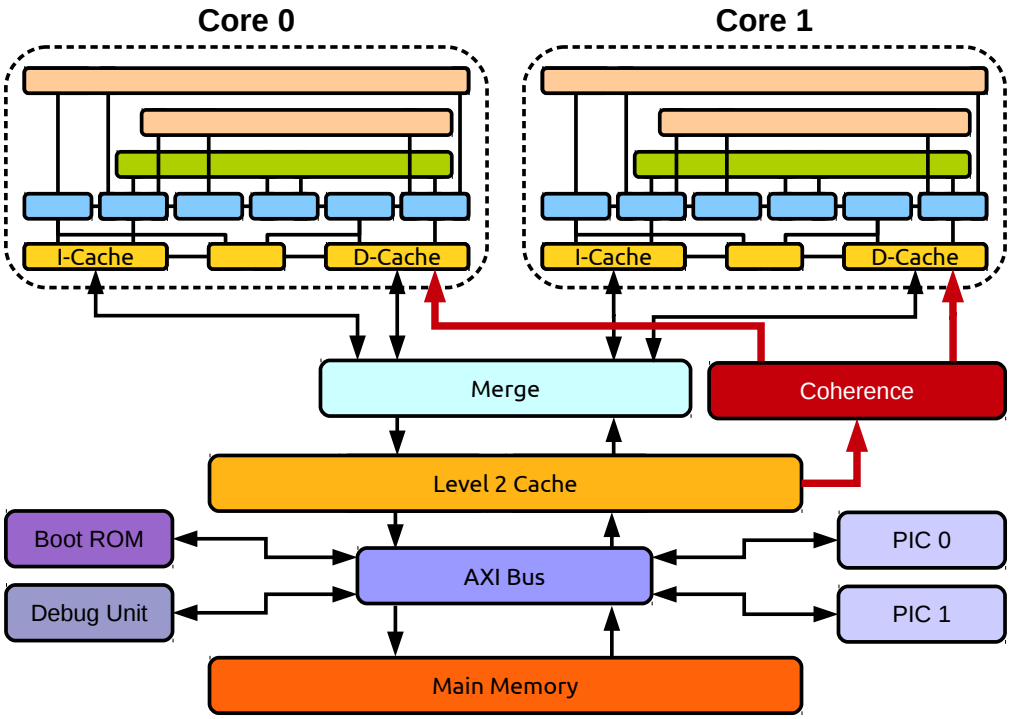


Figure 3.2: BERI Dual-Core Directory Coherence Processor

Interrupt Delivery In commercial multi-core processors, interrupt are often serviced through a hierarchy of programmable interrupt controllers (PIC's). The master PIC distributes interrupts to PIC's further down the hierarchy [TODO: Reference and credit Robert N.]. BERI multi-core requires one PIC per core. The PIC hierarchy is flat and interrupts are delivered to all PIC's simultaneously. Individual PIC's decide whether an interrupt needs to be serviced by the given core. Our FreeBSD OS accesses PIC 0 by default, avoiding any PIC management issues. We have encountered a number of software faults where the OS would forget to re-enable a PIC after the interrupt service routine.

Load Linked - Store Conditional (TODO: Refine section) Single core systems support LL/SC, however simple pipelines with no reordering or multi-threading require only basic support for this scheme.

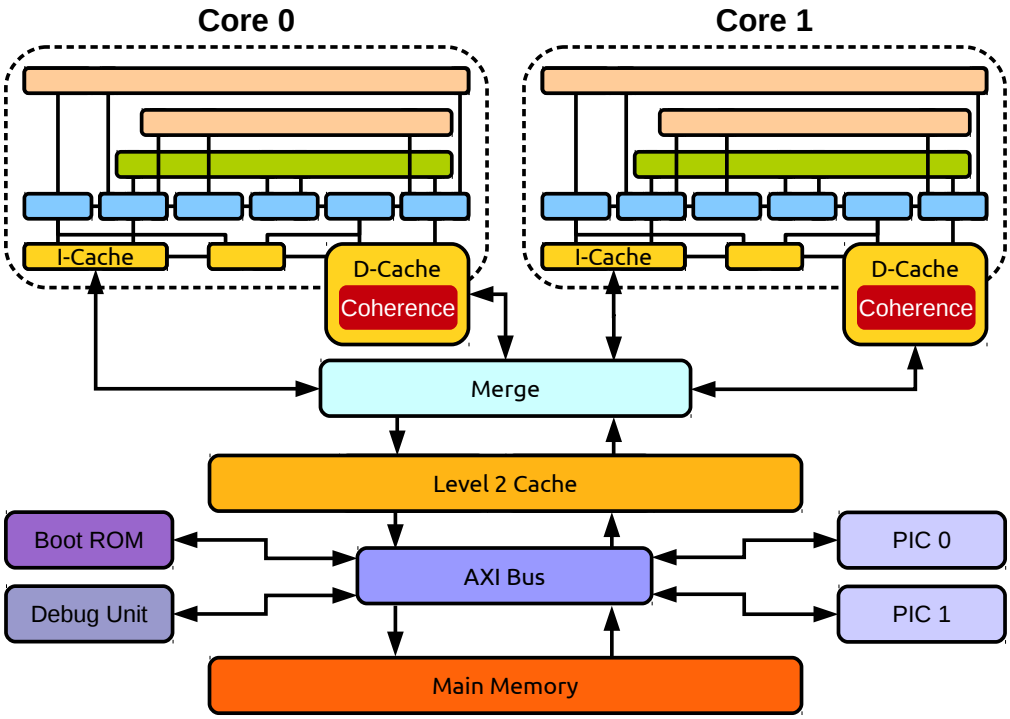


Figure 3.3: BERI Dual-Core Time-Based Coherence Processor

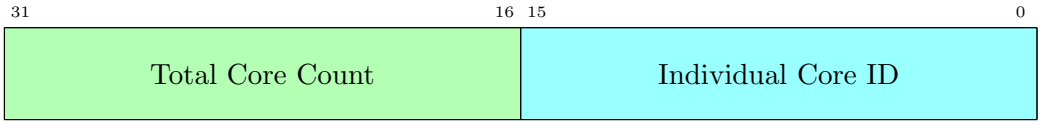


Figure 3.4: BERI Core Identification

TODO: description of LL/SC in context of MIPS, what does the software expect and guarantee.

Both LL and SC a special memory operations and they must be identified as such in the Execute stage. Additionally, SC is expected to return a success/fail message to the pipeline, thus producing a control hazard (TODO: Check). Due to the side-effect of SC the Memory-Access stage performs a special write operation that signals the Writeback stage to expect a response. The Writeback stage feeds the SC result back into the Execute stage.

- Scheduler marks an SC or SCD as a pending write.

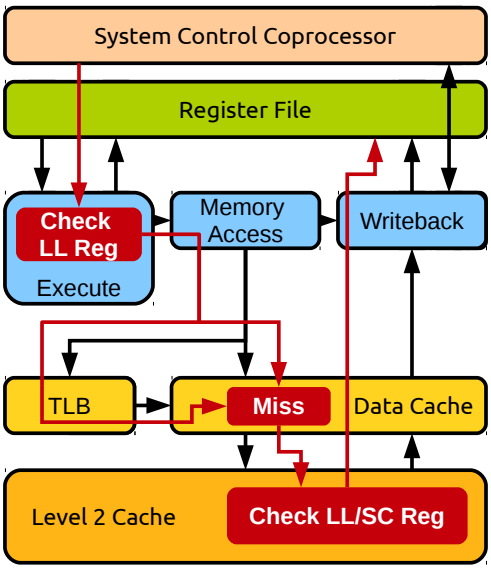


Figure 3.5: BERI LL/SC Mechanism

- Execute checks with the CP0 LL/SC register, if success then proceed. It is still a pending write so the result will be sent back to Execute from Writeback.
- MemAccess, write tagged as SC.
- Memory expects a response for SC.
- D-Cache: LL are treated as uncached, SC are special writes, uncached and response expected.
- L2: One LL/SC register per core (Same structure as shown in Figure 4.14). Each register holds a Maybe# Tag, the Register is written when the cache is accessed with an LL. The cache Tag's also hold a "Linked" field (Figure 4.16), this field is set on an LL and cleared on any store operation to the same line (Note: MIPS LL/SC specifies that even a regular load instruction may not be allowed to an LL address). The per core LL/SC register is preserved on a load-miss, once the line is fetched the "Linked" flag is set in the Tag's. An SC operation triggers a check of the cache line Tag's and the LL/SC register, if both are confirmed

to be valid and the address fields match and the access is not uncached and the access is from the same core, only then is an SC successful and the cache line is updated and a success is returned to the pipeline. In all other cases the cache line is not updated, the LL/SC register is cleared and a fail is returned.

- Writeback registers are allowed to be modified if SC is successful.

3.4 FPGA Implementation

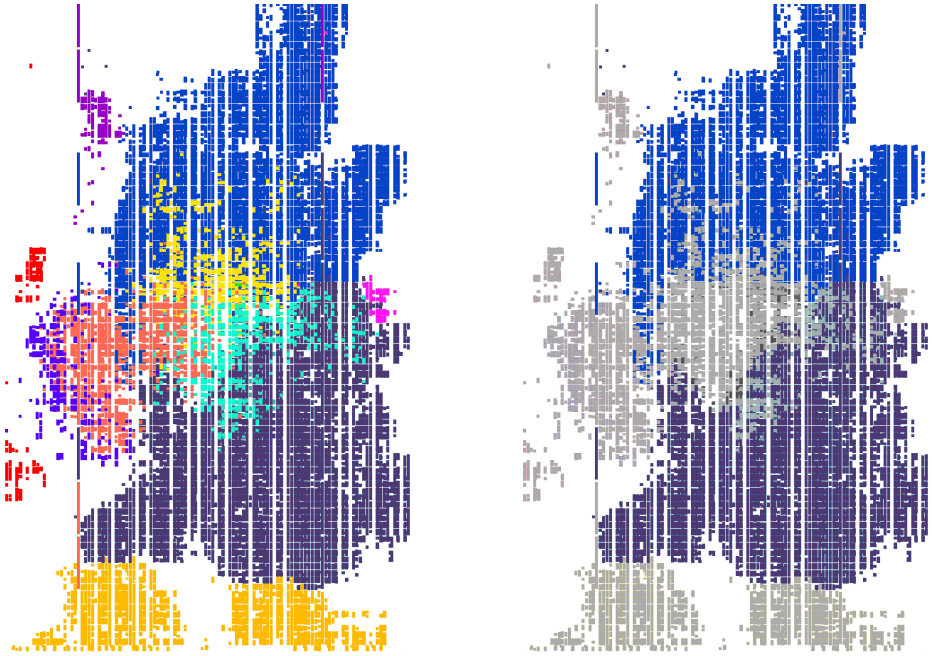
The BERI processor and its variants can be synthesised for an FPGA. We use the Terasic DE4 board equipped with a Stratix-4 FPGA. The board includes dual DDR2 sockets, USB, PCI-E, Gigabit Ethernet, SD-Card support, SATA, and other interfaces. Most of the debugging and testing of BERI is done via the above mentioned interfaces.

The Bluespec compiler can produce verilog which can be synthesised through the Altera Quartus tools into a complete design. Figure 3.6 shows the Quartus layout of the BERI dual-core processor. Due to the visualisation features of Quartus, some of the architecture components are overlaid and are not to scale. The set of figures show: two BERI pipelines, two private data caches, shared level 2 cache, main memory, boot memory, AXI bus interface, two uart's, and the debug unit. It is not possible to show the PIC's as they mostly consist of wiring and minimal logic. The data caches appear large in the figures as they communicate with the TLB, Memory Access and Writeback pipeline stages, and other processor components, as a result the cache logic is spread over a large portion of the FPGA.

TODO: Clock rates and other Quartus settings.

3.5 Hardware and Software Tracing

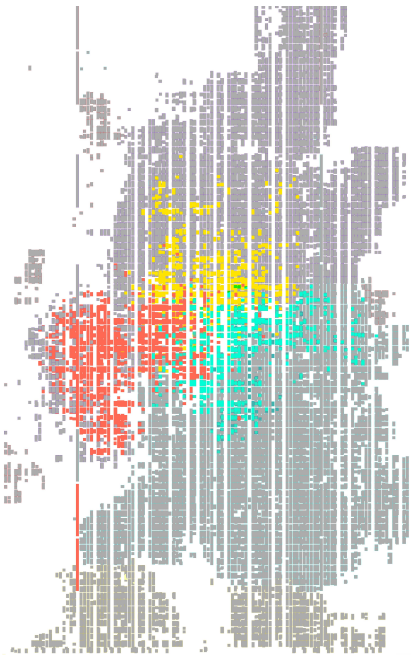
TODO



(a) Complete Design

(b) Core Pipelines

(c) Memory Hierarchy



(d) Cache Hierarchy

Figure 3.6: Quartus Dual-Core FPGA Layout (TODO: Legend)

3.6 Cheritest

TODO

3.7 SW Tests

TODO

3.8 A Checker for SPARC Memory Consistency

Memory consistency model behaviour defines the relative ordering of memory operations. The model can provide a fixed guarantee for the behaviour of a specific program. Consistency tends to be a greater concern when executing parallel code, as the code relies on a certain memory behaviour. In cache coherent processors, the memory consistency model can be influenced by the selected coherence mechanism. There is a wide variety of consistency models implemented in different processor designs.

3.8.1 Bluecheck Memory Models

The checker tool supports four consistency models described below.

TODO: Mathematical definition of consistency properties for each of the models below. Information available in Matt's report.

3.8.1.1 Sequential Consistency (SC)

"The result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." [Quote by Lamport (1979), REF]

3.8.1.2 Total Store Order (TSO)

TODO

3.8.1.3 Partial Store Order (PSO)

TODO

3.8.1.4 Relaxed Memory Order (RMO)

TODO

3.8.1.5 Other Consistency Models

- Strict Consistency
- Causal Consistency
- Processor Consistency
- PRAM/FIFO Consistency
- Cache Consistency
- Slow Consistency
- Release Consistency
- Entry Consistency
- General Consistency
- Local Consistency
- TODO: More???

3.8.2 Bluecheck Tests

Even the basic consistency models exhibit vast amounts of non-determinism. The checker tool exhaustively enumerates all behaviour of a set of memory operations. A general-purpose constraint-solver (Yices, [REF]) is used to check the traces for inconsistencies.

TODO: Insert example of a code check here

TODO: Insert graph regarding the performance of the checker time vs number of instructions.

3.9 Running FreeBSD

TODO

3.10 Baremetal Benchmarks

TODO

3.11 Benchmarks on FreeBSD

TODO

Chapter 4

Consistency and Coherence

4.1 Formal Definition of 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.

4.1.1 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 4.1.9.

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.

4.1.2 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).

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.

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

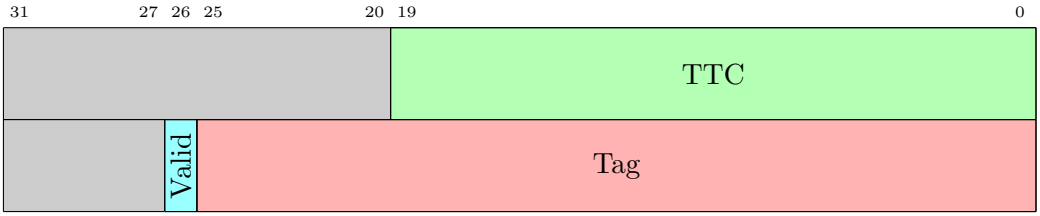


Figure 4.2: D-Cache Tag's, long TTC (16KB Size)

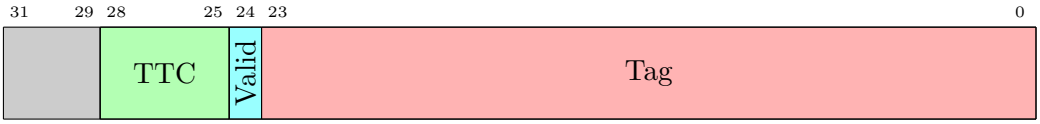


Figure 4.3: D-Cache Tag's, short TTC (64KB Size)

4.1.3 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.

4.1.4 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.

4.1.5 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.

4.1.6 AXE Litmus Tests

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.

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

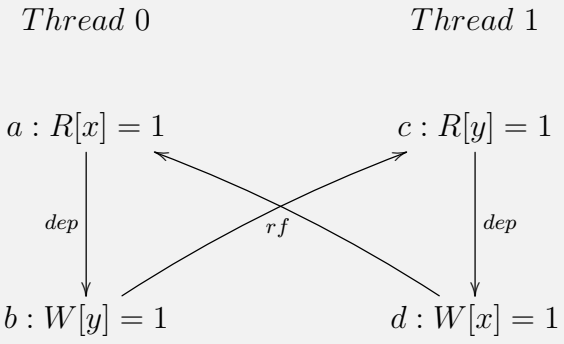


Figure 4.4: LB+addrs.axe

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
```

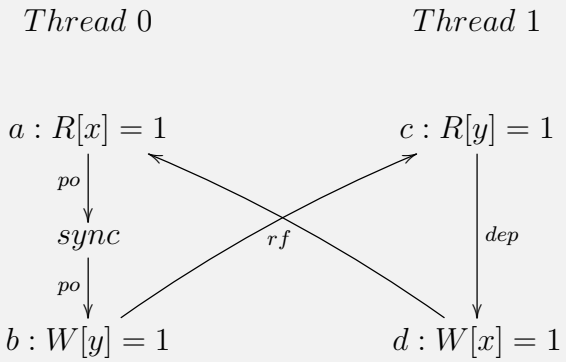


Figure 4.5: LB+addrs.axe

This example shows a cyclic dependency between all operations and bar-

ring 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.

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
```

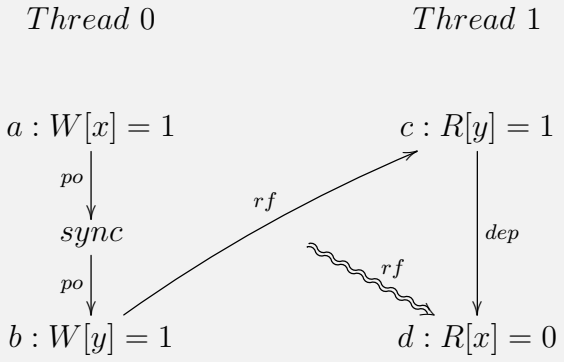


Figure 4.6: MP+sync+addr.axe

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
```

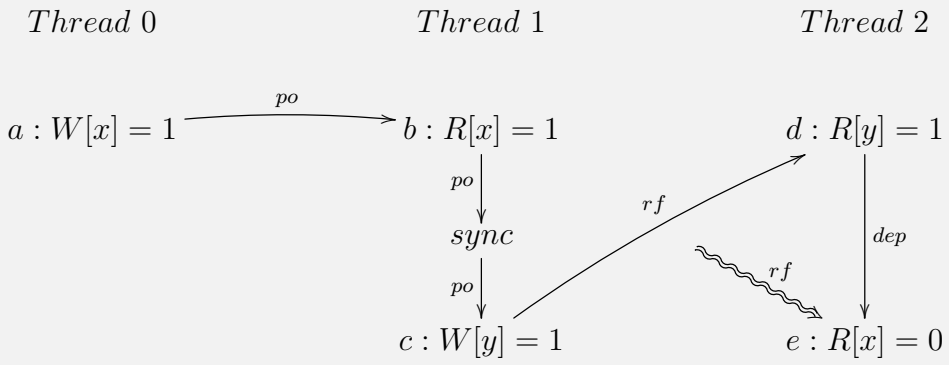


Figure 4.7: WRC+sync+addr.axe

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
```

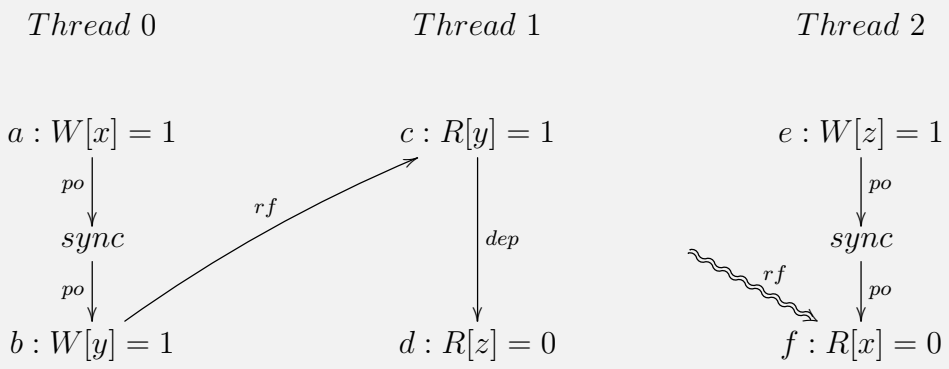


Figure 4.8: W+RWC+sync+addr+sync.axe

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.

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
```

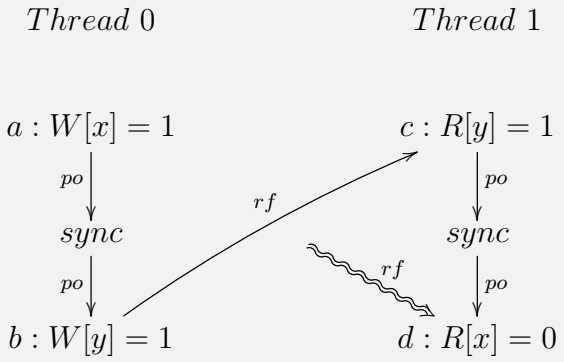


Figure 4.9: MP+syncs.axe

4.1.7 CHERI Litmus Tests

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

```
*       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

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

```
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

values as 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.

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

4.1.8 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 4.1.5].

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

4.1.9 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

4.1.10 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

4.1.11 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

4.2 Comparison with Other RMO Systems

TODO

4.3 Formal Definition of BERI Directory Coherence

I have tested several coherence mechanisms while developing BERI multicore. 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.

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.

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

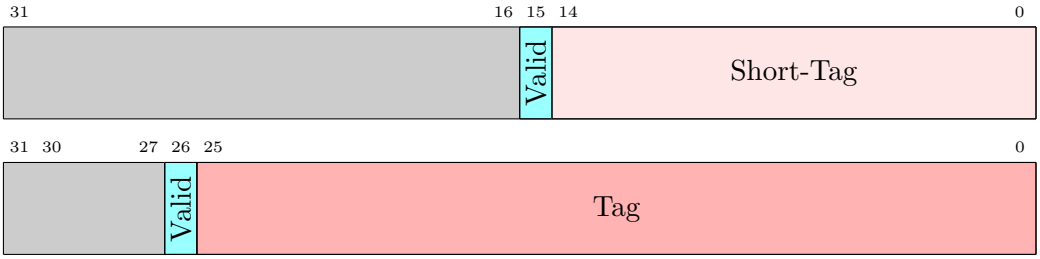


Figure 4.15: D-Cache Tag's, Dual-Core [Directory Coherence using Short-Tag optimization] (16KB Size)

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.

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

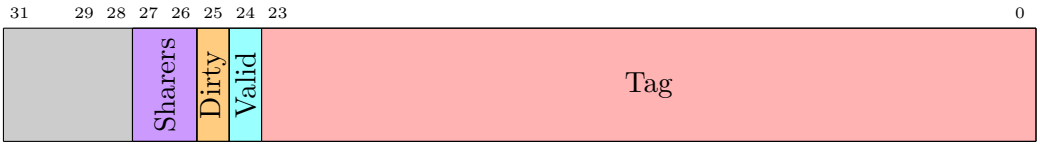


Figure 4.16: L2-Cache Tag's, Dual-Core [Directory Coherence for D-Caches] (64KB Size)

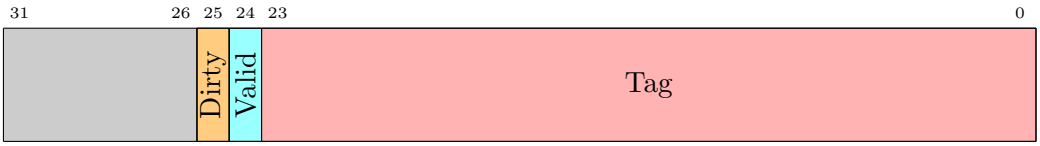


Figure 4.17: L2-Cache Tag's, Dual-Core [Time-Based Coherence] (64KB Size)

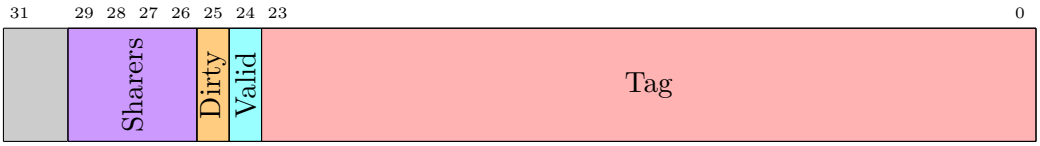


Figure 4.18: L2-Cache Tag's, Quad-Core [Directory Coherence for D-Caches] (64KB Size)

Data-Caches, we do not run JIT compiled code and Instruction-Cache coherence is not necessary.

4.3.1 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.

4.3.2 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).

4.3.3 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

4.4 Comparison of Overheads

Time-Based Coherence

- Extra control logic in the D-Cache.
- 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.
- Extra control logic in the L2 cache.
- 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). These lines leave the L2 cache as a broadcast and get distributed by the Multicore module.
- 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.

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 (TODO: Multiple build average)

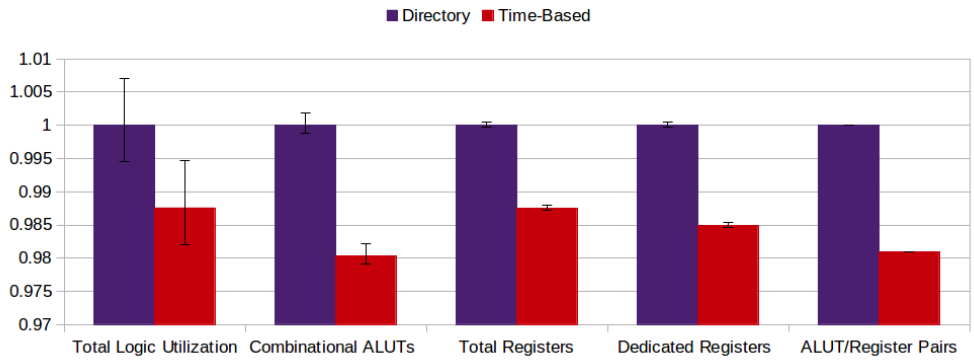


Figure 4.19: Quartus Overheads

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.

4.5 Application of Time-Based Coherence

TODO

Chapter 5

Coherence Results and Evaluation

5.1 Chapter Summary

- BERI software simulation and verification
- Bare metal BERI tests
- BERI test suite
- Hardware consistency verification, Bluecheck
- Splash-2 benchmarks on FreeBSD
- My parallel benchmark test suite (Should include tests with performance counters)
- False sharing performance tests???
- ...
- Bare metal victim process memory footprint capture (core pin and core split)
- FreeBSD victim process memory footprint capture (core pin and core split)

- AES bare metal side-channel attack
- AES FreeBSD side-channel attack

5.2 Splash-2 Benchmarks Background

The Splash-2 benchmark suite was used to evaluate and validate BERI modifications. This suite is no longer state of the art but it was selected for several reasons: written in C which is supported by CHERI LLVM, widely used in memory architecture research and evaluation, and the suite benchmarks are well understood.

The Splash-2 suite contains a number of applications and kernels based around scientific parallel computing. The tests heavily rely on the efficiency of shared block transfer. It is important to note that several tests in this benchmark suite pose similar requirements on the efficiency of the underlying architecture. Hence, it may not always be necessary to run the entire test suite in order to determine any performance variations [Reference:]. Several tests from the suite were selected for primary evaluation of the dual-core BERI design. The selected tests are sufficiently diverse and display many problem flavours. We draw parallels between our findings and the data presented in the papers in the results section.

5.2.1 Ocean

The pair of Ocean tests (Contiguous and Non-Contiguous) simulate the flow of ocean currents, representing an high performance compute (HPC) workload. Ocean dynamically allocates data in four dimensional arrays, designed to achieve good data distribution as well as reduce false sharing. Most of the application's execution time is spent on nearest-neighbour sweeps on a grid hierarchy (multi-grid solver). Due to the nature of grid hierarchy, higher grids have fewer points and smaller block transfer messages. The overheads per transfer are greater than those at the lower levels. As a result, little performance will be gained at lower levels through communication efficiency. Higher levels also have a higher communication to computation ratio. As

the problem size increases, the algorithm will experience substantial capacity and conflict misses [Ref 2]. The simulation tests architectural parameters such as memory bandwidth, communication overheads, and general processor execution speed.

"Regular-grid nearest-neighbour problems such as Ocean represent a dominant class of applications with communication that is quite natural to block transfer, but which do not benefit much from block transfer in a tightly-coupled multiprocessor [Ref ?]."

The Splash-2 suite contains many paired tests, mostly different algorithms for the same problem. Unlike other pairs, the two Ocean tests are very diverse. Each test uses a different style of inter-core communication. Analysis of the Non-Contiguous version has shown that around 50% of all memory references were shared data related [Ref 1][Ref 3]. Additionally, a third of the operations to shared data were writes [Ref 1]. This is unlike any other test in the suite.

[Ref 2: page 4] TODO: Memory access data table.

Ocean benchmarks observe a linear speedup with increasing number of cores [Ref 2]. The benchmark was evaluated in [Ref 2], on a Tango-Lite system simulator with a directory based Illinois cache coherence protocol. When simulated with 32 processors and a data cache structure (64 byte line size, 16KB capacity, direct mapped) similar to that used by BERI (32 byte line size, 16KB capacity, direct mapped), the test experiences a total cache miss rate of around 7%-8% [Ref 2]. The design is sufficiently different from BERI so we are not able to make a direct comparison, however, these results can be used as a point of reference for our data analysis. With an increase in processor number in Ocean the remote shared memory accesses and capacity related traffic show a proportional increase. Ocean shows a low migratory sharing pattern between cores and producer-consumer relationships are largely core local [Ref 3].

Subsections below are a direct reference to the README file

Ocean Contiguous "This implementation implements the grids to be operated on with 3-dimensional arrays. The first dimension specifies the pro-

cessor which owns the partition, and the second and third dimensions specify the x and y offset within a partition. This data structure allows partitions to be allocated contiguously and entirely in the local memory of processors that "own" them, thus enhancing data locality properties."

Processors	2
Grid Size	258 × 258
Grid Resolution	20000.00
Time Between Relaxations	28800
Error Tolerance	1e-07

Table 5.1: Ocean Contiguous test parameters

Ocean Non-Contiguous "This implementation implements the grids to be operated on with two-dimensional arrays. This data structure prevents partitions from being allocated contiguously, but leads to a conceptually simple programming implementation." The test parameters used were identical to those shown in table 5.1.

5.2.2 LU

This benchmark focuses on matrix triangulation and is an HPC application. The kernel factors a dense matrix. It is an example of a dense linear algebra calculation [Ref 2]. The matrix A(n × n) is divided into an N × N array of B × B blocks (n = NB). "Blocking is performed in LU to exploit temporal locality on individual sub-matrix elements." The blocking property results in the synchronisation time accounting for roughly 25% of the total execution time. LU does not scale linearly with an increasing number of processors, scaling is linear at 8 processors and drops off to half at 64 processors (Exact knowledge of the values is not critical to our research) [Ref 2].

The benchmark was modelled in the same environment as specified for Ocean. When simulated with 32 processors and a data cache structure (64 byte line size, 16KB capacity, direct mapped) similar to that used by BERI

(32 byte line size, 16KB capacity, direct mapped), the test experiences a total cache miss rate of around 2%. Remote sharing grows with the increase in the number of processors [Ref 2]. Noticeable fraction of the shared space is migratory data, this pattern is most evident when using 2 processors [Ref 3].

subsections below are a direct reference to the README file

LU Contiguous unique value "This implementation implements the matrix to be factored as an array of blocks. This data structure allows blocks to be allocated contiguously and entirely in the local memory of processors that "own" them, thus enhancing data locality properties."

Processors	2
Matrix Size	512×512
Element Blocks	16×16

Table 5.2: LU Contiguous test parameters

LU Non-Contiguous "This implementation implements the matrix to be factored with a two-dimensional array. This data structure prevents blocks from being allocated contiguously, but leads to a conceptually simple programming implementation." Unlike the two Ocean tests, LU tests are not quite as dissimilar, hence the second test was evaluated using a different matrix size.

Processors	2
Matrix Size	128×128
Element Blocks	16×16

Table 5.3: LU Non-Contiguous test parameters

5.2.3 Water

Water is an example of a molecular dynamics HPC application. Compared to Ocean and LU, only around 5% of all memory accesses are to shared data related [Ref 1]. This benchmark is available in two flavours: Nsquared and Spacial. Unlike other Splash-2 benchmarks, both Water Nsquared and Spacial use more memory locations for communication with more than 8 cores. A noticeable portion of the shared memory space is used for migratory data (Also: fmm, lu). In Water Nsquared almost all migratory locations are shared between all processors. Both Water's are involved in using the entire shared space for producer-consumer patterns. Broadcast techniques in an onchip interconnect or coherence protocol are likely to benefit Water's, may not affect other benchmarks as much [Ref 3]. Cache miss rates for 32 processors and the set up in [Ref 2] reveal an average of about 2% for a BERI equivalent configuration. Remote memory accesses increase significantly with an increase in processors [Ref 2].

Water Nsquared "This application evaluates forces and potentials that occur over time in a system of water molecules. The forces and potentials are computed using an $O(n^2)$ algorithm." [Ref 2: page 4]

Processors	2
Step Size	3
Problem Size	512

Table 5.4: Water test parameters

Water Contiguous "This application is similar to Water Nsquared, implemented with a more efficient algorithm. It is based on a O(n) computation. The advantage of the grid of cells is that processors which own a cell need only look at neighbouring cells to find molecules that might be within the cutoff radius of molecules in the box it owns. The movement of molecules

into and out of cells causes cell lists to be updated, resulting in communication." [Ref 2: page 4] The input test parameters for this test are identical to those shown in Table 5.4.

5.2.4 FFT

FFT is a widely used signal processing operation. It is typically used to convert from time to frequency domains. "The algorithm consists of n complex data points, and n root if unity complex data points. Both sets of data are arranged in $\sqrt{n} \times \sqrt{n}$ matrices, partitioned so every processor can access its portion of data in its local cache. The algorithm is optimised for low interprocessor communication. Interprocessor communication occurs once a local transpose has been computed [Ref 2]." The FFT algorithm scales almost linearly with the number of processors. The cache miss rate is approximately 4% for the setup described in [Section 5.2.1]. Since FFT is designed for low inter-core communication, the cache capacity, associativity, etc, are crucial for good performance. Small private caches may not accommodate all of the local data required which will lead to shared memory penalties. FFT exhibits a bursty communication pattern. The amount of remote data sharing levels off in the region of 8-64 processors [Ref 2]. FFT shows strong all to all communication [Ref 3]. There is only one version of FFT in the test suite, the benchmark was evaluated with two sets of parameters.

Processors	2
Complex Doubles	1024
Cache Lines	65536
Bytes per Line	16
Bytes per Page	4096

Table 5.5: FFT Small test parameters

FFT Small

Processors	2
Complex Doubles	16384
Cache Lines	65536
Bytes per Line	64
Bytes per Page	4096

Table 5.6: FFT Large test parameters

FFT Large

5.2.5 FMM

FMM is another HPC application. "The application simulates a system of bodies over a number of time steps. The interactions are simulated in a two dimensional format. The communication pattern of FMM is unstructured, and no attempt is made at intelligent distribution of particle data in main memory [Ref 2]." FMM scales well with the number of processor cores selected, but not as linear as FFT, Water's, and Ocean's. Using the test setup in [Section 5.2.1], the cache miss rate is around 4% [Ref 2]. FMM is subject to false sharing as particle data may reside in the same data line as the one used by another processor. " This effect is seen in Barnes and FMM. In both these programs, true sharing misses continue to drop with larger lines (though not linearly), and false sharing misses start to grow and outweigh the true-sharing reduction by about 128-byte lines. If cache lines are larger than a single record, false sharing across records may result. This is more likely in Water-Nsquared than in Barnes or FMM, since in the former a processors particles are contiguous in the array of records while in the latter the assignment of particles to processors changes dynamically so that a processors particles usually are not contiguous [Ref 2]." FMM shows increased neighbour communication. "We find that only 5 Splash-2 benchmarks (barnes, fmm, lu, and water-nsquared) use a noticeable fraction of their shared memory space for migratory data. The same applies to producer-consumer patters [Ref 3]."

Processors	2
Particles	256

Table 5.7: FMM-256 test parameters

FMM 256

Processors	2
Particles	2048

Table 5.8: FMM-2048 test parameters

FMM 2048

5.2.6 Radix

This application is used for sorting integers. "The algorithm is iterative, performing one iteration for each radix r digit of the keys. In each iteration, a processor passes over its assigned keys and generates a local histogram. The local histograms are then accumulated into a global histogram. Finally, each processor uses the global histogram to permute its keys into a new array for the next iteration. This permutation step requires all-to-all communication. The permutation is inherently a sender-determined one, so keys are communicated through writes rather than reads [Ref 2]." When using the setup descirbed in [Section 5.2.1], the cache miss rate for this benchmark is around 12-15%. Radix does not scale linearly with processors, "for Radix the poor speedup at 64 processors is due to a parallel prefix computation in each phase that cannot be completely parallelised [Ref 2]." "Radix streams through different sets of keys with both regular and irregular strides in two types of phases, and also accesses a small histogram heavily. This results in a working set that is also not sharply defined, and which may or may not fit in the cache [Ref 2]." Radix produces bursty communication traffic. "A much more dramatic example is provided by Radix. In the permutation

phase of this program, a processor reads keys contiguously from its partition in one array and writes them in scattered form (based on histogram values) to another array. The pattern of writes by processors to the second array is such that on average, writes by different processors are interleaved in the array at a granularity of keys, where n, r, and p are the number of keys being sorted, the radix, and the number of processors, respectively. While the exact pattern is dependent on the distribution of keys, whether or not we have substantial false sharing clearly depends on how $n/(r \times p)$ compares with the cache line size. We therefore see the sharing miss rate drop with line size, until this ratio is less than a line. At this point, the true sharing miss rate continues to drop while the false sharing miss rate rises dramatically, making large cache lines hurt performance [Ref 2]."

Processors	2
Keys	262144
Radix	1024
Max Key	524288

Table 5.9: Radix test parameters

5.2.7 Other Splash-2 Tests

TODO: why these were not used.

5.2.8 Combined results

TODO

5.2.9 Capability Enhanced Coherence

TODO

5.3 Parallel Benchmark Suite for BERI

In order to assess the parallel behaviour of both: directory and time-based coherence schemes on BERI, a parallel benchmark suite was created. The suite contains several flavours of benchmarks that are designed to highlight the pros and cons of each coherence mechanism. Algorithms have been selected from the examples provided in [Reference: http://www.cs.cmu.edu/~scandal/nesl/algorithms.html]

TODO: Run the same tests under X86 to highlight the validity of the algorithms and the speed up achieved.

5.3.1 Linear Scan Test

TODO: This is my own variant of the Tree Scan algorithm which is linear.

5.3.2 Block Sort Test

TODO

5.3.3 Pair Sort Test

TODO

5.3.4 Bathcher Sort Test

TODO

5.3.5 Quick Sort Test

TODO

5.3.6 Dense Linear Algebra Test

TODO

5.3.7 Prime Identity Test

TODO

5.4 Moldyn Benchmark

TODO

5.5 Pipe vs Pthread Benchmark

TODO

5.6 DD FreeBSD

TODO

5.7 CP FreeBSD

TODO

5.8 Side-Channel Attack Results

TODO Memory Footprint Extraction (MFE).

5.8.1 MFE - Bare Metal Simulation

TODO

5.8.2 MFE - FreeBSD on FPGA

TODO

5.8.3 SCA - Bare Metal Simulation

TODO

5.8.4 SCA - FreeBSD on FPGA

TODO

5.8.5 Capability Enhanced SCA Mitigation

TODO

Chapter 6

Cache Side-Channel Attacks

Side-Channel Attack(SCA)

6.1 ** TODO List **

- Use other SCA techniques to test
- Find a cache leakage tester program - DOES NOT EXIST, FUTURE WORK?
- Test AES attack successfully
- Quantify the effects of coherence/consistency on SCA
- Check that cache normalizing mitigates SCA in AES test case
- ...
- Use the Capability based coherence time tuning mechanism to improve side-channel attack mitigation

6.2 CHERI and SCA Mitigation

The CHERI architecture is designed to provide fine grained memory protection. Capabilities allow the OS to allocate a fixed memory chunk to a security

critical application. This chunk of memory is available exclusively to trusted processes. This mechanism provides robust protection again software based attacks such as buffer overflows. However, the capability model is helpless against SCA's.

An SCA is a software or hardware based attack that attempts to extract useful information by exploiting specific hardware behaviour. The attack does not seek weaknesses in software algorithms (in many cases cryptographic), instead it relies on physical properties of the system. System timing information, power consumption, electromagnetic emanation, are just some of the basic exploitable properties.

Various computations exhibit different power consumption. Tracing the power source can provide a detailed analysis of the power usage variation in the system. Power models can be used to analyse this trace and determine the computations performed by the system with a degree of probability. Similarly, an application can be used to memory latency variation. This fluctuation could determine the memory footprint of other applications ruining on the system. In this chapter we primarily focus on mitigating cache SCA's on CHERI.

6.2.1 Leaking data protected by Capabilities

Capabilities provide memory bounds for application data. This data is stored in memory just like any other data. Capabilities primarily influence the TLB translations of the data but do not affect the caches or main memory itself. Cap's are also stored in memory but with an additional Tag bit.

Cache SCA's attempt to determine the memory footprint of a particular application and thereby uncover the operations performed by this application. As far as the spy is concerned, the memory footprint of an capability protected application and a regular application is similar. Major differences will be linked to the TLB translation and any additional memory occupied by the capabilities themselves. If a cryptographic algorithm is performed by the system with capability protection, it will be safeguarded against memory leaks but not SCA's.

6.2.2 Cryptography and SCA's

The behaviour of a cryptographic algorithm is dependant on the input data, key, or often both. A variation in any of the parameters will result in a different set of cryptographic steps. Computing resources required by the algorithm can be limited in order to eliminate any information leakage though side-channels. However, the robustness of these techniques varies.

Encryption algorithms such as RSA, ElGamal, and Digital Signature Algorithm [Ref: wiki, timing attack] use modular exponentiation in one of the steps. This operation calculates the remainder (z) when as integer (x) is raised to the power (p) and divided by a positive integer (y). This algorithm is recursive and the input parameters will affect the memory usage.

TODO: More information here

6.2.3 State of the art SCA Mitigation

TODO

6.2.4 Solution: CHERI SCA Mitigation

Cache coherence largely affects the behaviour and shared memory usage of parallel applications. Traditional cache coherence mechanisms are designed for parallel performance and memory consistency. While the addition of a coherence scheme affects memory behaviour, it does nothing for concealing memory usage.

We have already introduced the concept of time-based cache coherence. This mechanism simple and easy to implement in the context of BERI architecture. Private caches systematically self-invalidate and thereby maintain coherence. An added side effect of this behaviour is a subtle masking of the memory footprint of a specific process.

TODO: Capability enhanced time-based coherence with SDA protection.

6.2.5 Related Work

TODO

6.3 BERI SCA Analysis

Modern processor designs incorporate multi-tired caches with a diverse behaviour. The BERI memory sub-system is fairly simple compared to commercial designs. Direct mapped caches, in-order execution, no write buffers, no memory access reordering, simple prefetching, and other factors make the BERI memory highly susceptible to side channel leakage.

6.4 Memory Footprint Analysis

A number of cache based memory attacks have been proposed and tested, the Prime+Probe attack is one of the stronger options. This attack is described in [TODO: References]. This attack has been described in the background chapter. Essentially the attacker plants a trojan in the desired system. The trojan's primary aim is to interfere with the cache behaviour and thereby extract information about processes running on the system. I have tested this attack on my system for both directory and Time-Based coherence protocols. A baremetal test and an FPGA, FreeBSD test were conducted.

TODO: Figure showing Prime+Probe attacks.

All of the testing described in this section is based on SCA analysis of hardware with no explicit SCA mitigation mechanisms.

Directory – more SCA An efficient directory coherence protocol should provides no additional protection against SCA's. The coherence protocol is designed to do the complete opposite, efficient caching of most frequently used data, updating stale memory lines, and eliminating false sharing of data. The Prime+Probe attack relies on the caching of any critical applications by the host. Since the critical application will evict trojan data during its run, the probe phase will reveal the memory usage.

The behaviour of a directory coherence system towards SCA's should not be any different to that of snooping coherence or even a single core system. Surprisingly, I have found that a multiprocessor system might be more vulnerable to a SCA as compared to a uniprocessor, at least when

using an OS. The OS uses the multi-threaded nature of the architecture to distribute workloads and achieve higher efficiency, as a result the probe phase timing data contains less noise due to other processes, interrupts, exceptions, etc. These results are futher illustrated in Section 6.4.3.1.

Time-Based – less SCA This coherence protocol adds unpredictability to cache behaviour by design. The protocol evicts data based on a set lifespan, clearing any stale data. Increasing the miss rate of a cache is not usually desirable, however, when dealing with side channels, memory entropy could be beneficial.

The Prime+Probe attack relies on controlled cache data eviction, random memory purging will not yield desired timing information. This is precisely what a Time-Based coherent system is able to achieve. When the attacker loads data into the cache during the Prime phase, each memory line is assigned a maximum lifespan value. When this value is exceeded the line is evicted. If sufficient time passes between the prime and probe phases, the probe stage will observe data misses and no hits. Thus, the coherence mechanism is introducing some SCA mitigation. The coherence mechanism will also have a fairly low impact on the critical application (TODO: Baremetal measurements of victim code performance overheads). The Time-Based coherence protocol has already shown good relative performance on our system, thus, no significant performance drop is expected for the critical application.

6.4.1 Memory Testing Granularity

I have crafted an side channel test that aims to identify the memory footprint of a given critical application. In order to simplify the analysis of the timing data and remove any unexpected noise due to other data in the cache, the critical application or victim is a simple loop containing memory load and store operations. The trojan program consists of two loops: prime and probe. The prime loop loads an array of data equivalent is size to the data cache and stores it repeatedly in a volatile variable, this ensures that the operation is not optimised away by the compiler, the loaded value is then updated

with a unique known value and stored back into the cache. The probe phase simply loads all the previously primed data, the execution time of this loop is timed, any fluctuations in total time will reveal the hit/miss ratio. The algorithm [TODO: Reference] is further described below and illustrated in Figure (TODO).

1. Cache warm up: Prior to running any trojan or victim operations, every cache byte is preloaded with a fixed value. This value is never used in either of the critical loops. The warm up processes simply puts the cache in a known state and eliminates any potential false memory sharing.

2. Prime: The trojan populates the entire data cache with its data.

3. Victim: The victim program is allowed to execute. This program loads and stores chunks of memory. The size of this memory can be adjusted. A larger chunk of memory would result in more trojan data evictions from the data cache.

4. Probe: The trojan loads all of its data and measures the execution time of the loop.

TODO: Figure – Memory footprint algorithm

The BERI data cache used in this evaluation is 16KB in size with 32 bytes per line. The data cache line is associated with one Tag only, thus, our analysis is limited to a line granularity. In order to reduce the execution time of the trojan, the program only loads and stores every 32'nd byte of its memory chunk. This ensures that each data cache line will contain at least one byte of trojan data. The exact placement of the trojan byte within the line is irrelevant, as any other memory access to this line will be a miss due to mismatched Tag's, resulting in trojan data eviction.

(TODO: Rephrase sentence) This test also operates at the granularity of the L2 cache, however, the victim memory chunk has been limited to the data cache size so in most cases the best trojan timing information is extracted when only the data cache is populated. Additionally Time-Based

coherence provides no protection for the shared cache, it has no control over L2 evictions. Side channel attacks on shared caches are discussed further in Section 6.6.2.

6.4.2 Baremetal Testing

The details of the baremetal SCA testing environment are listed below:

TODO: Describe testing environment in detail. Include critical compiler options and methodology. Omit anything that has already been described in the BERI architecture chapter.

In the baremetal environment we have access to all the hardware counters that are normally restricted to kernel space by the operating system. All the hardware counters implemented in the data cache are accessible to the test software. All counters are read through coprocessor 0 (CP0) and use the read hardware register (RDHWR) instruction. The data cache counters are fed into CP0 every cycle. Some of the counters mentioned be are memory architecture specific, since coherence model behaviour are different.

Counters used in our evaluation are:

1. **Time:** This counter follows the standard MIPS model.
2. **Miss:** Total data cache misses.
3. **Hit:** Total data cache hits.
4. **Invalidate:** Number of invalidate messages received by the data cache (Directory coherence only).
5. **Invalidate Hit:** Number of invalidate messages that matched a valid line in the cache (Directory coherence only). The combination of invalidate count and invalidate hit count shows the efficiency of the directory.
6. **Time-Based Miss:** Total number of misses caused through the data cache line expiry mechanism (Time-Based coherence only).

7. **Time-Based Hit:** Total number of hits in the data cache where the line was valid and the tag-time-counter had not expired (Time-Based coherence only).

6.4.2.1 Core Pinned Tests

Multiprocessor systems allow application parallelism, an important consideration when it comes to memory side channel evaluation. Many SCA's rely on the trojan and the victim application to be collocated; same private cache. Some attacks can extract useful information from a shared cache such as [TODO: Reference]. Our evaluation evaluates an SCA on processes sharing the same private cache, required for analysing Time-Based coherence mitigation properties. However, we also show how a victims memory operations can affect a trojan operating on a separate core. Most of this data is extracted through the shared cache.

The core pin test executes the trojan and victim code on BERI core 0. Core 1 is held in an infinite loop while the tests execute.

TODO: Figure showing test flow

6.4.2.2 Split Core Tests

TODO

6.4.3 OS Testing

TODO

6.4.3.1 Core Pinned Tests

TODO

6.4.3.2 Split Core Tests

TODO

6.5 AES Analysis

TODO

6.5.1 AES Algorithm

TODO

6.5.2 Why Test AES?

TODO

6.5.3 Results?

TODO

6.6 What Protection does Time-Based Coherence Provide?

TODO

6.6.1 Just a nuisance or real protection?

TODO

6.6.2 Protecting LLC

TODO

Does LLC benefit from this protection? TODO

How to protect LLC? TODO

Chapter 7

Conclusion

7.1 Field Contributions

- Exploration of cache coherence mechanisms
- Implementation and testing of directory coherence and time-based coherence
- New time-based relaxed memory consistency scheme, more relaxed than RMO. Requires a name, perhaps very relaxed memory order (V-RMO)
- Extensive testing and verification of V-RMO:
    1. Bluecheck verification
    2. Bare metal tests
    3. Hardware implementation of the processor
    4. FreeBSD supports the memory model
    5. Widely used benchmarks in FreeBSD (Splash-2, others)
- While time-based coherence is not a new concept, most of the research has focused on using it to supplement other stronger memory consistency schemes and coherence models. V-RMO has been used as a standalone coherence and consistency scheme

- Capability enhancement of V-RMO (* VERIFY *)
- ...
- Analysis of cache side-channel attacks on BERI
- Cache memory footprint detection on two coherence models; directory and time-based coherence, on the same architecture
- The side-channel attack test environment includes both software simulation (Bluesim) and a hardware implementation (BERI on FPGA, with FreeBSD)
- Time-based coherence assists with masking some of the cache side-channel leakage without any additional modification (* VERIFY *)
- Demonstration of a side-channel attack on AES in simulation and hardware, for BERI Directory and BERI Time-Based (* VERIFY *)
- Time-based coherence assists in masking side-channels for all applications and not just specific cryptographic operations (* VERIFY *)
- Capability enhancement of side-channel attack mitigation (* VERIFY *)

7.2 Engineering Contributions

- Implementation of BERI multi-core
    1. Core identification
    2. Memory interfaces
    3. Coherence mechanism
    4. LL/SC support
    5. IPC support
    6. Cache redesign

7. Testing and verification
8. Hardware synthesis

- Bringing up FreeBSD on multi-core BERI
  1. Coherence bugs
  2. Debugging scheme
  3. LL/SC bugs
  4. TLB debugging
  5. PIC debugging
- Tests created for multi-core BERI
  1. Bare metal
  2. OS compatible

References