Introduction

1 The Cache State Machine in SLICC

a) Transient States and their Event Triggers

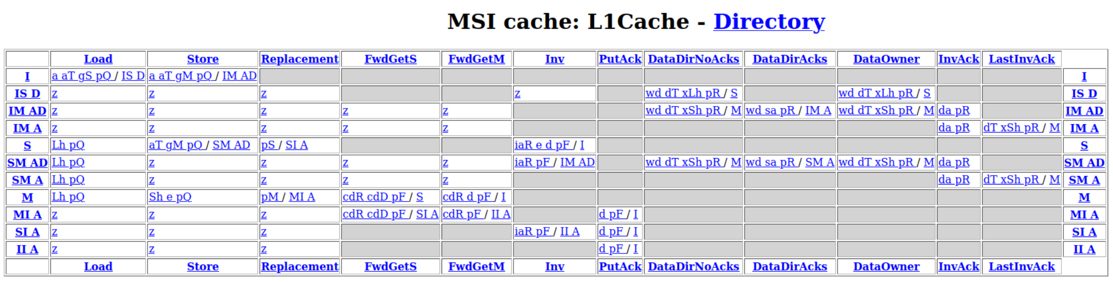
The main states of the MSI protocol are, of course, M, S and I. The rest are transient states.

As two examples we could examine the IS_D and MI_A states. These states stand for Invalid-to-Shared, Waiting for Data and Modified-to-Invalid, Waiting for PutAck, respectively.

According to the state transition table for L1 cache, the first transient state is triggered by a load while in the I-state. The table also allows us to see what actions are taken on this transition.

The second transient state is triggered by a *replacement* event (triggered when the CPU chooses the given cache block as victim) while in the M-state.

The full state table for cache events is shown in fig. 1.



MSI cache: L1Cache - Directory

	Load	Store	Replacement	FwdGetS	FwdGetM	Inv	PutAck	DataDirNoAcks	DataDirAcks	DataOwner	InvAck	LastInvAck	
I	a aT gS pQ / IS D	a aT gM pQ / IM AD											**I**
IS D	z	z	z			z		wd dT xLh pR / S		wd dT xLh pR / S			**IS D**
IM AD	z	z	z	z	z			wd dT xSh pR / M	wd sa pR / IM A	wd dT xSh pR / M	da pR		**IM AD**
IM A	z	z	z	z	z						da pR	dT xSh pR / M	**IM A**
S	Lh pQ	aT gM pQ / SM AD	pS / SI A			iaR e d pF / I							**S**
SM AD	Lh pQ	z	z	z	z	iaR pF / IM AD		wd dT xSh pR / M	wd sa pR / SM A	wd dT xSh pR / M	da pR		**SM AD**
SM A	Lh pQ	z	z	z	z						da pR	dT xSh pR / M	**SM A**
M	Lh pQ	Sh e pQ	pM / MI A	cdR cdD pF / S	cdR d pF / I								**M**
MI A	z	z	z	cdR cdD pF / SI A	cdR pF / II A		d pF / I						**MI A**
SI A	z	z	z			iaR pF / II A	d pF / I						**SI A**
II A	z	z	z				d pF / I						**II A**
	Load	**Store**	**Replacement**	**FwdGetS**	**FwdGetM**	**Inv**	**PutAck**	**DataDirNoAcks**	**DataDirAcks**	**DataOwner**	**InvAck**	**LastInvAck**	

Figure 1: MSI L1cache state-table

b) Interpret SLICC Code

According to the reference material, the SLICC code in the example can be interpreted as: if the current state is `SM_AD` or `SM_A` then upon the events: Store, Replacement, FwdGetS, FwdGetM, the action taken should be to stall. There is no destination state in the syntax of this transition function so the state does not change.

c) Necessity of Transient States and Reason for Stalling

Transient states are needed when a transition between two stable states can not proceed without the occurrences of some external event. For instance, the IS_D transient state waits for a D(ata) message to arrive for the particular cache block before it can transition to a new state. According to the state table, two events satisfy this criteria: DataDirNoAcks and DataOwner. Either will allow a state transition to S.

Other event requests (i.e. load, store, replacement...) to this cache block are stalled. That is, they can not be serviced in the current state and are placed in a FIFO.

d) Deadlocks and how to avoid them

Since the cache coherence protocol allows for the controller to stall until an external event occurs, there must be guarantees that the external event will, in fact, occur. Otherwise there is the potential for resource starvation. If there is a cyclic dependency of controllers stalling and thus not issuing the needed events, deadlock occurs.

2 Validating the Protocols via the Ruby Random Tester

We performed a run of ruby_random_tester.py -n 2 in order to test the MSI protocol with 2 CPUs. The simulation completed successfully on repeated runs.

```
Exiting @ tick 176341 because Ruby Tester completed
```

3 Analysing Coherence using Multithreaded Codes

The `threads.c` code spawns two threads to independently multiply two vectors (that is, they are fully parallelisable).

The default implementation alternates between the threads on each vector element.

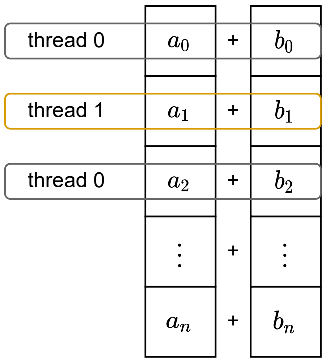


Figure 2: Threaded application with naive approach

By outputting the cache coherency protocol events we can observe how the cache is behaving. (Note: when outputting the protocol trace, `uniq` was used to filter duplicate lines generated in the MSI case. E.g.

```
./build/ARM_MSI/gem5.opt --debug-flags=ProtocolTrace
./configs/simple_ruby/simple_ruby_MSI.py | uniq > threads_MSI_trace.log
```

)

a) Problems with *threads.c*

Since we have a thread on each core, each executing `c[i] = a[i] + b[i]` on each alternate element, the threads are accessing independent data that are, by their proximity, nevertheless within a single cache-block. This leads to situations in which a core writes `c[i]`, putting the cache-block into the modified state, followed by the *other* core reading the independent variables `a, b` from within the same cache-block.

Because the cache-coherency protocol operates on the granularity of cache-blocks, reading a modified block triggers a synchronization event.

So, in summary, even though `a, b` do not change, reads from them trigger unnecessary coherency synchronization traffic of data that is not used, i.e. false-sharing. The solution is to make sure that the threads are not both reading/writing to data within the same cache-block, to the extent possible.

b) Improved *threads.c*

In order to avoid false sharing and to utilise the cache efficiently we decided to split the code into two halves: the first thread operates on the first 500 elements and the second thread on the last 500 elements. When using the MSI protocol, our modification yielded a speedup of 4% and eliminated about 3000 replacements; the number of loads, however, was barely affected.

A naive implementation is to change that access pattern to contiguous accesses in different memory regions for each thread, as in the following example:

```
if (tid == 0){
   for (int i = 0; i < num_values/2; i++){
      c[i] = a[i] + b[i];
    }
}
if (tid == 1){
  for (int i = num_values/2; i < num_values; i++){
     c[i] = a[i] + b[i];
  }
}
```

c) MI vs. MSI

In general, the MI protocol outperforms the MSI-protocol when the amount of data that is shared is negligible, due to the lower protocol-related overhead coherency traffic. In this case, the MSI-protocol introduces overhead by its additional complexity compared to the MI-protocol.

The improvement made in b) resulted in a speedup of 30% for the MI-protocol, since threads are no longer invalidating each others caches, making it marginally faster than the MSI-protocol (0.64% speedup over MSI). This may be within the margin of error, however.