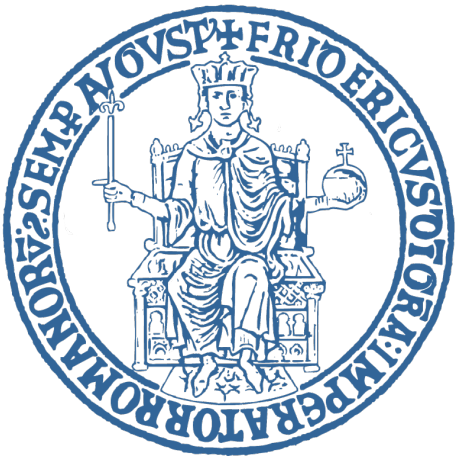
Università degli Studi di Napoli

Federico II



Dipartimento di Ingegneria Elettrica e

delle Tecnologie dell'Informazione

Scuola Politecnica e delle Scienze di Base

Corso di Laurea Magistrale in Ingegneria Informatica

Sviluppo di un Protocollo di Coerenza di Cache Expiration-based nel Simulatore Gem5

Studente:

Emanuele Di Maio M63001427

Docente:

Prof. Alessandro Cilardo

Anno Accademico

2024/2025

Indice

Capitolo 1

Introduzione

1.1 Abstract

Il crescente utilizzo di architetture multi-core ha portato a una crescente necessità di sviluppo di protocolli di coerenza più complessi. In particolare, gli attuali protocolli oggi utilizzati, come MESI, risultano scarsamente scalabili a causa di meccanismi di invalidazione e memorizzazione di maschere dei core associati ad ogni linea. In questo paper presentiamo lo sviluppo e l'implementazione di un nuovo protocollo di coerenza delle cache ottimizzato per ambienti multi-core nel simulatore gem5, ampiamente utilizzato per la simulazione di sistemi a livello architetturale. Il protocollo proposto affronta limitazioni comuni dei suddetti modelli esistenti, eliminando la necessità di utilizzare meccanismi di invalidazione delle linee e dell'utilizzo delle maschere dei core, migliorando così la scalabilità spaziale e temporale del protocollo, in scenari di alta contesa dei blocchi.

1.2 Il Simulatore Gem5

Il simulatore gem5 è una piattaforma modulare basata su eventi discreti, ampiamente utilizzata per la ricerca nell'ambito dell'architettura dei calcolatori. Grazie alla sua struttura modulare, gem5 consente di riconfigurare, parametrizzare, estendere o sostituire facilmente i suoi componenti, adattandosi

a diverse esigenze di simulazione. Simula il passare del tempo come una serie di eventi discreti, permettendo la creazione di ambienti complessi e realistici.
gem5 è in grado di simulare uno o più sistemi di calcolo, supportando architetture eterogenee come Alpha, ARM, MIPS, Power, SPARC, RISC-V e x86 a 64 bit. Può operare in full system mode (FS mode), simulando dispositivi e sistemi operativi completi, o in syscall emulation mode (SE mode), limitandosi all'esecuzione di programmi nello spazio utente. Il simulatore è scritto principalmente in C++ e Python e offre flessibilità nel costruire sistemi di memoria, utilizzando cache e crossbar o il modulo Ruby per una modellazione avanzata.
Nonostante gem5 offra molte funzionalità preconfigurate, è pensato per essere espandibile, rendendolo particolarmente adatto per progetti di ricerca innovativi che richiedono la creazione o la modifica di componenti esistenti.

1.3 I Protocolli di Coerenza in Gem5

In questo lavoro, ci concentreremo in particolare sul modulo Ruby, che offre una simulazione dettagliata e realistica del comportamento delle interconnessioni e dei sistemi di memoria a livello dei processori. Ruby permette di modellare con grande flessibilità la gerarchia della memoria e i protocolli di coerenza della cache, rendendolo uno strumento ideale per lo sviluppo e la valutazione di nuovi protocolli. Nel nostro studio, utilizzeremo Ruby per implementare il nuovo protocollo di coerenza della cache proposto, che rappresenta il contributo principale di questo paper.

1.4 Workflow

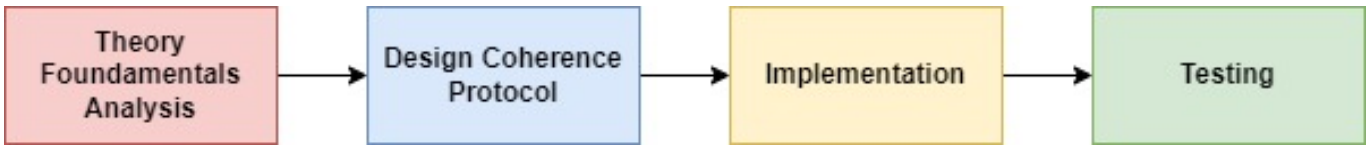


Figura 1.1: Workflow generale

Il workflow è articolato in quattro fasi, come mostrato nella Figura 1.1:

- **Analisi dei Fondamenti Teorici**: si parte dall'analisi di un protocollo di coerenza di base già presente in letteratura. In questa fase, verranno approfondite le nozioni teoriche e il protocollo sarà esteso per adattarsi a scenari reali. Questo include l'assunzione di transazioni di tipo pipelined o split-transaction e un modello di consistenza della memoria più rilassato, adatto per i processori moderni.

- **Progettazione del Protocollo di Coerenza**: basandosi sulle assunzioni teoriche definite nella prima fase, si procederà con la progettazione di un nuovo protocollo mediante la definizione di macchine a stati finiti, messaggi ed eventi.

- **Implementazione**: gli elementi progettati nella fase precedente saranno tradotti nella sintassi del linguaggio del simulatore gem5. In questa fase, verranno implementati i dettagli tecnici necessari per il funzionamento del protocollo all'interno del simulatore.
- **Testing**: si passerà quindi alla configurazione di un sistema multicore nel simulatore per eseguire workload multithread. Tali test verranno utilizzati per analizzare diversi casi d'uso e valutare le prestazioni del protocollo progettato.

Capitolo 2

Il Simulatore Gem5

Il protocollo progettato, verrà implementato e testato con il simulatore gem5. Esso è un simulatore a grana molto fine, che permette di eseguire workload o testare configurazioni di sistema con architetture di calcolatori personalizzate. È possibile simulare architetture: x86-64, ARM, RISC-V, Alpha, MIPS, Power, SPARC. A differenza di altri simulatori che permettono solo di eseguire applicazioni, gem5 permette anche di modificare ISA, processore, memoria e altri componenti.

- Eseguire workload per diverse ISA.
- Eseguire un sistema operativo o applicazioni bare-metal.
- Definire la configurazione del sistema e le caratteristiche dei suoi componenti (clock del processore, numero di processori, tipo di memoria DDR, velocità, periferiche).
- Definire la topologia di interconnessione dei core dei processori (sia multicore, sia multicomputer).
- Modificare la struttura interna del core di un processore.
- Definire la gerarchia delle cache e le loro caratteristiche (grandezza, velocità, interconnessioni).
- Definire un protocollo di coerenza delle cache.
- Modificare un ISA esistente.

2.1 Accessi in Memoria

Per continuare la trattazione con i tipi di processori disponibili, è necessario distinguere i tipi degli accessi alla memoria definiti in gem5:

- **Atomic**: quando viene effettuata una richiesta di accesso alla memoria, viene restituito immediatamente il risultato e un tempo di accesso approssimato. Questi tipi di accesso sono inadatti se si vuole testare un sistema di memoria realisitica.
- **Functional**: viene utilizzato dal simulatore per caricare i binari dei programmi o del sistema operativo da eseguire. Non apporta contributi alla latenza di simulazione.

- **Timing**: modella in maniera fedele alla realtà accessi alla memoria di tipo pipelined o split-transaction, infatti tiene conto della gerarchia delle cache, la latenza dei singoli livelli della gerarchia, la latenza della memoria e la latenza dovuta alla contesa delle risorse.

2.2 CPU

Gem5 offre diversi tipi di processori che possono essere utilizzare per eseguire le proprie applicazioni. La differenza tra i diversi tipi di processori riguarda la fedeltà di esecuzione rispetto a un processore reale:

2.2.1 AtomicSimpleCPU

È il processore più semplice, infatti non è presente una pipeline in quanto le istruzioni vengono eseguite immediatamente, e gli accessi in memoria vengono eseguiti in maniera atomica.
Pipeline: assente, le istruzioni vengono eseguite in 1 colpo di clock;
Memory access: atomic;
Scopo: utilizzato per testare la funzionalità dei programmi, senza tener conto di assunzioni architetturali.

2.2.2 TimingSimpleCPU

È il processore che introduce accessi alla memoria tempificati mentre restano le istruzioni eseguite immediatamente senza pipeline.
Pipeline: assente, le istruzioni vengono eseguite in 1 colpo di clock;
Memory access: timing;
Scopo: utilizzato per testare il comportamento della memoria e della gerarchia di cache.

2.2.3 MinorCPU

È un processore dettagliato, costituito da una pipeline che esegue le istruzioni in-order e gli accessi alla memoria sono tempificati.
Pipeline: 4-stage in-order pipeline: Fetch1, Fetch2 (branch predictor), Decode, Execute;
Memory access: timing;
Scopo: utilizzato per analizzare in maniera dettagliata il comportamento del processore e della memoria, ogni stadio della pipeline è caratterizzato da un certo numero di cicli (configurabile) per completare l'operazione.

2.2.4 O3CPU

È un processore dettagliato, costituito da una pipeline che esegue le istruzioni out-of-order e gli accessi alla memoria sono tempificati.
Pipeline: 5-stage out-of-order pipeline: Fetch, Decode, Rename, Issue-Execute-Writeback, Commit;
Memory access: timing;

Scopo: utilizzato per analizzare in maniera dettagliata gli effetti di un workload sulla memoria, gerarchia di cache e componenti interni del processore. Ogni stadio della pipeline è caratterizzato da un certo numero di cicli (configurabile) per completare l'operazione.

2.2.5 KvmCPU

È un modello di processore che riesce a sfruttare la tecnologia KVM del sistema host per accelerare la simulazione. Per utilizzarlo, è necessario che l'host e il guest si basano sulla stessa architettura: x86 oppure ARM.
Pipeline: -;
Memory access: -;
Scopo: utilizzato per accelerare alcune fasi di esecuzione del sistema, ad esempio l'avvio di un sistema operativo non necessario ai fini del workload.
Dunque, l'AtomicSimple e il TimingSimple risultano essere processori poco fedeli alla realtà, ma la cui esecuzione è molto più veloce rispetto ai processori Minor e O3, caratterizzati invece da un maggior livello di dettaglio. Per il nostro scopo, dato che la nostra enfasi è posta sulla gerarchia di cache, il modello TimingSimpleCPU risulta essere sufficiente per testare il comportamento del protocollo.

2.3 Ruby

2.3.1 Ruby overview

Ruby implementa un modello di simulazione dettagliato per il sottosistema di memoria. Modella gerarchie di cache inclusive/esclusive con varie politiche di sostituzione, implementazioni di protocolli di coerenza, reti di interconnessione, controller DMA e di memoria, oltre a diversi sequenziatori che avviano richieste di memoria e gestiscono le risposte. I modelli sono modulari, flessibili e altamente configurabili. Tre aspetti chiave di questi modelli sono:
Separazione delle responsabilità: ad esempio, le specifiche del protocollo di coerenza sono separate dalle politiche di sostituzione e dalla mappatura degli indici della cache, mentre la topologia della rete è definita separatamente dall'implementazione.
Configurabilità: quasi ogni aspetto che influisce sul funzionamento e sul timing della gerarchia di memoria può essere controllato.
Prototipazione rapida: un linguaggio di specifica ad alto livello, SLICC, viene utilizzato per definire la funzionalità dei vari controller.

2.3.2 SLICC

SLICC sta per Specification Language for Implementing Cache Coherence. È un linguaggio specifico per il dominio utilizzato per definire i protocolli di coerenza della cache. In sostanza, un protocollo di coerenza della cache si comporta come una macchina a stati, e SLICC viene utilizzato per specificarne il comportamento.

Poiché l'obiettivo è modellare l'hardware nel modo più accurato possibile, SLICC impone vincoli sulle macchine a stati che possono essere definite. Ad esempio, può limitare il numero di transizioni che possono avvenire in un singolo ciclo.
Oltre alla specifica dei protocolli, SLICC integra anche alcuni componenti del modello di memoria. Come mostrato nell'immagine seguente, la macchina a stati riceve input dalle porte di ingresso della rete di interconnessione e mette in coda l'output alle porte di uscita della rete, collegando così i controller di cache/memoria direttamente alla rete di interconnessione.

2.4 Simulazione

Per testare il protocollo, è necessario eseguire un workload. Tale workload può essere eseguito tramite due modalità di simulazione differenti:
Syscall Emulation: i programmi del workload eseguono senza il supporto di un sistema operativo, utilizzando delle system call emulate, fornite dal simulatore stesso. Il vantaggio è un avvio molto più veloce dato che non viene utilizzato un vero e proprio sistema operativo, ma meno affidabili rispetto ad esecuzioni reali.
Full System Emulation: i programmi del workload eseguono con il supporto di un vero sistema operativo e scheduler, col vantaggio di ottenere un'esecuzione più realistica, ma con lo svantaggio di un avvio estremamente lento. Infatti, prima di poter eseguire i programmi, è necessario avviare tutto il sistema operativo. Inoltre, bisogna configurare anche tutti i componenti hardware.

2.5 Struttura di Gem5

Comprendere il contenuto delle directory del simulatore gem5 è estremamente importante per capire dove e cosa andare a modificare per implementare il protocollo. La struttura delle directory si articola come segue:

- build: contiene la build del simulatore.
- build_opts: contiene dei file di configurazione per la build.
- build_tools
- configs: contiene diverse configurazioni di sistema per avviare il simulatore.
- ext
- include
- m5out: contiene le statistiche e contatori dell'ultima esecuzione del simulatore.
- site_scons
- **src**: contiene tutti i sorgenti del simulatore.
- system

- tests: contiene i programmi di test da eseguire nel simulatore.
- util: contiene funzioni di utilità per il simulatore.

La directory **src** è estremamente importante dato che contiene tutti i sorgenti del simulatore. Lavoreremo principalmente in questa directory. Essa si articola come segue:

- arch: contiene tutti i sorgenti per implementare gli aspetti specifici delle architetture supportate. Ad esempio, è possibile trovare l'ISA, l'implementazione del KvmCPU per x86.
- base: contiene i sorgenti che definiscono strutture dati di utilizzo comune nel simulatore.
- cpu: contiene i sorgenti per implementare i diversi tipi di CPU già enunciati. Il processore O3CPU è presente nella sottodirectory o3, il processore KvmCPU nella sottodirectory kvm, il processore MinorCPU nella sottodirectory minor, TimingSimpleCPU nella sottodirectory simple.
- dev: contiene i sorgenti per implementare molteplici device, ad esempio I2C, i device virtio e device specifici dell'architettura x86, come il SouthBridge.
- doc
- doxygen
- gpu-compute
- kern
- learning_gem5: contiene i sorgenti dei tutorial presenti sul sito di gem5.
- mem: contiene tutti i sorgenti inerenti alla memoria e alla gerarchia delle cache. In particolare, la sottocartella ruby sarà di fondamentale importanza.
- proto
- python: contiene i sorgenti dei componenti hardware già implementati da gem5 pronti per essere utilizzati.
- sim
- sst
- systemc

Capitolo 3

Design

3.1 Il Protocollo Tardis e il Modello Sequential Consistency

Il primo protocollo che analizzeremo è stato proposto in letteratura assumendo un modello di consistenza di tipo Sequential Consistency. Sebbene quest'assuzione semplifichi parecchio la progettazione, essa risulta estremamente inefficiente per contesti reali.

3.1.1 Il Modello Sequential Consistency

La Sequential Consistency (coerenza sequenziale) è un modello di coerenza per sistemi concorrenti, come sistemi di memoria condivisa o multiprocessori, che definisce un comportamento prevedibile e intuitivo per l'esecuzione di operazioni.
Definizione Un sistema è sequentially consistent se l'esecuzione delle operazioni (letture e scritture) da parte di tutti i processi è tale che:

- L'ordine delle operazioni all'interno di ciascun processo è rispettato. Ciò significa che le operazioni di un singolo processo appaiono nell'ordine definito dal programma.
- Le operazioni di tutti i processi si combinano in un unico ordine globale (serializzazione), che è rispettato da tutti i processi.

In altre parole, si può pensare che tutte le operazioni di memoria accadano in un certo ordine (sequenziale) che sia consistente con l'ordine specificato nei programmi dei singoli processi.
Questo modello è stato inizialmente proposto e formalizzato da Lamport. Un programma parallelo è sequentially consistent se:
"the result of any execution is the same as if the operations of all processors (cores) were executed in some sequential order, and the operations of each individual processor (core) appear in this sequence in the order specified by its program"
Più formalmente, siano $<_p$ e $<_m$ operatori che detonato l'ordine secondo il programma e l'ordine della memoria globale, allora il modello sequential consistency rchiede che siano rispettate le seguenti regole:

- **Regola 1**: $X <_p Y \Rightarrow X <_m Y$

- **Regola 2**: Valore di L(a) = Valore di $Max_{<m}\{S(a)|S(a) <_m L(a)\}$, dove L(a) e S(a) sono operazioni di Load e Store rispettivamente all'indirizzo a.

La Regola 1 enuncia che se un'operazione X (Load o Store) avviene prima di un'altra opearzione Y (Load o Store) nell'ordine del programma in un qualsiasi core, allora X deve precedere Y nell'ordine della memoria globale. La Regola 2 afferma che una Load a un indirizzo a deve restituire il valore della *più recente* store all'indirizzo a.

Esempio

Supponiamo due processi P_1 e P_2, e due variabili condivise A, B inizializzate a 0.
Sia così definito l'ordine del programma:

- Processo P_1
  1. Scrive A = 1
  2. Legge B
- Processo P_2
  1. Scrive B = 1
  2. Legge A

Secondo la sequential consistency, l'esecuzione deve essere equivalente a un'ordine sequenziale globale, che rispetta l'ordine delle operazioni all'interno di ciascun processo. Un possibile interleaving che rispetta la sequential consistency potrebbe essere:

1. P_1: Scrive A = 1
2. P_2: Scrive B = 1
3. P_1: Legge B = 1
4. P_2: Legge A = 1

In questo caso:

- Quando P_1 legge B, vede il valore scritto da P_2
- Quando P_2 legge A, vede il valore scritto da P_1

Supponiamo invece che l'ordine sia il seguente:

1. P_1: Scrive A = 1
2. P_2: Legge A = 0
3. P_2: Scrive B = 1
4. P_1: Legge B = 0

In questo caso, P_2 legge un valore di A (0) che precede l'operazione di scrittura di P_1, e P_1 legge un valore di B (0) che precede la scrittura di P_2. Questo scenario viola la sequential consistency, poiché le operazioni non sono visibili in un ordine coerente con un'interleaving globale.
In definitiva, la Sequential Consistency si basa sulle seguenti 4 condizioni fondamentali di precedenza degli accessi alla memoria:

- $L(a) <_p L(b) \Rightarrow L(a) <_m L(b)$
- $L(a) <_p S(b) \Rightarrow L(a) <_m S(b)$
- $S(a) <_p S(b) \Rightarrow S(a) <_m S(b)$
- $S(a) <_p L(b) \Rightarrow S(a) <_m L(b)$

Tuttavia, la Sequential Consistency non garantisce necessariamente la freschezza immediata o l'unicità del risultato, ma solo che il comportamento percepito sia conforme a una possibile sequenza seriale.
In letteratura sono stati presentati altri modelli di coerenza quali:

- **Strict Consistency**: Più forte, richiede che ogni operazione sia immediatamente visibile a tutti i processi.
- **Relaxed Consistency**: Più debole, consente ottimizzazioni come buffering o riordino per migliorare le prestazioni.

La Sequential Consistency è un via di mezzo, poiché un'avvenuta operazione non riflette i suoi effetti immediatamente sul sistema globale come la Strict Consistency, ma non è così rilassata da permettere il riordino delle operazioni globali come nelle Relaxed Consistency.

3.1.2 Tardis Sequential Consistency

Il protocollo Tardis descritto in letteratura è un protocollo di coerenza delle cache basato su directory e timestamp. L'obiettivo è quello di rendere il protocollo maggiormente scalabile rispetto ai classici MSI e MESI, riducendo la quantità di informazioni memorizzate nella directory e il traffico sulle interconnessioni, eliminando totalmente il meccanismo di Invalidate classico di MSI e MESI. Infatti, considerando un processore a 128 core, ciascuno con una cache di primo livello privata che utilizza un protocollo MSI o MESI per la coerenza, la directory deve tenere traccia degli sharer utilizzando una maschera a 128 bit, col risultato di memorizzare per ogni linea di cache, un gran numero di bit degli sharer, nell'ordine di O(n). Inoltre, in caso di una transizione Shared -> Modified State di una linea cache, è necessario inviare un numero di messaggi di Invalidate pari al numero di di core che in quel momento la condividono (Shared State). Il protocollo Tardis risolve tali problemi, ma vedremo che la sua prima versione descritta in letteratura, con l'assuzione del modello Sequential Consistency, lo rende inadatto a contesti reali, in quanto pone troppe limitazioni alle ottimizzazioni effettuabili.

3.1.2.1 Ordinamento delle Operazioni di Memoria

L'ordinamento delle operazioni di memoria sono rinforzate tramite il tempo globale fisico tali che, sia X e Y due operazioni di memoria e sia una di esse un'operazione di Store, allora
$X <_m Y \Rightarrow X <_{pt} Y$
Ciò significa che se l'operazione X è stata vista prima dalla memoria globale, allora essa è accaduta prima secondo il tempo globale fisico. Noi romperemo la correlazione tra l'ordinamento della memoria globale e l'ordinamento secondo il tempo fisico per le dipendenze WAR, mentre manterremo la correlazione per le WAW e RAW:

- $S_1(a) <_m S_2(a) \Rightarrow S_1(a) <_{pt} S_2(a)$
- $S(a) <_m L(a) \Rightarrow S(a) <_{pt} L(a)$
- $L(a) <_m S(a) \not\Rightarrow S(a) <_{pt} S(b)$

In altri termini, se la memoria vede secondo un certo ordine due Store successive o una Store seguita da una Load, allora esse sono avvenute nello stesso ordine secondo il tempo fisico, mentre per Load seguite da Store (WAR), l'ordinamento non è rinforzato. Tardis riesce ad ottenere tale ordinamento assegnando esplicitamente un timestamp ad ogni operazione di memoria, che indica il loro ordine globale di memoria.
Specificamente, l'ordinamento della memoria globale in Tadis è definito come la combinazione del tempo fisico e un tempo logico:
$X <_m Y := X <_{ts} Y or (X =_{ts} Y \text{ and } X <_{pt} Y)$
Questa espressione enuncia che se un'operazione X è stata osservata dalla memoria prima dell'operazione Y, allora vuol dire che:

- il timestamp di X è più piccolo del timestamp di Y, cioé X è avvenuto prima di Y secondo l'ordinamento mediante i timestamp;
- oppure che i timestamp di X e Y coincidono, e X è avvenuto prima di Y secondo il tempo fisico.

Applicando tale definizione alla Regola 1 della Sequential Consistency, essa diventa:
$X <_p Y := X <_{ts} Y \text{ or } (X =_{ts} Y \text{ and } X <_{pt} Y)$
Inoltre, assumendo che i processori effettuano le operazioni di commit sempre in-order, garantiamo la condizione per l'ordinamento rispetto al tempo fisico: $X <_p Y \Rightarrow X <_{pt} Y$
Quindi Tardis deve garantire solo la condizione di ordinamento mediante i timestamp: $X <_p Y \Rightarrow X \leq_{ts} Y$

3.1.2.2 Timestamps

Tardis, per mantenere tale ordinamento fa uso del meccanismo dei timestamp, i quali altro non sono che dei contatori logici da assegnare alle operazioni di memoria. Viene utilizzato un timestamp globale al core detto program timestamp (pts), che rappresenta l'ultima operazione di memoria avvenuta secondo l'ordine del programma. Nel caso di cache di 1° livello private, ad ogni blocco di cache vengono assegnati due timestamp, come in Figura 3.1:

- **read timestamp**: rappresenta un lease time, ovvero un intervallo di tempo (logico) entro il quale il core può leggere liberamente la linea. Esso quindi rappresenta il tempo logico della scadenza della linea di cache. Quando pts > rts, allora il blocco scade, ed è necessario riacquisirla dal LLC.

- **write timestamp**: rappresenta il timestamp dell'ultima store che è avvenuta in questo blocco.

È necessario osservare che il pts non è né il tempo fisico né il clock del processore. Esso verrà incrementato durante l'esecuzione del programma secondo un paradigma differente. Il rts, in accordo alla Regola 2, non deve essere inferiore al pts, quindi in caso di Load l'rts deve essere aggiornato nel seguente modo: rts := max(pts, wts). Inoltre, il wts, in accordo alla Regola 2, deve essere maggiore dell'rts, pertanto anch'esso verrà aggiornato nel seguente modo nel caso di una Store: wts := max(pts, rts+1). Infine, il Last Level Cache funge da Timestamp Manager, ovvero gestisce i timestamp durante gli accessi in memoria.

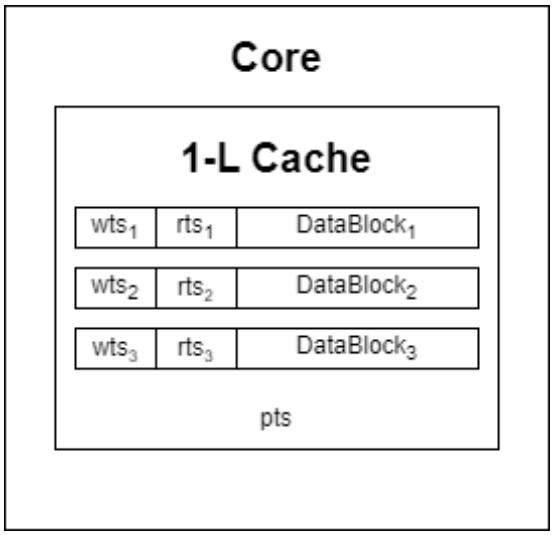


Figura 3.1: Timestamps in un Core con L1 Cache privata

3.1.2.3 Acquisizione Esclusiva dei Blocchi

Come i protocolli a directory, i blocchi possono essere modificati solo se sono acquisiti in maniera esclusiva da una cache privata, pertanto è necessario un quantità di bit pari a O(logn) per memorizzare l'owner del blocco. Se un altro core richiede la linea in lettura (sharer), anche l'owner diventerà uno sharer e verrà trasferita una copia del blocco al richiedente, mentre se un altro core la richiede in scrittura, il blocco verrà trasferito e aggiornata l'informazione di owner. Inoltre, è presente il meccanismo di Upgrade per modellare la transizione da un blocco Shared a Exclusive in una stessa L1 Cache.
Quando le cache richiedono un nuovo blocco al Timestamp Manager, esse allegano con la richiesta i loro timestamp che serviranno a quest'ultimo per stabilire se il blocco è aggiornato o meno. Questo è un vantaggio in quanto consente al Timestamp Manager di evitare transazioni dirette alla memoria, molto più lenta, qualora il blocco fosse già aggiornato e rispondere direttamente al richiedente con un messaggio leggero di acknowledgement, alleggerendo il carico sull'interconnessione.

3.1.3 Design del Protocollo Tardis

Per progettare il protocollo, è necessario definire il numero dei livelli di cache, e per ciascuno di essi implementare il relativo automa. Progetteremo il protocollo per un livello di cache, anche se è possibile un'estensione per due o tre livelli:

- L1: è il livello più vicino ai core elaborativi ed è privato per ciascuno di essi.
- DIR: sempre presente, memorizza lo stato e il proprietario di ogni linea di cache attualmente presenti nelle cache di primo livello.

Dunque, di questi due componenti, è necessario implementare i relativi automi, riferiti ovviamente a un generico blocco. In letteratura, questo protocollo è già stato descritto nelle seguenti forme tabellari, per la L1 Cache 3.2 e il Last Level Cache 3.3.

States	Core Event			Network Event		
	Load	Store	Eviction	SH_REP or EX_REP	RENEW_REP or UPGRADE_REP	FLUSH_REQ or WB_REQ
Invalid	send SH_REQ to TM **M.wts⇐0,** **M.pts⇐pts**	send EX_REQ to TM **M.wts⇐0**		Fill in data **SH_REP** **D.wts⇐M.wts** **D.rts⇐M.rts** state⇐Shared **EX_REP** **D.wts⇐M.wts** **D.rts⇐M.rts** state⇐Excl.	**RENEW_REP** **D.rts⇐M.rts** **UPGRADE_REP** **D.rts⇐M.rts** state⇐Excl.	
Shared *pts* ≤ *rts*	Hit **pts⇐Max(pts, D.wts)**	send EX_REQ to TM **M.wts⇐D.wts**	state⇐Invalid No msg sent.			
Shared *pts* > *rts*	send SH_REQ to TM **M.wts⇐D.wts,** **M.pts⇐pts**					
Exclusive	Hit **pts⇐Max(pts, D.wts)** **D.rts⇐Max(pts, D.rts)**	Hit **pts⇐Max(pts, D.rts+1)** **D.wts⇐pts** **D.rts⇐pts**	state⇐Invalid send FLUSH_REP to TM **M.wts⇐D.wts,** **M.rts⇐D.rts**			**FLUSH_REQ** **M.wts⇐D.wts** **M.rts⇐D.rts** send FLUSH_REP to TM state⇐Invalid **WB_REQ** **D.rts⇐Max(D.rts, D.wts+lease, reqM.rts)** **M.wts⇐D.wts** **M.rts⇐D.rts** send WB_REP to TM state⇐Shared

Figura 3.2: Automa L1 Cache

La macchina a stati finiti prevede che una linea di cache in una cache di 1° livello, possa trovarsi in uno dei seguenti stati:

- Invalid: la linea di cache non è presente in questa cache di 1° livello. Questo stato non viene effettivamente memorizzato con una linea di cache, in quanto se essa è invalida, allora non è neanche presente nella cache di 1° livello.
- Shared ($pts \leq rts$): la linea di cache è valida, e pertanto può essere letta, ed è condivisa nelle cache di 1° livello.
- Shared ($pts > rts$): la linea di cache è scaduta (rts rappresenta una scadenza) ed è necessario un rinnovo del lease time (rinnovo del rts).
- Exclusive: la linea di cache memorizzata è in modalità scrittura e lettura.

Vengono definiti i seguenti eventi:

- Load: evento proveniente dal core, tramite cui viene richiesta una linea di cache in sola lettura.

- Store: evento proveniente dal core, tramite cui viene richiesta una linea di cache in scrittura e lettura.
- Eviction: evento proveniente dal core, tramite cui viene richiesta la sostituzione di una linea di cache.
- SH_REQ e SH_REP: messaggi inviati verso l'interconnessione utilizzati per richiedere una linea di cache in sola lettura.
- EX_REQ e EX_REP: messaggi inviati verso l'interconnessione utilizzati per richiedere una linea di cache in scrittura e lettura.
- RENEW_REP: messaggio di risposta inviato dalla directory alla cache per rinnovare il time-stamp della linea di cache Shared, se non è stata modificata da un altro core. Questo messaggio non trasporta dati, pertanto è un messaggio leggero.
- UPGRADE_REP: messaggio di risposta inviato dalla directory alla cache per promuovere lo stato della cache da Shared a Exclusive, se non è stata modificata da un altro core. Questo messaggio non trasporta dati, pertanto è un messaggio leggero.
- FLUSH_REQ e FLUSH_REP: messaggi inviati verso l'interconnessione quando un blocco viene richiesto da una cache in modalità scrittura (Exclusive) ed è già posseduta da un'altra cache.
- WB_REQ e WB_REP: messaggi inviati verso l'interconnessione utilizzati per inoltrare una linea di cache (in stato Exclusive) a un'altra cache che la utilizzerà in lettura; alla fine, entrambe le cache possiederanno tale linea in Shared.

States	SH_REQ	EX_REQ	Eviction	DRAM_REP	FLUSH_REP or WB_REP
Invalid	Load from DRAM			Fill in data **D.wts⇐mts** **D.rts⇐mts** state⇐Shared	
Shared	**D.rts⇐Max(D.rts, D.wts+lease, reqM.pts+lease)** **if reqM.wts=D.wts** send RENEW_REP to requester **M.rts⇐D.rts** **else** send SH_REP to requester **M.wts⇐D.wts** **M.rts⇐D.rts**	**M.rts⇐D.rts** state⇐Excl. **if reqM.wts=D.wts** send UPGRADE_REP to requester **else** **M.wts⇐D.wts** send EX_REP to requester	**mts⇐Max(mts, D.rts)** Store data to DRAM if dirty state⇐Invalid		
Exclusive	send WB_REQ to the owner **M.rts⇐reqM.pts+lease**	send FLUSH_REQ to the owner			Fill in data **D.wts⇐M.wts,** **D.rts⇐M.rts** state⇐Shared

Figura 3.3: Automa Last Level Cache

La macchina a stati finiti del Last Level Cache prevede che una linea di cache possa trovarsi in uno dei seguenti stati:

- Invalid: la linea di cache non è presente in nessuna cache di 1° livello. Questo stato non viene effettivamente memorizzato con una linea di cache, in quanto se essa è invalida, allora non è neanche presente nella cache di 1° livello.

- Shared: il blocco è condiviso nelle cache di 1° livello.
- Exclusive: la linea di cache è in modalità scrittura e lettura in una ed una sola delle cache.

Tuttavia, in questo elaborato sarà la Directory ad assumere il ruolo del Timestamp Manager e non da un Last Level Cache. Le macchine a stati finiti rappresentate tramite grafi, risultanti dalle forme tabellari (senza includere le operazioni interne dei stati) sono presentate in Figura 3.4 e 3.5:

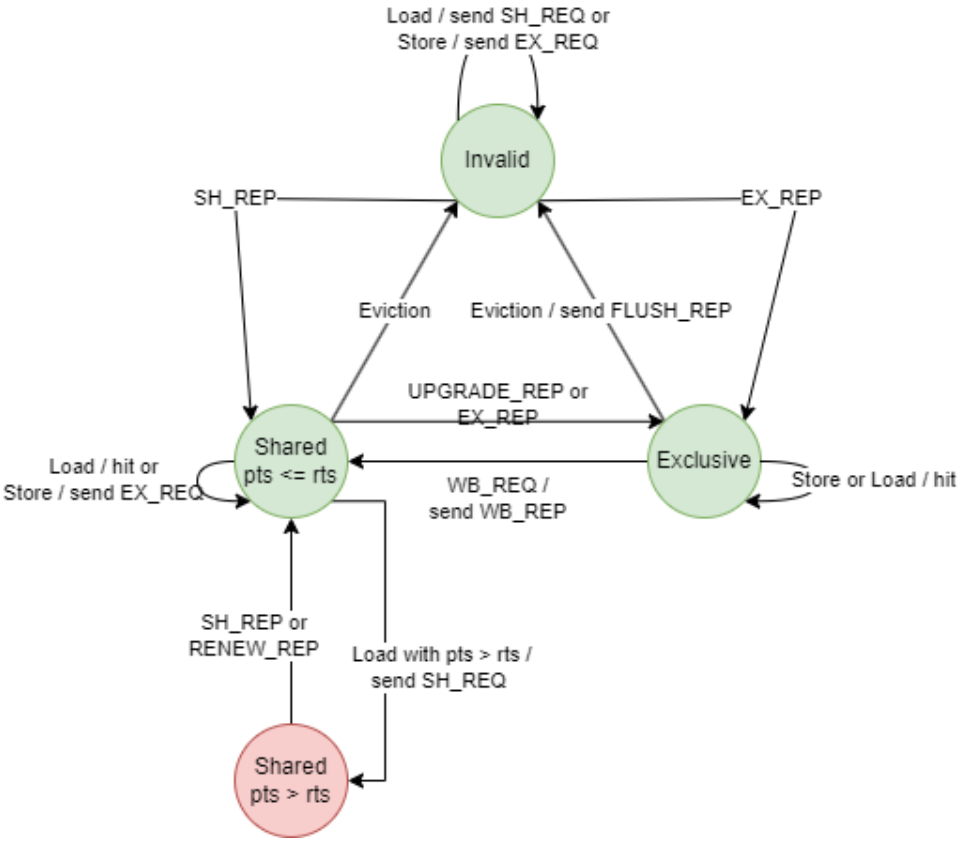


Figura 3.4: Grafo L1 Cache

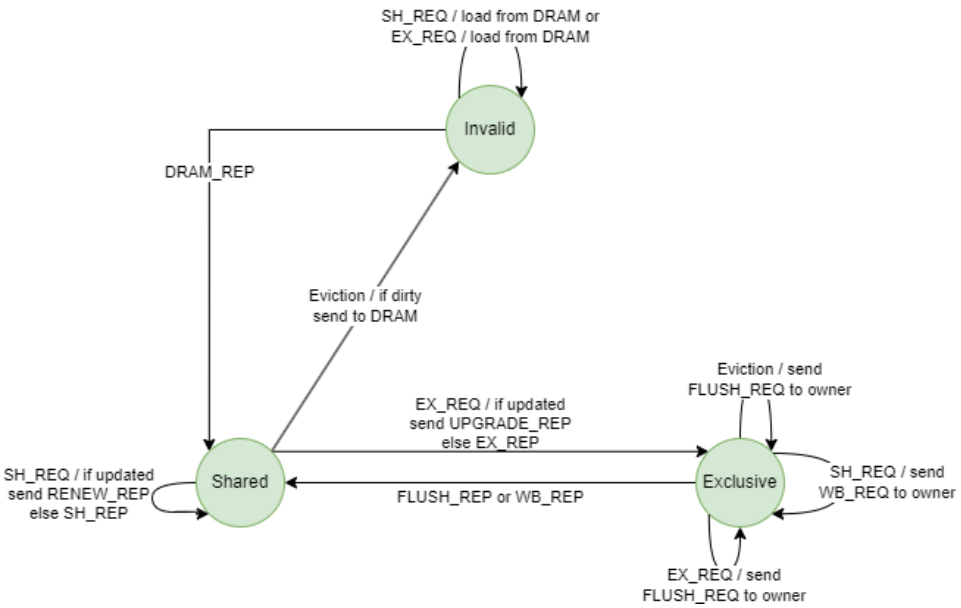


Figura 3.5: Grafo LLC

3.1.4 Problemi del protocollo

Tuttavia, il protocollo presentato ha gravi problemi di praticabilità. Prima di tutto, le attuali architetture di multiprocessori, non si basano sul modello di consistenza Sequential Consistency, ma su modelli più rilassati. Inoltre, il protocollo così com'è si basa sull'assunzione ideale che le transazioni siano atomiche, mentre in realtà sono di tipo pipelined o split-transaction. Infatti, è molto facile verificare che il protocollo va in crisi seguendo lo schema degli automi sopra presentati in caso di assunzioni di transazioni sovrapposte (più richieste per lo stesso indirizzo o indirizzi diversi), pertanto deve essere modificato per poter essere applicato a contesti reali.
Per risolvere il primo problema, è necessario assumere un modello di consistenza diverso da Sequential Consistency. Nella trattazione considereremo un modello di consistenza Total Store Order. Tuttavia, l'assunzione di un modello più rilassato, ha come effetto una diversa gestione del program timestamp.
Per il secondo problema, è necessario prevedere dei messaggi con una struttura più complessa, che trasportano anche le informazioni riguardante il destinatario del messaggio tramite un header. Infine, è necessaria l'estensione dell'automa con stati intermedi, in attesa di ricevere i messaggi di risposta, e nei quali ulteriori richieste vengono poste in attesa.
Un problema comune nei applicazioni multithread è rappresentato dal fenomeno del livelock in scenari che utilizzano variabili di sincronizzazione basate su spinlock. In tali casi, la versione cache della variabile di sincronizzazione, acceduta ripetutamente attraverso continue operazioni di load, potrebbe non venire mai invalidata. Questo accade perché, in assenza di un meccanismo di incremento automatico del timestamp (PTS), il programma cicla continuamente sulla stessa variabile senza causare modifiche allo stato della cache. Questa situazione è tipica dei contesti che utilizzano lo spinlock. Per risolvere il problema, si implementa un apposito algoritmo di Livelock Prevention, che introduce un meccanismo per gestire tali condizioni.

3.1.4.1 Modifiche L1 Cache

L'automa risultante per la cache di 1° livello è presentato in Figura 3.6. Dal momento che può esistere la corrispondenza tra un evento e più tipi di messaggi, la notazione utilizzata nel grafo per le transizioni è:
MESSAGE_ TYPE: *Event.*
In fase di implementazione bisogna definire esplicitamente tali corrispondenze. Inoltre, gli stati verdi rappresentano quelli stabili, gli stati grigi indicano gli stati intermedi, mentre il rosso segnala che il blocco è scaduto.

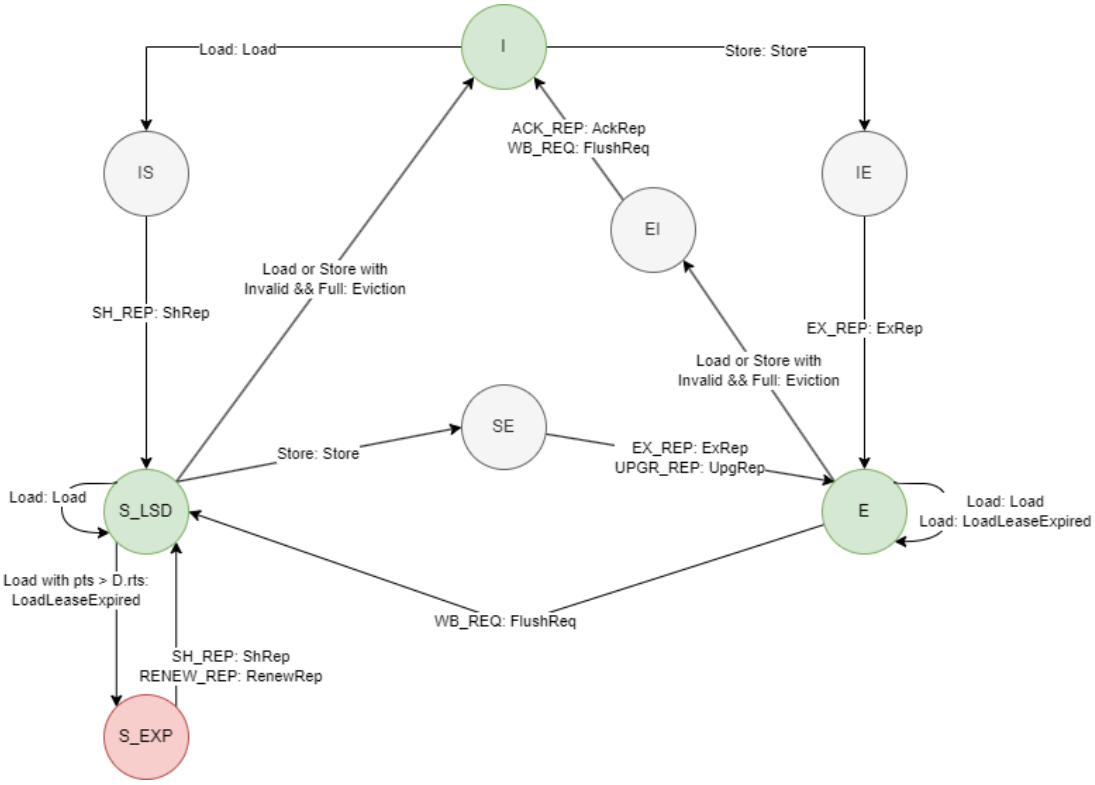


Figura 3.6: Automa L1 Cache Adattato per Transazioni Split-Transaction

Gli stati aggiuntivi della L1 Cache sono:

- IS: se viene effettuata una lettura (Load) su un blocco Invalid, si transita in questo stato, in cui La L1 Cache è in attesa di ricevere il blocco tramite un messaggio di Shared Response dalla Directory. Ulteriori Load, Store o Eviction sul blocco oggetto della transizione vengono poste in attesa.

- IE: se viene effettuata una scrittura (Store) su un blocco Invalid, si transita in questo stato, in cui L1 Cache è in attesa di ricevere il blocco tramite un messaggio di Exclusive Response dalla Directory. Ulteriori Load, Store o Eviction sul blocco oggetto della transizione vengono poste in attesa.

- EI: se viene sostituito (Eviction) un blocco in stato di Exclusive, si transita in questo caso, in cui la L1 Cache invia tale blocco (modificato) alla Directory ed attende un messaggio di acknoledgement (Ack) da quest'ultima a seguito del Writeback in memoria. Ulteriori Load, Store o Eviction sul blocco oggetto della transizione vengono poste in attesa.
  Caso particolare *WB_REQ: FlushReq* Può succedere che la L1 Cache effettua una Eviction ed è in attesa di ricevere l'Ack dalla Directory, ma quest'ultima sta gestendo una SH_REQ o EX_REQ avanzata da un'altra L1 Cache, inviando un messaggio di WB_REQ alla L1 Cache in attesa di ricevere l'Ack. Se l'evento fosse WbReq invece di FlushReq, la cache non saprebbe di aver ricevuto una richiesta di writeback dato che è in attesa dell'ack, causando un deadlock.

- SE: a seguito di un tentativo di scrittura (Store) su un blocco di sola lettura, viene inviato un messaggio di EX_REQ per richiedere tale privilegio alla Directory. In questo stato, la cache è in attesa di ricevere o un messaggio di UPGR_REP se il blocco non è stato modificato da

un'altra cache oppure di EX_REP se il blocco è stato nel frattempo modificato. Ulteriori Load, Store o Eviction sul blocco oggetto della transizione vengono poste in attesa.

Durante le transizioni verso gli stati stabili (verdi), la directory preleva da memoria il blocco richiesto e lo inoltra al richedente con i messaggi appositi.

- Transizione E → S_LSD: Nel momento in cui il proprietario del blocco riceve un messaggio di tipo WB_REQ, transita immediatamente allo stato Shared senza attraversare stati intermedi. Ciò avviene poiché la Directory, al momento dell'invio di tale messaggio, ha già provveduto a rimuovere lo stato di proprietario associato al blocco.

- Transizione S_LSD → S_EXP: Se il Load Timestamp risulta essere maggiore del Read Timestamp (lease time) del blocco oggetto della lettura, allora esso è scaduto, pertanto transiterà nello stato di Shared Expired. In questo stato, tutte le operazioni vengono poste in attesa.

- Transizione S_EXP → S_LSD: Alla ricezione di un messaggio di rinnovo del lease (RENEW_REP) o di un nuovo blocco aggiornato (SH_REP), il blocco torna ad essere valido. È importante notare che il messaggio RENEW_REP non trasporta il blocco, risultando quindi leggero, mentre il messaggio SH_REP include un blocco aggiornato, che è stato modificato da un'altra L1 Cache nel frattempo.

- Transizione E → E: Le operazioni di Load, pur incrementando i timestamp, non determinano mai la scadenza (tramite LoadLeaseExpired) di un blocco in stato Exclusive. Ciò avviene perché si assume che la L1 Cache che detiene il blocco nello stato Exclusive sia anche quella in possesso della versione più aggiornata del blocco

3.1.4.2 Modifiche Directory

L'automa illustrato in Figura 3.5 relativo al Timestamp Manager (Directory) viene esteso per gestire il tracciamento dei blocchi durante le transizioni intermedie. L'evento Memory_Data si riferisce alla ricezione, da parte della Directory, dei blocchi restituiti dalla memoria, mentre Memory_Ack e gli stati col pedice _m indicano che essa è in attesa di ricevere un messaggio di acknowledgement (Ack) da parte della memoria a seguito di un'operazione di writeback. Gli stati della Directory sono cache-centrici, quindi riflettono lo stato dei blocchi nelle L1 Cache.

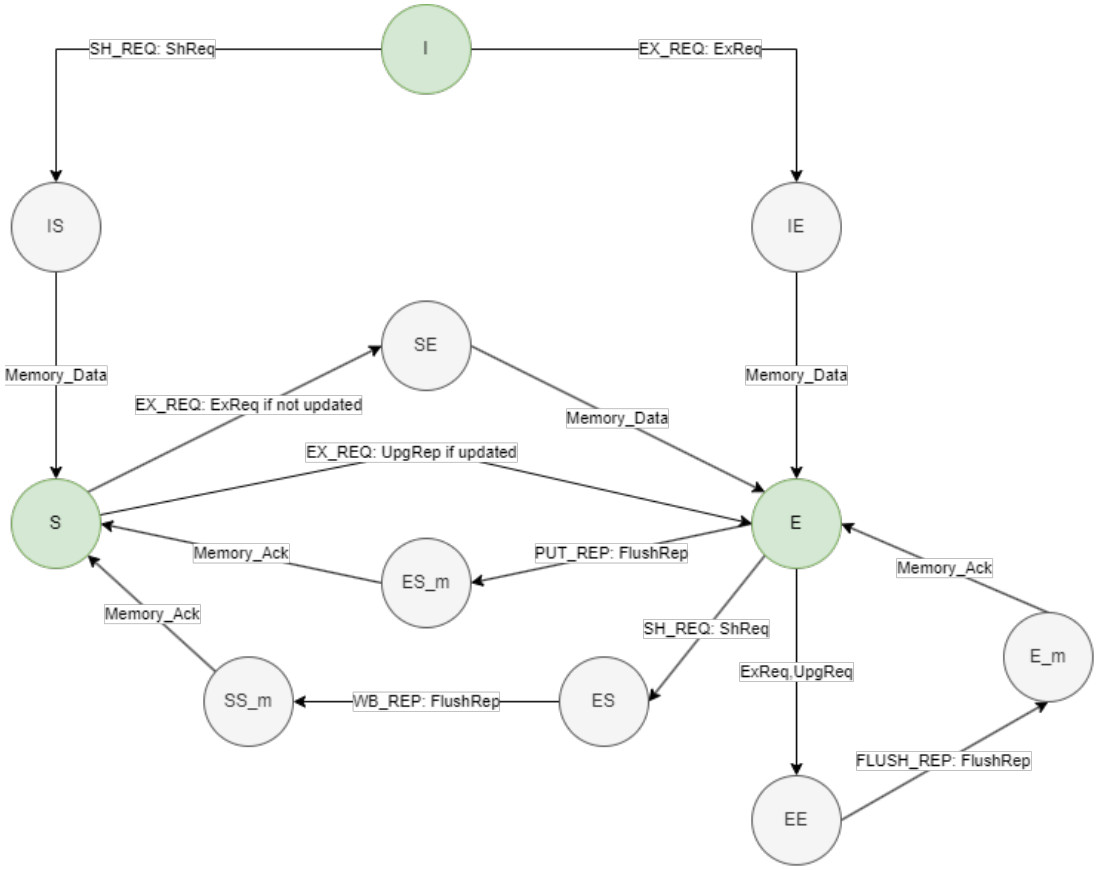


Figura 3.7: Automa Timestamp Manager (Directory) Adattato per Transazioni Pipelined o Split-Transaction

I nuovi stati intermedi del Timestamp Manager:

- IS e IE: a seguito della ricezione di un messaggio di SH_REQ o EX_REQ rispettivamente, si transita in uno di questi due stati, che modellano le transizioni I $\rightarrow$ S ed I $\rightarrow$ E. In entrambi gli stati, la Directory invia un messaggio di richiesta alla memoria per prelevare il blocco e, alla sua ricezione (Memory_Data), lo inoltra al richiedente. Ulteriori richieste di Load, Store e UpgReq vengono poste in attesa.

- SE: in seguito alla ricezione di un messaggio EX_REQ, qualora il blocco non risulti aggiornato (e.g. un'altra L1 Cache lo ha acquisito in stato Exclusive, lo ha modificato e successivamente rilasciato), la Directory deve recuperare il blocco aggiornato dalla memoria e inoltrarlo al richiedente.
  Invariante: Il blocco presente in memoria deve essere sempre la versione più aggiornata. Per

garantire tale condizione, è necessario eseguire un'operazione di Writeback verso la memoria ogni volta che la Directory riceve un blocco precedentemente in stato Exclusive.

- ES_m: se la Directory riceve una PUT_REP, vuol dire che l'attuale proprietario del blocco l'ha sostituito. Per mantenere l'invariante, la Directory deve inoltrarlo alla memoria con un'operazione di Writeback. In questo stato è in attesa di ricevere il Memory_Ack da essa.

- ES: stato che modella un contesto differente dal precedente. In questo caso, una L1 Cache ha richiesto un blocco Exclusive in lettura (SH_REQ), quindi la Directory inoltrerà un messaggio di WB_REQ all'attuale proprietario per acquisirlo. Alla ricezione del blocco (WB_REP), transita nello stato SS_m.

- SS_m: garantisce l'invariante, in particolare modella la procedura di Writeback che deve effettuare la Directory al momento dell'acquisizione del blocco aggiornato dal proprietario (tramite WB_REP). In questo stato, la Directory è in attesa di ricevere l'acknoledgement (Memory_Ack) dalla memoria.

- EE: stato intermedio che modella il cambio di Owner di un blocco. La directory invia un messaggio di FLUSH_REQ all'attuale proprietario e rimane in attesa di ricevere il blocco.

- E_m: una volta che la directory ha ricevuto il blocco, la invia alla memoria e attende l'acknoledgment (Memory_Ack) del Writeback da quest'ultima, e alla sua ricezione inoltra il blocco al nuovo proprietario.

Osserviamo che dal momento che una Directory non memorizza esplicitamente i blocchi, ma solo le loro informazioni, l'evento di Eviction non sarà considerato.

3.2 Protocollo Tardis 2.0 e il Modello Total Store Order

Un'altra versione del protocollo Tardis, chiamato Tardis 2.0, assume un modello di consistenza di memoria noto come Total Store Order. Tale modello, intuitivamente è più rilassato rispetto alla Sequential Consistency, inoltre è adottato anche dai processori moderni di architettura X86 e RISCV.

3.2.1 Total Store Order

Il modello Total Store Order, rilassa il Sequential Consistency eliminando una delle 4 condizioni della seguente **Regola 1**:

- $L(a) <_p L(b) \Rightarrow L(a) <_m L(b)$
- $L(a) <_p S(b) \Rightarrow L(a) <_m S(b)$
- $S(a) <_p S(b) \Rightarrow S(a) <_m S(b)$
- ~~$S(a) <_p L(b) \Rightarrow S(a) <_m L(b)$~~

E modifica la **Regola 2** della Sequential Consistency:
Regola 2-TSO: Valore di L(a) = Valore di $Max_{<m}\{S(a)|S(a) <_m L(a)$ or $S(a) <_p L(a)\}$
dove L(a) e S(a) sono operazioni di Load e Store rispettivamente all'indirizzo a.
Questo significa che le Load precedute da Store possono bypassare quest'ultime. Questo rilassamento permette l'introduzione di tecniche di ottimizzazione dei buffer delle store, facendo sì che se una store è presente nello store buffer, e quindi i suoi effetti non sono ancora avvenuti nella memoria globale, il suo valore può essere letto direttamente da un load successiva (secondo l'ordine del programma). La terza condizione invece asssicura un ordinamento FIFO dello store buffer, quindi l'ordine Store - Store è preservato. Tuttavia, sebbene per la maggior parte delle applicazioni questo rilassamento non comporta gravi problemi, può minare la semantica delle applicazioni parallele, risultando in comportamenti inattesi. Per ovviare a ciò, le architetture che supportano il modello Total Store Order, implementano delle istruzioni assembly dette barriere di memoria, che rafforzano il modello Sequential Consistency (ripristinando di fatto, l'ultima condizione) per una sezione di codice. Infatti, quando un programma incontra un'istruzione di barriera, chiamate **FENCE**, non procede finché tutte le operazioni di Load e Store non vengono completate. Introduce quindi, una nuova regola:
Regola 3: $X <_p FENCE \Rightarrow X <_m FENCE$
$FENCE <_p X \Rightarrow FENCE <_m X$
Dove X è una operazione di Load o Store. Il primo statement vincola tutte le operazioni di Load e Store precedenti a una FENCE secondo l'ordine del programma, di terminare prima della FENCE, secondo l'ordine globale della memoria. Mentre il secondo statement vincola tutte le operazioni di Load e Store, successive a una FENCE secondo l'ordine del programma, di essere eseguite *dopo* secondo l'ordine della memoria globale.

3.2.2 Tardis Total Store Order

Il protocollo Tardis Total Store Order si basa sempre sull'utilizzo dei timestamp, a meno di gestire accuratamente il caso dell'eliminazione della quarta condizione della Regola 1. Infatti, secondo tale regola, vengono mantenute le precedenze L $\rightarrow$ L, S $\rightarrow$ S e L $\rightarrow$ S mentre S $\rightarrow$ L non vale più, aprendo la strada a possibili ottimizzazioni hardware per l'accesso alla memoria, in particolare, con il riordino delle operazioni di Load successive a una Store e l'utilizzo di Store buffer. Da questo si evince che un singolo timestamp, il pts, non è sufficiente per modellare le operazioni di riordino di Load e Store, pertanto, esso viene suddiviso in ulteriori due timestamp. Dal momento che le Load e le Store possono avvenire in ordini differenti rispetto a quanto specificato dal programma, tale suddivisione prevede:

- Store Timestamp (sts): timestamp (logico) del commit dell'ultima Store.
- Load Timestamp (lts): timestamp (logico) del commit dell'ultima Load.

Il Read Timestamp e il Write Timestamp preservano la loro funzione di lease time e timestamp dell'ultima Store in uno specifico blocco rispettivamente. In accordo alla Regola 2, dal momento che sono preservate le precedenze L $\rightarrow$ L e S $\rightarrow$ S, i due timestamp lts ed sts, sono monotoni crescenti. Inoltre, in accordo alla condizione 2 della Regola 1, il timestamp di una store (sts) non deve essere mai

inferiore al timestamp della Load precedente (lts); mentre in accordo all'eliminazione della condizione 4 dalla Regola 1, il timestamp di una Load (lts) può essere inferiore al timestamp sts, a causa della mancanza del vincolo S $\rightarrow$ L.
Tuttavia, questo apre una problematica di tipo architetturale per le istruzioni di FENCE, in particolare, è necessario forzare in qualche modo la condizione 4 della Regola 1 qualora venisse eseguita un'istruzione di barriera. Questo lo si può ottenere tramite la sincronizzazione dei timestamp quando viene eseguita una di tali istruzioni: lts = Max(lts, sts). Questa rinforza la condizione eliminata, rendendo Tardis temporaneamente SC quando viene eseguita un'istruzione di FENCE. Dal punto di vista del processore però, è necessario inviare l'informazione di FENCE anche alla L1 Cache, che altrimenti, non riuscirebbe a sincronizzare i timestamp e forzare il modello SC. Questo può essere fatto in sede di accesso alla Cache da parte del processore allegando l'informazione riguardo l'esecuzione di un'istruzione di FENCE.
Infine, un vantaggio di utilizzare Tardis TSO è che l'utilizzo di due timestamp rende il loro incremento più lento, riducendo il numero di mesaggi di Renew sulle interconnessioni.

3.2.3 Livelock Prevention

Poiché i PTS (per Tardis) e LTS (per Tardis 2.0) vengono incrementati restituendo il valore più recente della store, è evidente che, se un'applicazione parallela si sincronizza su una variabile di spinlock (tramite operazioni di Load continue), i timestamp non aumentano mai. Questo impedisce il rinnovo della linea di cache. Con altri protocolli, tale situazione sarebbe gestita tramite un messaggio di Invalidate, inviato quando una variabile di spinlock viene modificata con un'operazione di Store eseguita da un altro core. Nel caso di Tardis, invece, la cache che effettua le operazioni di Load sulla variabile di spinlock non rileverà mai le eventuali modifiche apportate alla stessa variabile da un'altra cache che esegue operazioni di Store. Per risolvere il problema del livelock, può essere adottato un algoritmo descritto in letteratura che si basa sull'introduzione di un valore di soglia e di un contatore per il numero di operazioni di load effettuate su ciascun blocco di cache (preferibilmente nella Data Cache). A ogni operazione di load, il contatore viene incrementato; al raggiungimento della soglia prestabilita, quest'ultima viene dimezzata (per semplificare la divisione tramite hardware), il contatore viene azzerato e il valore del lts incrementato di 1. Con il passare del tempo, il valore del lts cresce fino a superare il rts (lease time) associato al blocco, provocando così un aggiornamento forzato del blocco, presumibilmente con la variabile di spinlock aggiornata.

Tardis TSO - Livelock Detection Algorithm in C language

```
cache_entry{
    load_counter = 0; //Init to 0
    livelock_period = 32; //Multiple of 2. 64,128...
};

LivelockAlgorithm(Entry cache_entry){
    cache_entry.load_counter = cache_entry.load_counter+1;
    if((cache_entry.load_counter)%cache_entry.livelock_period ==
        0){
        lts = lts + 1;
        cache_entry.load_counter = 0;
        cache_entry.livelock_period = cache_entry.
           livelock_period/2;
        if(cache_entry.livelock_period == 0){
            cache_entry.livelock_period = 1;
        }
    }
}
```

Capitolo 4

Implementazione

Il protocollo implementato è Tardis 2.0 con modello di consistenza Total Store Order. A tale scopo è necessario implementare gli automi che sono stati definiti durante la fase di progettazione. L'implementazione, articolata secondo la Figura 4.1, sarà effettuata mediante il linguaggio SLICC, Specification Language for Implementing Cache Coherence, il quale è un linguaggio C-like specifico del dominio di gem5, utilizzato per specificare i protocolli di coerenza della cache. SLICC viene utilizzato per specificare il comportamento della macchina a stati finiti, e poiché lo scopo è modellare l'hardware il più fedelmente possibile, SLICC impone vincoli che possono essere specificati sulle macchine a stati. In questo capitolo vedremo anche molti paradigmi e sintassi del linguaggio. L'implementazione avverrà nella directory seguente: *gem5/src/learning_gem5/tardis_tso/*

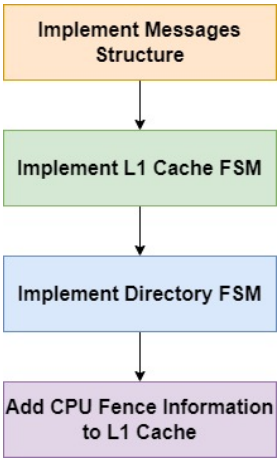


Figura 4.1: Fasi dell'implementazione

4.1 Messaggi

L'implementazione dei messaggi, seguirà i seguenti step:

Figura 4.2: Fasi dell'implementazione dei messaggi

È necessario creare il file in: *gem5/src/learning_gem5/tardis_tso/TARDISTSO-msg.sm*

4.1.1 Messaggi di Richiesta

I messaggi di richiesta, definiti in fase di progettazione, risultano essere i seguenti: SH_REQ, EX_-REQ, FLUSH_REQ e WB_REQ. I primi due sono inviati dalle cache per richiedere blocchi in lettura o scrittura, mentre gli ultimi dalla Directory per informare una cache proprietaria del blocco, di effettuare l'eviction per soddisfare la richiesta di un'altra cache. La keyword enumeration permette di creare delle enumerazioni di elementi, che in questo caso, rappresentano i messaggi di richiesta appena citati, con una breve descrizione. CoherenceRequestType è un tipo built-in di gem5, e serve a specificare una categoria di messaggio.

Enum Request Message Type

```
enumeration(CoherenceRequestType, desc="Types of request messages")
{
  SH_REQ,      desc="Request sent from cache for a block with read
     permission";
  EX_REQ,      desc="Request sent from cache for a block with write
     permission";
  FLUSH_REQ,   desc="Request sent from directory to cache to evict a
     block (due Exclusive Req)";
  WB_REQ,      desc="Request sent from directory to cache to evict a
     block (due Writeback)";
}
```

La struttura del messaggio prevede dei campi che identificano l'indirizzo del blocco richiesto (*Addr addr*), il tipo di richiesta che abbiamo definito tramite l'enumerazione (*CohrenceRequestType Type*), l'identificativo del core che invia il messaggio di richiesta (*MachineID Requestor*), l'identificativo del nodo di destinazione (*NetDest Destination*, è una maschera, supportando comunicazioni multicast) e la grandezza del messaggio (*MessageSizeType MessageSize*). Il campo Datablock è uno stub, necessario per implementare le due funzioni obbligatorie: funcionalRead() e functionalWrite(), le quali servono ad implementare la lettura o la scrittura sul datablock del messaggio di richiesta contenuto nel pacchetto. La loro implementazione in questo caso è semplice in quanto questi tipi di messaggi non trasportano dati. RequestMsg è il nome della struttura mentre Message è un'interfaccia definita in gem5.
Al messaggio vanno aggiunti anche i timestamp, che serviranno alla Directory (che funge da Timestamp Manager) per stabilire se il blocco posseduto da una cache è aggiornato (e.g. Una cache possiede già il blocco in SHARED, e invia una richiesta EX_REQ per poter effettuare una Store, ed i timestamp contenuti serviranno alla Directory per verificare se il blocco in SHARED è già aggiornato).

Request Message Structure

```
structure(RequestMsg, desc="Used for request and forward-request
    message", interface="Message") {
  Addr addr,                    desc="Physical address for this
     request";
  CoherenceRequestType Type,    desc="Type of request";
  MachineID Requestor,          desc="Node who initiated the request";
  NetDest Destination,          desc="Used for multicast destination
     mask";
  DataBlock DataBlk,            desc="Datablock";
  MessageSizeType MessageSize, desc="Size category of the message";

  int lts,                   desc="Load timestamp sent into request";
  int sts,                   desc="Store timestamp sent into request";
  int rts,                   desc="Read timestamp sent into request";
  int wts,                   desc="Write timestamp sent into request";

  bool functionalRead(Packet *pkt) {
    return false;
  }

  bool functionalWrite(Packet *pkt) {
    return testAndWrite(addr, DataBlk, pkt);
  }
}
```

4.1.2 Messaggi di Risposta

Seguendo lo stesso principio, definiamo l'enumerazione dei messaggi di risposta come segue:

Enum Response Message Type

```
enumeration(CoherenceResponseType, desc="Types of response messages")
    {
  SH_REP, desc="Shared Response sent from directory to cache";
  EX_REP, desc="Exclusive Response sent from directory to cache";
  RENEW_REP, desc="Renew for lease sent from directory to cache";
  UPGR_REP, desc="Upgrade Response sent from directory to cache (due
     to owned EX_REQ)";
  FLUSH_REP, desc="Flush Response sent from cache to directory";
  WB_REP, desc="Writeback Response sent from cache to directory";
  PUT_REP, desc="Put Response sent from cache to directory due to
     eviction";
  ACK_REP, desc="Ack from Memory Response sent from directory to
     cache due to finished Writeback of data in Memory";
}
```

La struttura del messaggio prevede dei campi che identificano l'indirizzo richiesto, il tipo di risposta che abbiamo definito tramite l'enumerazione, l'identificativo del core che invia il messaggio di risposta (indicato con Sender), l'identificativo del nodo di destinazione (è una maschera, quindi anche un multicast) e la grandezza del messaggio. In questo caso, il campo Datablock trasporta il vero blocco, e l'implementazione delle due funzioni prevede di effettuare un controllo sul tipo di messaggio di risposta. La functionalWrite() resta inviariata, in quanto il messaggio non va modificato, mentre la functionalRead() per poter leggere il datablock deve prima verificare se il messaggio di risposta corrisponde a uno di quelli enumerati che trasporta dati, cioé tutti tranne l'Ack Response. ResponseMsg è il nome della struttura mentre Message è un'interfaccia definita in gem5.
Alla struttura del messaggio di risposta vanno aggiunti anche i timestamp, che verranno utilizzati dalle cache e dalla Directory durante lo scambio dei datablock, per aggiornare il lease, verificare se il blocco è aggiornato ed altre operazioni.

Response Message Structure

```
structure(ResponseMsg, desc="Response and forward-response message",
    interface="Message") {
  Addr addr,                    desc="Physical address for this
     response";
  CoherenceResponseType Type,   desc="Type of response";
  MachineID Sender,             desc="Node who is responding to the
     request";
  NetDest Destination,          desc="Used for multicast destination
     mask. In Tardis TSO will be always 1-destination value (no
     multicast)";
  DataBlock DataBlk,            desc="Datablock";
  MessageSizeType MessageSize, desc="Size category of the message";

  int lts,                  desc="Load timestamp sent into response";
  int sts,                  desc="Store timestamp sent into response";
  int rts,                  desc="Read timestamp sent into response";
  int wts,                  desc="Write timestamp sent into response";

  bool functionalRead(Packet *pkt) {
    if (Type == CoherenceResponseType:SH_REP || Type ==
       CoherenceResponseType:EX_REP
        || Type == CoherenceResponseType:RENEW_REP || Type ==
           CoherenceResponseType:FLUSH_REP
        || Type == CoherenceResponseType:WB_REP || Type ==
           CoherenceResponseType:UPGR_REP
        || Type == CoherenceResponseType:PUT_REP) {
      return testAndRead(addr, DataBlk, pkt);
    }
    return false;
  }

  bool functionalWrite(Packet *pkt) {
    return testAndWrite(addr, DataBlk, pkt);
  }
}
```

4.1.3 Messaggi del DMA Stub

Per utilizzare la Full System Emulation, è necessario implementare anche l'automa del DMA. In questo caso utilizzeremo un DMA di stub con i propri messaggi, che non dipendono dal tipo di implementazione del protocollo. Tali messaggi del DMA sono gli stessi del sorgente di un altro

protocollo già implementato, in particolare da *gem5/src/mem/ruby/protocol/MI_ example-msg.sm*

4.2 CPU

In fase di progettazione, abbiamo visto che Tardis in Total Store Order prevede il rilassamento di una delle condizioni di precedenza di accesso alla memoria. Questa può essere rinforzata rendendo il modello di consistenza temporaneamente Sequential Consistency tramite le istruzioni di FENCE. Pertanto, è necessario implementare la logica per rendere noto alla L1 Cache che una barriera di memoria è stata eseguita e dunque che deve sincronizzare i timestamp lts ed sts. Le modifiche differiscono a seconda del tipo di CPU che si vuole adattare al protocollo, e tali modifiche verranno annotate con *//TARDIS TSO*.

4.2.1 TimingSimpleCPU

Per il TimingSimple CPU è necessario aggiungere un flag chiamato memoryBarrierFetched tra le variabili membro pubbliche, nel sorgente *gem5/src/cpu/simple/timing.hh*. Tale flag serve a notificare alla cache se è stata eseguita una barriera di memoria. Il flag viene impostato dalla CPU quando viene eseguita un'istruzione di barriera e la cache verificherà il valore del flag.

Timing Simple CPU modifications

```
namespace gem5
{

class TimingSimpleCPU : public BaseSimpleCPU
{
  public:
    bool memoryBarrierFetched; //TARDIS TSO

    TimingSimpleCPU(const BaseTimingSimpleCPUParams &params);
    virtual ~TimingSimpleCPU();

    void init() override;

  private:
  //....
```

Per impostare al valore *true* il flag, viene utilizzata la funzione *curStaticInst->isFullMemBarrier()*, già implementata in gem5 che restituisce *true* se l'istruzione attuale è un'istruzione di barriera. Questa operazione verrà effettuata nella funzione di completeIfetch() del modulo *gem5/src/cpu/simple/timing.cc* come di seguito.

Timing Simple CPU modifications

```
void
TimingSimpleCPU::completeIfetch(PacketPtr pkt)
{
    //...

    // hardware transactional memory
    if (curStaticInst && curStaticInst->isHtmStart()) {
        // if this HtmStart is not within a transaction,
        // then assign it a new htmTransactionUid
        if (!t_info.inHtmTransactionalState())
            t_info.newHtmTransactionUid();
        SimpleThread* thread = t_info.thread;
        thread->htmTransactionStarts++;
        DPRINTF(HtmCpu, "htmTransactionStarts++=%u\n",
            thread->htmTransactionStarts);
    }

    //Tardis TSO
    if (curStaticInst && curStaticInst->isFullMemBarrier()) {
        memoryBarrierFetched = true;
    }

    if (curStaticInst && curStaticInst->isMemRef()) {
                DPRINTF(SimpleCPU, "Execute inititateAcc()\n");
            //...
}
```

Il passo successivo riguarda l'aggiunta della logica per allegare il valore del flag con gli accessi Load o Store. Tali accessi sono definiti in 3 funzioni principali implementate nel medesimo sorgente.

- Fault TimingSimpleCPU::initiateMemRead()
- Fault TimingSimpleCPU::writeMem()
- Fault TimingSimpleCPU::initiateMemAMO()

Per fare questo, è necessario aggiungere un nuovo flag, *isMemBarrier*, nei messaggi di di richiesta *req*. L'aggiunta di questo nuovo flag ai messaggi verrà mostrata alla fine di questa sezione. Per la initiateMemRead():

Timing Simple CPU modifications

```
Fault
TimingSimpleCPU::initiateMemRead(Addr addr, unsigned size,
                                 Request::Flags flags,
                                 const std::vector<bool>&
                                    byte_enable)
{
    //...
    req->taskId(taskId());

    ///Tardis TSO
    if(memoryBarrierFetched){
        req->isMemBarrier = true;
        memoryBarrierFetched = false;
    }else{
        req->isMemBarrier = false;
    }

    Addr split_addr = roundDown(addr + size - 1, block_size);
    assert(split_addr <= addr || split_addr - addr < block_size)
       ;
    //...
}
```

Per la writeMem():

Timing Simple CPU modifications

```
Fault TimingSimpleCPU::writeMem(uint8_t *data, unsigned size,
                            Addr addr, Request::Flags flags,
                               uint64_t *res,
                            const std::vector<bool>& byte_enable)
{
    //...
    RequestPtr req = std::make_shared<Request>(
        addr, size, flags, dataRequestorId(), pc, thread->
           contextId());
    req->setByteEnable(byte_enable);

    req->taskId(taskId());

    ///Tardis TSO
    if(memoryBarrierFetched){
        req->isMemBarrier = true;
        memoryBarrierFetched = false;
    }else{
        req->isMemBarrier = false;
    }

    Addr split_addr = roundDown(addr + size - 1, block_size);
    assert(split_addr <= addr || split_addr - addr < block_size)
       ;
    //...
}
```

Per la initiateMemAMO():

Timing Simple CPU modifications

```
Fault
TimingSimpleCPU::initiateMemAMO(Addr addr, unsigned size,
                                Request::Flags flags,
                                AtomicOpFunctorPtr amo_op)
{
    //...
    RequestPtr req = std::make_shared<Request>(addr, size, flags
       ,
                           dataRequestorId(), pc, thread->
                              contextId(),
                           std::move(amo_op));
    assert(req->hasAtomicOpFunctor());

    req->taskId(taskId());

    ///Tardis TSO
    if(memoryBarrierFetched){
        req->isMemBarrier = true;
        memoryBarrierFetched = false;
    }else{
        req->isMemBarrier = false;
    }

    Addr split_addr = roundDown(addr + size - 1, block_size);
    //...
}
```

4.2.2 O3CPU

Dal momento che una CPU Out of Order rispecchia le CPU reali e tutti i loro componenti interni (Load and Store Queue, Prefetcher, Brand predictors, Scoreboard, ..), a differenza della TimingSimple è stata progettata su più moduli, pertanto le modifiche da apportare sono notevoli. Tutti i suoi sorgenti sono presenti in *gem5/src/cpu/o3*.
Per determinare in quale punto del codice inserire e gestire tali flag, bisogna analizzare il flusso di una istruzione di Load (o Store) di una O3CPU.

Call flow

```
IEW::tick()->IEW::executeInsts()
  ->LSQUnit::executeLoad()
    ->StaticInst::initiateAcc()
      ->LSQ::pushRequest()
        ->LSQUnit::read()
          ->LSQRequest::buildPackets()
          ->LSQRequest::sendPacketToCache()
    ->LSQUnit::checkViolation()
DcachePort::recvTimingResp()->LSQRequest::recvTimingResp()
  ->LSQUnit::completeDataAccess()
    ->LSQUnit::writeback()
      ->StaticInst::completeAcc()
      ->IEW::instToCommit()
IEW::tick()->IEW::writebackInsts()
```

È facile verificare che tutto parte nel contesto dell'IEW (Issue, Execute, Writeback). Il primo passo quindi prevede di modificare il sorgente *iew.hh* per aggiugnere il flag di barriera, e procedere all'inizializzazione del valore nel costruttore implementato in *iew.cc*.

iew.hh

```
class IEW
{
    //...
  public:

    //Tardis TSO
    bool memoryBarrierExecuted;

    /** Instruction queue. */
    InstructionQueue instQueue;
    //...
```

E inizializziamo al valore *false* tale flag.

iew.cc

```
namespace gem5
{

namespace o3
{

IEW::IEW(CPU *_cpu, const BaseO3CPUParams &params)
    //...
{
    //Tardis TSO
    memoryBarrierExecuted = false;

    //...
```

La difficoltà nell'impostare tali flag, risiede nel fatto che le istruzioni di fence non sono considerate accessi alla memoria. Di conseguenza, durante la loro esecuzione, non è intrinsecamente possibile notificare alla cache L1 l'avvenuta esecuzione della barriera. Tuttavia, è possibile aggirare questo limite utilizzando un flag impostato durante l'esecuzione della fence e trasferendo tale valore nella prima istruzione di accesso alla memoria successiva. La cache L1 può quindi verificare il flag durante la ricezione di ogni operazione di load o store.
Per implementare questa funzionalità nella CPU O3, è necessario configurare il valore del flag all'interno della funzione executeInsts(). Grazie al metodo isFullMemBarrier fornito da gem5 nella classe Instr, è possibile identificare se è stata eseguita un'istruzione di fence. Nel seguente sorgente è possibile identificare due blocchi di codice inerenti a TARDIS TSO: il primo verifica se è stata eseguita un'istruzione di barriera chiamando isFullMemBarrier(), mentre il secondo blocco di codice viene eseguito quando l'istruzione eseguita è una Load o una Store (essendo nel branch inst->isMemRef()), ed imposta al valore *true* un secondo flag contenuto nelle istruzioni inst, chiamato *memBarrierExecutedBefore*. Questo secondo flag sarà necessario per la Load and Store Queue. Una volta aver impostato il secondo flag, è possibile resettare il suo valore.

iew.cc

```
void IEW::executeInsts()
{
    //...

        // Notify potential listeners that this instruction has
           started
        // executing
        ppExecute->notify(inst);

        //Tardis TSO
        if(inst->isFullMemBarrier()){
                memoryBarrierExecuted = true;
        }
        //...
        Fault fault = NoFault;

        // Execute instruction.
        // Note that if the instruction faults, it will be
           handled
        // at the commit stage.
        if (inst->isMemRef()) {
            DPRINTF(IEW, "Execute: Calculating address for
               memory "
                    "reference.\n");

                ///Tardis TSO
                if(memoryBarrierExecuted == true){
                        memoryBarrierExecuted = false;
                        inst->memBarrierExecutedBefore = true;
                }else{
                        inst->memBarrierExecutedBefore = false;
                }
            // Tell the LDSTQ to execute this instruction (if it
                is a load).
            if (inst->isAtomic()) {
```

Il passo successivo è di inserire nella classe DynInst il secondo flag utilizzato nel codice precedente, memBarrierExecutedBefore. Sarà anch'esso un booleano e servirà ad impostare a sua volta un terzo flag, isMemBarrier nei messaggi di richiesta.

dyn_inst.hh

```
class Packet;

namespace o3
{

class DynInst : public ExecContext, public RefCounted
{
  //...
  public:

    //Tardis TSO
    bool memBarrierExecutedBefore;

    size_t numSrcs() const { return _numSrcs; }
    //...
```

L'ultimo passaggio prevede l'inserimento del flag isMemBarrier per i messaggi di richiesta della O3 CPU e la logica per impostare tale flag in base a precedente esecuzione di una fence. La struttura dei messaggi di richiesta sono definiti insieme alla classe LSQ (Load and Store Queue). Si noti che il passaggio tra un flag e il successivo (come una pipeline) è necessario in quanto la Load and Store Queue non ha alcun modo di sapere se è stata eseguita una istruzione di fence in precedenza.

lsq.hh

```
namespace o3
{

class CPU;
class IEW;
class LSQUnit;

class LSQ
{
    //...
        bool isDelayed() { return flags.isSet(Flag::Delayed); }
    class LSQRequest : public BaseMMU::Translation, public
       Packet::SenderState
    {
      protected:
        typedef uint32_t FlagsStorage;
        typedef Flags<FlagsStorage> FlagsType;
        //...
        void markDelayed() override { flags.set(Flag::Delayed);
           }
        bool isDelayed() { return flags.isSet(Flag::Delayed); }

      public:
        //Tardis TSO
        bool isMemBarrier;

        LSQUnit& _port;
```

Dunque dalla figura del flusso iniziale, riusciamo a determinare che:

1. I pacchetti vengono costruiti in LSQRequest, nel contesto della LSQ.
2. La prima funzione chiamata dalla LSQ è la pushRequest()

Andando ad analizzare la pushRequest, è possibile vedere che l'ultima funzione chiamata è la initiateTranslation().

lsq.cc

```
Fault LSQ::pushRequest(const DynInstPtr& inst, bool isLoad,
   uint8_t *data,
         unsigned int size, Addr addr, Request::Flags flags,
            uint64_t *res,
         AtomicOpFunctorPtr amo_op, const std::vector<bool>&
            byte_enable)
{
         //...
         request->initiateTranslation();
}
```

La *request -> initiateTranslation()* metodo è reimplementato in 3 modi differenti, a seconda del tipo di accesso da parte della Load and Store Queue: SplitDataRequest, UnsquashableDirectRequest e SingleDataRequest. Analizzando anche il loro corpo, è possibile notare che viene chiamata la funzione addReq() che di fatto, popola e crea una richiesta da inviare alla memoria. Pertanto il flag dovrà essere inserito nel corpo di tale funzione. Infatti, è facile verificare che tale metodo viene chiamato in tutte le 3 reimplementazioni della initiateTranslation() (SplitDataRequest, UnsquashableDirectRequest e SingleDataRequest).

lsq.cc

```
void LSQ::SingleDataRequest::initiateTranslation()
{
    assert(_reqs.size() == 0);

    addReq(_addr, _size, _byteEnable);
    //...
```

Grazie al riferimento della variabile protected _inst, è possibile ricavare il valore del flag come mostrato nel codice seguente.

lsq.cc

```
void LSQ::LSQRequest::addReq(Addr addr, unsigned size,
           const std::vector<bool>& byte_enable)
{
    if (isAnyActiveElement(byte_enable.begin(), byte_enable.end
       ())) {
        auto req = std::make_shared<Request>(
                addr, size, _flags, _inst->requestorId(),
                _inst->pcState().instAddr(), _inst->contextId(),
                std::move(_amo_op));
        req->setByteEnable(byte_enable);

        //Tardis TSO
        req->isMemBarrier = _inst->memBarrierExecutedBefore;

        /* If the request is marked as NO_ACCESS, setup a local
           access */
        if (_flags.isSet(Request::NO_ACCESS)) {
        //....
```

Il seguente log è inerente a un'esecuzione di un workload che fa uso delle barriere di memoria (mfence), nel caso di O3CPU, ed è possibile notare che dopo l'esecuzione di una mfence nel contesto del processore (cpu0), viene rilevata la barriera e sincronizzati i timestamp nel contesto della L1 Cache (l1_cntrl0).

Log

```
17101000: system.cpu0: T0 : 0x422ddd @tcache\_init.part.0+29. 0
   :       CMPXCHG\_LOCKED\_M\_R.mfence : IntAlu :
17131000: system.ruby.l1\_cntrl0: TARDISTSO-cache.sm:289: [
   MEMBARRIER] Memory barrier executed. Synchronizing timestampss
```

4.2.3 Messaggi Ruby dal Core

I messaggi di richiesta di accesso nativi di gem5, provenienti dal core, devono anch'essi essere modificati per contenere il valore del flag che identifica l'esecuzione di una barriera di memoria. Il flag *isMemBarrier* (diverso da quello definito nella Load and Store Queue) va inserito nella classe Request nel sorgente *gem5/src/mem/request.hh*. Tali messaggi rappresentano le richieste di accesso da parte del processore verso la L1 Cache.

Flag in Request Class

```
class Request : public Extensible<Request>
{
  public:
        /// Tardis TSO
        bool isMemBarrier;

        typedef uint64_t FlagsType;
        typedef uint8_t ArchFlagsType;
        //...
```

Tali messaggi però, non possono essere direttamente gestiti dal sottosistema Ruby, e dunque hanno la necessità di essere convertiti. Questo è un passaggio necessario per rendere possibile il trasferimento delle richieste di accesso da parte del core verso Ruby che permette di implementare protocolli di coerenza personalizzati. Per fare questo, Ruby utilizza un altro tipo di messaggio, chiamato *RubyRequesto*. Pertanto anche questa classe, localizzata in *gem5/sc/mem/ruby/slicc_ interface/RubyRequest.hh* ha bisogno di tale flag. Viene aggiunto il flag isMemBarrier ed una funzione (InsertTardisTSOInfo()) che copia il valore del flag passato dal processore riguardo l'avvenuta esecuzione di una barriera di memoria.

Flag in RubyRequest Class

```
class RubyRequest : public Message
{
  public:
    Addr m_PhysicalAddress;
    //...
    bool m_isSLCSet;

    //TARDIS TSO
    bool m_isMemBarrier;
    void InsertTardisTSOInfo(PacketPtr _pkt){
        m_isMemBarrier = _pkt->req->isMemBarrier;
    }

    //...
}
```

Infine, la funzione InsertTardisTSOInfo() va inserita in tutti i costruttori presenti nel suddetto sorgente, nel seguente modo.

Flag in RubyRequest Class

```
RubyRequest(Tick curTime, uint64_t _paddr, int _len,
        uint64_t _pc, RubyRequestType _type, RubyAccessMode
           _access_mode,
        //...
          m_isTlbi(false),
          m_tlbiTransactionUid(0)
    {
        InsertTardisTSOInfo(_pkt);      //TARDIS TSO

        m_LineAddress = makeLineAddress(m_PhysicalAddress);
        //...
```

4.3 L1 Cache

L'implementazione della macchina a stati finiti della cache, le cui fasi sono descritte in Figura 4.3 e la struttura del file subito dopo, risulta essere più complessa.

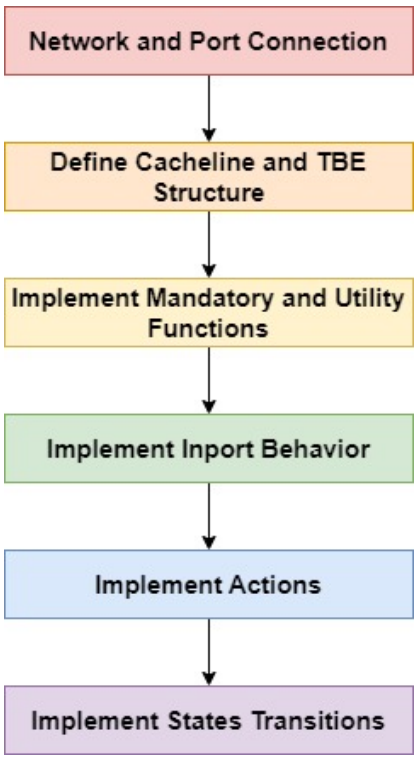


Figura 4.3: Fasi dell'implementazione della macchina a stati finiti della 1L Cache

Il primo passo consiste nel creare il sorgente: *gem5/src/learning_gem5/tardis_tso/TARDISTSO-cache.sm*. Il primo passo per l'implementazione consiste nella dichiarazione della struttura della cache di primo livello, tramite il costrutto machine, e della specifica del suo tipo, *MachineType:L1Cache*. Inoltre, gem5 mette a disposizione altri tipi di macchine rilevanti: Cache, L0Cache, L2Cache,

L3Cache, Directory e DMA. Ulteriori macchine sono specificate nel file *gem5/src/mem/ruby/protocol/RubySlic Exports.sm* in MachineType.

Tardis TSO L1 Cache

```
machine(MachineType:L1Cache, "TARDIS TSO L1 Cache"):
    //Global variables, Network Topology and Port Connection
    ...
{
    //Define Event, States, Cache Entry and TBE Structure
    ...

    //Mandatory and Utility Functions
    ...

    //Ports Behavior
    ...

    //Actions
    ...

    //State Transitions
    ...
}
```

4.3.1 Variabili Globali, Rete e Porte

L'implementazione comincia col dichiarare tutte le strutture e le variabili globali definite in fase di progettazione, abbiamo:

- Sequencer *sequencer: campo obbligatorio, inoltra le richieste (e.g. Load, Store) dal processore alla Cache, e restituisce le risposte dalla Cache al processore (e.g. restituire un dato).
- CacheMemory *cacheMemory: oggetto che contiene i veri blocchi della cache, sia dati sia istruzioni. Nel caso di Cache separate per Instruction Cache e Data Cache, si istanzieranno due campi differenti, per l'ICache e uno per la DCache.
- bool send_evictions: variabile utilizzata dai processori Out of Order per supportare l'mwait. Pertanto, non è obbligatorio qualora non si utilizzi tale processore.
- sts, lts e lease: timestamp globali e lease time (valore costante e relativo) da sommare al read timestamp.
- Cycles cache_response_latency e Cycles issue_latency: un Cycles è definito come la differenza (relativa) tra due istanti di tempo, espressa come un numero di cicli di clock. Cache_response_-latency e issue_latency corrispondono ai cicli di clock impiegati dalla Cache per fornire una

risposta (e.g. FLUSH_REP o WB_REP) o ad emettere una richiesta (SH_REQ o EX_REQ) rispettivamente. I loro valori saranno gli stessi delle controparti di altri protocolli, per renderli appunto confrontabili.

Dal momento che il sistema sarà composto da numerosi nodi interconnessi tra di loro, è necessario definire una topologia di interconnessione e definire il numero di buffer di snooping che leggeranno i messaggi che transitano in essa. Considerando una sola Cache, essa può essere dotata di uno o più buffer verso una o più interconnessioni. (e.g. è possibile assegnare un buffer di output e un buffer di input alla stessa interconnessione, o anche su interconnessioni diverse). Tuttavia, è necessario notare che in base alla topologia definita, possono instaurarsi dei deadlock. Per evitare ciò, assegneremo un buffer apposito per ogni tipologia di messaggio. Definiamo quindi quattro buffer e tre reti (senza considerare la rete della mandatoryQueue), come riportato anche in Figura 4.4:

- SH_REQ e EX_REQ: richieste inviate dalla Cache alla Directory, mediante la *request to directory*.
- FLUSH_REQ e WB_REQ: richieste inviate dalla Directory alla Cache proprietaria del blocco richiesto da un'altra Cache, mediante la *forward from directory*.
- SH_REP, EX_REP, RENEW_REP, UPGR_REP e ACK_REP: risposte di vario tipo inviate dalla Directory alla Cache, mediante la *response from directory*.
- FLUSH_REP, WB_REP e PUT_REP: risposte inviate dalla Cache alla Directory dovuto a richieste già citate, o Eviction di un blocco, mediante la *response to directory*

A queste aggiungamo un ulteriore buffer speciale obbligatorio, chiamato *mandatoryQueue*, nel quale verranno inserite tutte le richieste di Load e Store provenienti dal processore (e inoltrate dal Sequencer in tale buffer).

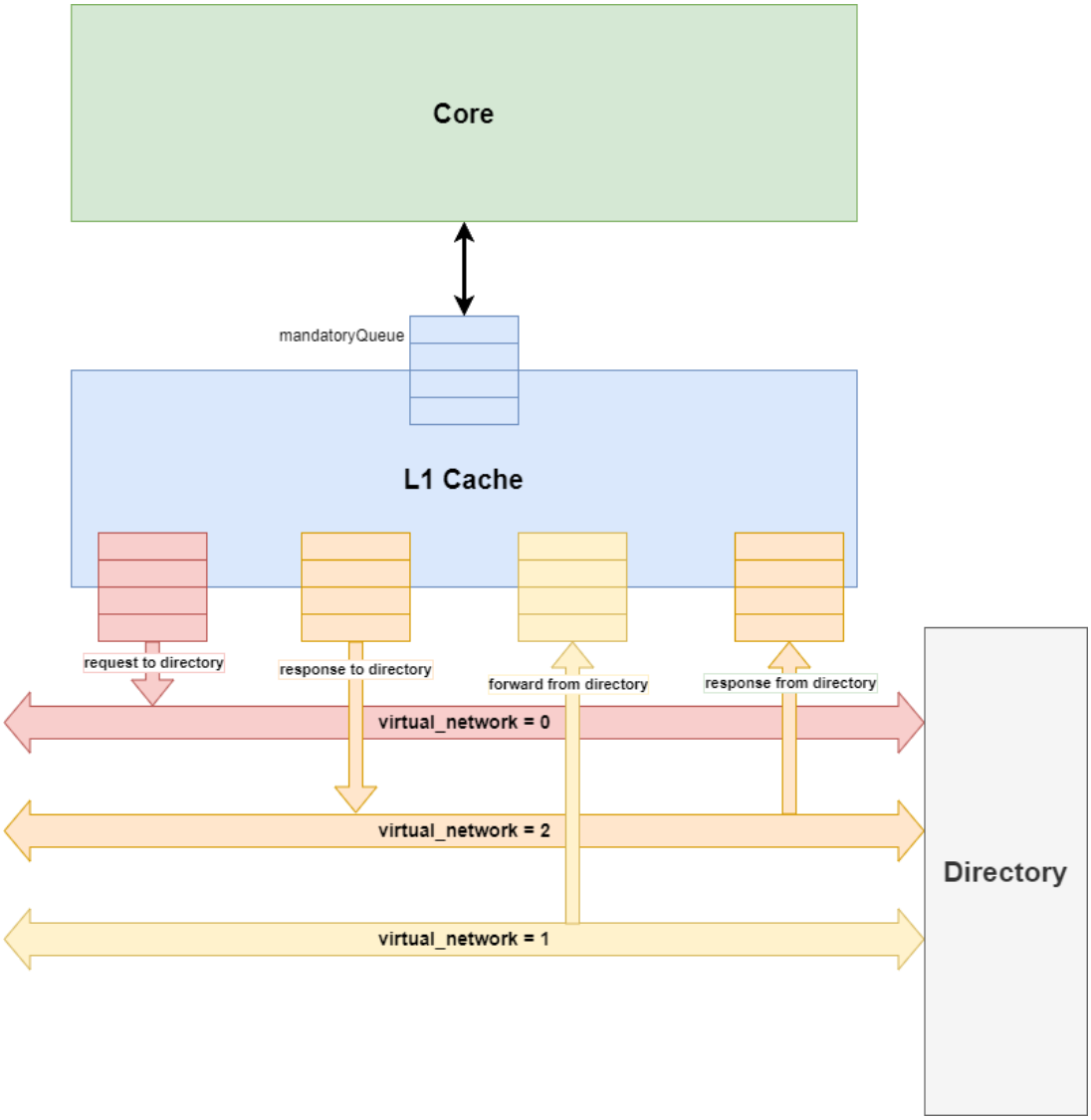


Figura 4.4: Topologia delle interconnessioni e dei buffer

Tardis TSO L1 Cache - Global Variables Network and Port Connection

```
machine(MachineType:L1Cache, "TARDIS TSO L1 Cache"):
  Sequencer *sequencer;
  CacheMemory *cacheMemory;
  bool send_evictions; // Needed to support O3 CPU and mwait
  int sts := 1;
  int lts := 1;
  int lease;
  Cycles cache_response_latency := 12;
  Cycles issue_latency := 2;

  MessageBuffer * requestToDir, network="To", virtual_network="0",
     vnet_type="request";
  MessageBuffer * responseToDir, network="To", virtual_network="2",
     vnet_type="response";

  MessageBuffer * forwardFromDir, network="From",
     virtual_network="1", vnet_type="forward";
  MessageBuffer * responseFromDir, network="From",
     virtual_network="2", vnet_type="response";

  MessageBuffer * mandatoryQueue;
{
...
}
```

Per ogni MessageBuffer, è necessario specificare la direzione del buffer (*From* se preleva o *To* se invia i messaggi all'interconnessione), l'identificativo dell'interconnessione *virtual_network*, e il tipo della rete con *vnet_type*.

4.3.2 Definizione di Eventi, Stati, Struttura Cache Entry e TBE

Il codice dichiara una struttura denominata state_declaration, utilizzata per definire e documentare i vari stati che un blocco in una cache può assumere, stabiliti in fase di Design. Ogni stato è specificato tramite un identificatore univoco, un attributo che rappresenta i permessi di accesso (AccessPermission), e una descrizione testuale (desc) che chiarisce il significato o il contesto dello stato.

Gli stati stabili I, S_LSD ed E, rappresentano situazioni statiche in cui un blocco si trova in una configurazione determinata e senza transizioni, come essere invalido, condiviso in modalità sola lettura, o posseduto esclusivamente con diritti di lettura e scrittura. Per questi stati, i permessi di accesso specificano i diritti del blocco in quel determinato stato:

- Invalid: il blocco è in uno stato invalido di base (e.g. il sistema è appena stato avviato o il blocco invalidato).
- Read_Only: il blocco è in modalità stato di sola lettura.
- Read_Write: il blocco è in modalità lettura e scrittura.
- NotPresent: il blocco non è presente in nessuna cache (e.g. tutte le cache sostituisono il blocco. Non è il caso di questo protocollo dato che non si può sapere quali cache hanno il blocco in lettura).
- Busy: il blocco è in uno stato transitorio.

Gli stati transitori, come IS, IE, o SE, modellano i momenti di transizione da uno stato stabile all'altro. Durante queste transizioni, il blocco si trova in una condizione intermedia e il L1 Cache Controller attende i messaggi dal resto del sistema (ad esempio, il blocco da una directory). Per rappresentare queste situazioni, l'attributo AccessPermission assume il valore Busy, indicando che il blocco non è immediatamente accessibile per le operazioni richieste.

Tardis TSO L1 Cache - States

```
machine(MachineType:L1Cache, "TARDIS TSO L1 Cache"):
{
    state_declaration(State, desc="Cache states") {
        I, AccessPermission:Invalid, desc="Not present/Invalid.";
        S_LSD, AccessPermission:Read_Only, desc="Shared Leased.
           Read-only, other caches may have the block.";
        E, AccessPermission:Read_Write, desc="Exclusive. Read and
           write permissions.";

        IS, AccessPermission:Invalid, desc="Not present/Invalid.
           Transistion state from Invalid to Shared. Wait for Data
           with Lease.";
        IE, AccessPermission:Invalid, desc="Not present/Invalid.
           Transistion state from Invalid to Exclusive. Wait for
           Data.";
        SE, AccessPermission:Busy, desc="Shared. Transition state
           from Shared to Exclusive. Wait an Exclusive or Upgrade
           response.";
        ES, AccessPermission:Busy, desc="Exclusive. Transition state
           from Exclusive to Shared";
        SS, AccessPermission:Busy, desc="Shared busy. Transition
           state from Exclusive to Shared. Waiting for Ack from
           directory.";
        S_EXP, AccessPermission:Busy, desc="Shared cacheline is
           expired. It need to wait for a RENEW_REP or SH_REP.";
        EI, AccessPermission:Busy, desc="Exclusive to Invalid due to
           Eviction.";
    }
}
```

La successiva struttura dichiara un'enumerazione denominata Event, progettata per rappresentare gli eventi che possono verificarsi in un sistema di gestione della cache. Ogni elemento dell'enumerazione è associato a un identificatore che descrive un particolare evento, accompagnato da una descrizione testuale (desc) che spiega il contesto e la natura dell'evento.

Gli eventi sono suddivisi tra quelli che provengono dal processore e quelli che derivano da interazioni con altri componenti del sistema, come la Directory. Gli eventi relativi al processore includono operazioni come Load e Store, che rappresentano rispettivamente il caricamento di dati e l'operazione di scrittura. Altri eventi come Eviction e LoadLeaseExpired indicano situazioni specifiche, ad esempio quando un blocco viene selezionato come vittima per essere rimosso o quando il tempo di leasing di un blocco è scaduto.

Gli eventi provenienti dal gestore di timestamp rappresentano comunicazioni e risposte del sistema

di coerenza. Ad esempio, ShRep e ExRep sono risposte che confermano rispettivamente l'accesso condiviso e l'accesso esclusivo a un blocco di dati. Altri eventi, come FlushReq o RenewRep, rappresentano richieste di pulizia della cache o risposte per rinnovare la validità di un blocco. Gli eventi come UpgRep e AckRep indicano risposte per aggiornamenti dei permessi o conferme di operazioni completate.
La struttura di questa enumerazione consente di modellare in modo esplicito e ordinato tutte le possibili interazioni tra il processore, la cache e le altre componenti del sistema, fornendo una chiara documentazione del comportamento atteso in risposta a ciascun evento. Questo è particolarmente utile per implementare e analizzare protocolli di coerenza in ambienti complessi.

Tardis TSO L1 Cache - Event enumeration

```
machine(MachineType:L1Cache, "TARDIS TSO L1 Cache"):
{
    enumeration(Event, desc="Cache events") {
        Load, desc="Load from processor";
        Store, desc="Store from processor";
        Eviction, desc="Triggered when block is chosen as victim";
        LoadLeaseExpired, desc="Triggered when a block is expired";
        ShRep, desc="Shared response from Timestamp Manager";
        ExRep, desc="Exclusive response from Timestamp Manager";
        FlushReq, desc="Flush request from Timestamp Manager";
        RenewRep, desc="Renew response from Timestamp Manager";
        UpgRep, desc="Upgrade response from Timestamp Manger";
        AckRep, desc="Ack Response from Timestamp Manager";
    }
}
```

Il codice definisce una struttura denominata Entry, utilizzata per rappresentare una singola voce della cache. Questa struttura include vari campi che descrivono lo stato e i metadati associati a un blocco di dati memorizzato nella cache. È associata a un'interfaccia astratta denominata AbstractCacheEntry. Ogni campo della struttura ha un tipo, un valore predefinito (se specificato) e una descrizione (desc) che fornisce informazioni sul ruolo del campo. I dettagli sono i seguenti:

- State CacheState: Specifica lo stato corrente della cache per questa entry, utilizzando la rappresentazione degli stati definita come sopra (in *state_declaration*).
- int wts (default=0): Rappresenta il timestamp dell'ultima scrittura per questo blocco. Questo valore è utilizzato per tenere traccia dell'ultima Store sul blocco.
- int rts (default=0): Rappresenta il lease time per il blocco. È utilizzato per determinare se un blocco è ancora valido per la lettura in base alla politica di lease progettata.
- int load_counter (default=0): Contatore per tracciare il numero di volte che il blocco è stato oggetto di una Load. Viene utilizzato dall'algoritmo di Livelock Prevention quando il blocco

rappresenta una variabile di spinlock. Quando questa variabile raggiunge il valore del livelock_-period, il blocco viene invalidato per forzarne il rinnovo ed acquisire la possibile variabile di spinlock aggiornata.

- int livelock_period (default=32): Rappresenta una soglia temporale dopo cui un certo numero di Load causano un'invalidazione del blocco per evitare il Livelock per variabili spinlock. Rappresenta una soglia temporale utilizzata per evitare situazioni di livelock, in particolare quelle associate a variabili di tipo spinlock. Questo meccanismo forza il sistema ad acquisire un nuovo blocco aggiornato contenente la variabile di spinlock potenzialmente modificata.
- DataBlock DataBlk: Contiene il blocco dati effettivo associato a questa voce della cache. DataBlock è una struttura già definita in gem5.

Tardis TSO L1 Cache - Entry Structure

```
machine(MachineType:L1Cache, "TARDIS TSO L1 Cache"):
{
    structure(Entry, desc="Cache entry",
        interface="AbstractCacheEntry") {
        State CacheState, desc="Cache state";
        int wts, default=0, desc="Write timestamp";
        int rts, default=0, desc="Read timestamp (lease)";
        int load_counter, default=0, desc="Number of times of Loads
            were made on this block. Used for Livelock prevention.";
        int livelock_period, default=32, desc="Period of livelock
            prevention.";
        DataBlock DataBlk, desc="Data in the block";
    }
}
```

La struttura TBE (Transaction Buffer Entry), funziona come un *Miss Status Handling Register*, e rappresenta una voce di buffer dedicata a richieste transitorie. Serve per tracciare informazioni di un blocco durante le transizioni di stato nel protocollo di coerenza. I suoi campi sono:

- State TBEState: Questo campo memorizza lo stato corrente del blocco nella transizione. Viene utilizzata la rappresentazione degli stati definita come sopra (in *state_ declaration*).
- DataBlock DataBlk: Questo campo contiene i dati associati al blocco durante la transizione. È essenziale per consentire il completamento delle operazioni di stato (come la consegna del blocco aggiornato o la risoluzione di conflitti).

In particolare, una sua Entry viene allocata (e i suoi campi popolati) quando comincia una transizione, e viene deallocata quando essa termina.
Viene definita un'ulteriore struttura, ovvero la tabella delle Entry di un registro TBE, ovvero la TBETable. La keyword external="yes" indica che l'implementazione dei metodi dichiarati all'interno di essa, sono definiti in moduli esterni già forniti da gem5.

Tardis TSO L1 Cache - TBE Structure

```
machine(MachineType:L1Cache, "TARDIS TSO L1 Cache"):
{
    structure(TBE, desc="Entry for transient requests") {
        State TBEState, desc="State of block";
        DataBlock DataBlk, desc="Data for the block. Needed for state
            transitions";
    }

    structure(TBETable, external="yes") {
        TBE lookup(Addr);
        void allocate(Addr);
        void deallocate(Addr);
        bool isPresent(Addr);
    }
}
```

4.3.3 Implementazione Funzioni di Utilità e Obbligatorie

Il passo successivo è di istanziare i componenti utili per l'esecuzione del protocollo.

- TBETable TBEs: Questa variabile rappresenta l'istanziazione della tabella TBE (Transaction Buffer Entry), utilizzata per monitorare blocchi in stati transitori. È configurata utilizzando un template <L1Cache_TBE> e un costruttore già fornito da gem5 che definisce il numero massimo di voci (m_number_of_TBEs).
- Tick clockEdge(): dal momento che anche la L1 Cache è una macchina sequenziale, e dato che gem5 simula anche il dominio del clock, è necessario dichiarare questo metodo (la cui implementazione è già fornita ed acquisita automaticamente da gem5 in un altro modulo), per sincronizzare le operazioni col fronte del clock.
- set_cache_entry() e unset_cache_entry(): Assegna o rimuove una entry della cache.
- set_tbe() e unset_tbe(): Assegna o rimuove una entry del TBE.
- mapAddressToMachine(): Questa funzione calcola quale macchina (ad esempio, una Directory o un'altra cache) deve gestire un indirizzo specifico. Serve per decidere la destinazione dei messaggi. L'implementazione è già fornita da un altro modulo da gem5.
- getCacheEntry: una funzione di utilità che, fornito un indirizzo, restituisce una entry della cache.

Tardis TSO L1 Cache - Implement Mandatory and Utility Functions

```
machine(MachineType:L1Cache, "TARDIS TSO L1 Cache"):
{
    TBETable TBEs, template="<L1Cache_TBE>",
       constructor="m_number_of_TBEs";
    Tick clockEdge();

    void set_cache_entry(AbstractCacheEntry a);
    void unset_cache_entry();
    void set_tbe(TBE b);
    void unset_tbe();

    MachineID mapAddressToMachine(Addr addr, MachineType mtype);

    Entry getCacheEntry(Addr address), return_by_pointer="yes" {
        return static_cast(Entry, "pointer",
           cacheMemory.lookup(address));
    }
}
```

Le successive funzioni devono essere reimplementate (override) in quanto sono state solo definite in maniera generica da gem5. Pertanto, esse devono essere riadattate al nostro protocollo e alle nostre strutture (Entry, TBE, etc.).

- getState(): è la funzione che restituisce lo stato del blocco. Essa va a leggere lo stato del blocco nel TBE, se esiste una voce ad esso associato (se è valido), altrimenti lo legge nella entry della cache.
- setState(): imposta lo stato del blocco sia nel TBE (se valido), sia nella Cache.
- getAccessPermission(), setAccessPermission(), functionalRead() e functionalWrite(): metodi necessari, utilizzati da gem5.

Tardis TSO L1 Cache - Implement Mandatory and Utility Functions

```
machine(MachineType:L1Cache, "TARDIS TSO L1 Cache"):
{
    State getState(TBE tbe, Entry cache_entry, Addr addr) {
        if (is_valid(tbe)) { return tbe.TBEState; }
        else if (is_valid(cache_entry)) {
            return cache_entry.CacheState;
        }
        else { return State:I; }
    }

    void setState(TBE tbe, Entry cache_entry, Addr addr, State state) {
      if (is_valid(tbe)) { tbe.TBEState := state; }
      if (is_valid(cache_entry)) { cache_entry.CacheState := state; }
    }

    AccessPermission getAccessPermission(Addr addr) {
        TBE tbe := TBEs[addr];
        if(is_valid(tbe)) {
            return L1Cache_State_to_permission(tbe.TBEState);
        }

        Entry cache_entry := getCacheEntry(addr);
        if(is_valid(cache_entry)) {
            return
               L1Cache_State_to_permission(cache_entry.CacheState);
        }
        return AccessPermission:NotPresent;
    }

    void setAccessPermission(Entry cache_entry, Addr addr, State
       state) {
        if (is_valid(cache_entry)) {
            cache_entry.changePermission(
               L1Cache_State_to_permission(state) );
        }
    }
```

Tardis TSO L1 Cache - Implement Mandatory and Utility Functions

```
    void functionalRead(Addr addr, Packet *pkt) {
        TBE tbe := TBEs[addr];
        DPRINTF(RubySlicc, "functionalRead for address 0x%x\n", addr);
        if(is_valid(tbe)) {
            testAndRead(addr, tbe.DataBlk, pkt);
        } else {
            testAndRead(addr, getCacheEntry(addr).DataBlk, pkt);
        }
    }

    int functionalWrite(Addr addr, Packet *pkt) {
        TBE tbe := TBEs[addr];
        if(is_valid(tbe)) {
            if (testAndWrite(addr, tbe.DataBlk, pkt)) {
                return 1;
            } else {
                return 0;
            }
        } else {
            if (testAndWrite(addr, getCacheEntry(addr).DataBlk, pkt))
               {
                return 1;
            } else {
                return 0;
            }
        }
    }
}
```

Dato che SLICC non supporta nativamente il calcolo del massimo tra interi, è necessario implementare questi due semplici metodi, ovvero il calcolo del massimo tra 2 o 3 interi. Essi verranno utilizzati per comparare i timestamp.

Tardis TSO L1 Cache - Implement Mandatory and Utility Functions

```
machine(MachineType:L1Cache, "TARDIS TSO L1 Cache"):
{
    int max(int i1, int i2){
        if(i1 > i2){
            return i1;
        }else{
            return i2;
        }
    }

    int max(int i1, int i2, int i3){
        if (i1 >= i2 && i1 >= i3) {
            return i1;
        } else if (i2 >= i1 && i2 >= i3) {
            return i2;
        } else {
            return i3;
        }
    }
}
```

4.3.4 Implementazione Comportamento delle Porte

La sezione seguente si concentra sull'implementazione del comportamento delle porte, ovvero l'insieme di operazioni che il controller della cache deve eseguire quando riceve un messaggio tramite una porta di input. Le porte di uscita, essendo destinate esclusivamente all'invio di messaggi, non richiedono un comportamento specifico e sono definite solo per completare il modello della macchina. Al contrario, le porte di input richiedono un'implementazione dettagliata per processare i messaggi in arrivo e avviare le transizioni di stato appropriate. L'ordine di dichiarazione delle porte di ingresso determina anche la loro priorità. La prima dichiarata (e che avrà la priorità maggiore) sarà la response_in, per dare priorità alla terminazione delle transazioni, similmente a come accade nei protocolli di rete (priorità agli Ack piuttosto che altri messaggi). Infatti, se arrivano due messaggi: uno di risposta dalla directory e una di richiesta dal processore, la risposta della directory verrà gestita per prima, mentre la richiesta del processore sarà gestita successivamente.
La dichiarazione di una porta avviene nel seguente modo:
in_port(alias della porta di ingresso, tipi di messaggi su questa porta, MessageBuffer associato);
out_port(alias della porta di uscita, tipi di messaggi su questa porta, MessageBuffer associato);
Per le porte di ingresso, il pattern comune prevede di utilizzare il metodo isReady(clockEdge()) per avviare il processamento del messaggio al fronte del clock; poi usare il metodo peek(porta di ingresso, struttura del messaggio), che popola una variabile implicita *in_ msg*, la cui struttura è quella

passata come parametro nel metodo peek(). Grazie alla variabile in_msg è possibile ricavare tutte le informazioni riguardanti il messaggio, come indirizzo del blocco, nodo mittente e nodo destinatario, tipo del messaggio, eventuali datablock, timestamp e così via. Infine, tramite una serie di if-else if-else per verificare il tipo di messaggio, è possibile scatenare un evento di transizione dello stato, tramite *trigger()*.
La firma della funzione trigger() cambia a seconda delle strutture dichiarate nella macchina a stati finiti, in particolare, se non viene dichiarata una struttura TBE e Cache Entry in questo file, la sua firma possiede solo i due seguenti parametri:
trigger(Evento, indirizzo del blocco);
Se è presente la dichiarazione della Cache Entry, la firma prevede un campo aggiuntivo:
trigger(Evento, indirizzo del blocco, entry della cache);
Infine, se è presente anche la dichiarazione del TBE, la firma prevede un ulteriore campo aggiuntivo:
trigger(Evento, indirizzo del blocco, entry della cache, tbe);
Ulteriori funzioni utili in fase di debug sono l'assert() e la DPRINTF(). L'assert permette di asserire una certa condizione per garantire il corretto comportamento del protocollo, infatti li utilizzeremo per garantire che il destinatario sia quello corretto, e che il mittente sia sempre la Directory. In caso di assert falso, l'esecuzione verrà abortita restituendo l'errore associato. La DPRINTF() è un'importante funzione di debug, e prevede che sia specificato un flag di debug (tipicamente in SLICC: RubySlicc o ProtocolTrace), una stringa da formattare, e i parametri, come le tipiche printf() in C, in quanto SLICC non le supporta nativamente. Un ulteriore flag è Exec, il quale restituisce le istruzioni eseguite dal processore.

Tardis TSO L1 Cache - Implement Ports Behavior

```
machine(MachineType:L1Cache, "TARDIS TSO L1 Cache"):
{
    out_port(request_out, RequestMsg, requestToDir);
    out_port(response_out, ResponseMsg, responseToDir);

    in_port(response_in, ResponseMsg, responseFromDir) {
        if (response_in.isReady(clockEdge())) {
            peek(response_in, ResponseMsg) {
                assert(in_msg.Destination.isElement(machineID));
                assert(machineIDToMachineType(in_msg.Sender) ==
                   MachineType:Directory);
                DPRINTF(RubySlicc, "Received in response_in %s with
                   address 0x%x\n", in_msg.Type, in_msg.addr);

                Entry cache_entry := getCacheEntry(in_msg.addr);
                TBE tbe := TBEs[in_msg.addr];
                if(in_msg.Type == CoherenceResponseType:SH_REP) {
                    trigger(Event:ShRep, in_msg.addr, cache_entry,
                       tbe);
                }
                else if(in_msg.Type == CoherenceResponseType:EX_REP){
                    trigger(Event:ExRep, in_msg.addr, cache_entry,
                       tbe);
                }
                else if(in_msg.Type ==
                   CoherenceResponseType:UPGR_REP){
                    trigger(Event:UpgRep, in_msg.addr, cache_entry,
                       tbe);
                }
                else if(in_msg.Type ==
                   CoherenceResponseType:RENEW_REP){
                    trigger(Event:RenewRep, in_msg.addr, cache_entry,
                       tbe);
                }
                else if(in_msg.Type == CoherenceResponseType:ACK_REP){
                    trigger(Event:AckRep, in_msg.addr, cache_entry,
                       tbe);
                }
                else{
                    error("Coherence Response Type not recognized.");
                }
            }
        }
    }
```

Il successivo blocco dichiara la seconda porta di Input, che avrà una priorità inferiore a quella dichiara sopra, in particolare essa è una porta di forward delle richieste, principalmente per Writeback e Flush, quando un blocco in stato di Exclusive viene richiesto da un'altra Cache o per la lettura o per la scrittura. Osserviamo che al messaggio di Writeback corrisponde a un evento di FlushReq, coerentemente all'automa progettato. Se si utilizzasse un altro evento, presumibilmente WbReq e WbRep, il protocollo andrebbe in deadlock dato che in uno stesso ciclo una L1 Cache potrebbe sia ricevere il messaggio di forwarding, sia inviare il blocco richiesto da tale messaggio alla Directory dovuto a un'Eviction: la L1 Cache non potrebbe soddisfare la richiesta del forwarding dato che è in attesa di ricevere un Ack dalla Directory, mentre la Directory non saprebbe mai di aver ricevuto già il blocco (tramite Eviction) e che la Cache è in attesa di ricevere l'Ack.

Tardis TSO L1 Cache - Implement Ports Behavior

```
machine(MachineType:L1Cache, "TARDIS TSO L1 Cache"):
{
    in_port(forward_in, RequestMsg, forwardFromDir) {
        if (forward_in.isReady(clockEdge())) {
            peek(forward_in, RequestMsg) {
                assert(in_msg.Destination.isElement(machineID));
                assert(machineIDToMachineType(in_msg.Requestor) ==
                   MachineType:Directory);
                DPRINTF(RubySlicc, "Received in forward_in %s with
                   address 0x%x\n", in_msg.Type, in_msg.addr);

                Entry cache_entry := getCacheEntry(in_msg.addr);
                TBE tbe := TBEs[in_msg.addr];
                if (in_msg.Type == CoherenceRequestType:FLUSH_REQ) {
                    trigger(Event:FlushReq, in_msg.addr, cache_entry,
                       tbe);
                }
                else if (in_msg.Type == CoherenceRequestType:WB_REQ) {
                    trigger(Event:FlushReq, in_msg.addr, cache_entry,
                       tbe);
                }
                else {
                    error("Unexpected forward message!");
                }
            }
        }
    }
```

L'ultima In-port da implementare riguardano le richieste provenienti dal core. Queste sono inoltrate attraverso la coda mandatory_in e la struttura di queste richieste sono del tipo RubyRequest, già im-

plementate in gem5 (in */gem5/src/mem/ruby/protocol/RubySlicc_Types.sm*). I campi più rilevanti di tale struttura di messaggi sono i seguenti:

- Addr LineAddress: indirizzo oggetto della richiesta di Load o Store;
- RubyRequestType Type: tipo di richiesta avanzata dal processore verso la cache.

Il tipo di richiesta RubyRequestType è anch'essa già definita da gem5 nel file: *gem5/src/mem/ruby/protocol/RubySlicc_Exports.sm*. Le più rilevanti sono illustrate di seguito:

- **LD** accesso con Load.
- **ST** accesso con Store.
- **ATOMIC** accesso Load o Store atomico (deprecato).
- **ATOMIC_RETURN** accesso Load o Store atomica con restituzione del dato.
- **ATOMIC_NO_RETURN** accesso Load o Store atomica senza restituzione del dato.
- **IFETCH** accesso per acquisire un'istruzione.
- **REPLACEMENT** sostituzione forzata di un blocco.
- **FLUSH** cancellazione forzata di un blocco.

La funzione peek() viene utilizzata con il parametro aggiuntivo block_on, che blocca tutte le richieste se esiste già una richiesta in sospeso con lo stesso indirizzo. Queste vengono sbloccate quando viene soddisfatta la richiesta. A differenza delle altre porte di input, questa porta presenta la caratteristica distintiva di verificare la presenza dei blocchi e di effettuare controlli sui timestamp associati. Nello specifico, la prima operazione consiste nel verificare se è stata eseguita un'istruzione di barriera di memoria. In caso affermativo, è necessario sincronizzare i timestamp globali lts e sts, allineandoli al valore massimo tra i due.
La successiva operazione prevede l'ottenimento delle informazioni relative all'entry di cache (cache_entry) e al TBE (tbe) associati all'indirizzo richiesto (LineAddress). Nel caso in cui la cache_entry risulti invalida (il blocco non è presente) e non vi sia spazio sufficiente per caricare un nuovo blocco (verificato tramite cacheAvail()), viene selezionata un'entry vittima (victim_entry) da sostituire. Successivamente, si alloca una voce del TBE corrispondente alla vittima, e viene scatenato l'evento di Eviction, passando come parametri le voci selezionate per la sostituzione.
Se la condizione sopra descritta non si verifica allora o il blocco è già presente, o c'è abbastanza spazio per caricarne uno nuovo. Pertanto, si verifica il tipo di operazione avanzata dal processore (IFETCH, LD, ST o ATOMIC) tramite una serie di if-else if. Nel caso particolare della Load, viene dapprima eseguito l'algoritmo di Livelock Prevention, incrementando il load_counter della cache_entry oggetto della load e si verifica se tale variabile ha superato il valore di soglia livelock_period, in tal caso viene forzato l'incremento del lts e dimezzato il valore di soglia: in tal modo, load successive forzeranno ulteriori incrementi del lts e, alla fine, faranno scadere il blocco associato presumibilmente a una variabile di sincronizzazione di spinlock.

Le ultime operazioni delle load prevedono la verifica del timestamp, in particolare, se il valore del lts è inferiore al lease time (rts), allora il blocco è ancora valido (Load), altrimenti è scaduto (LoadLeaseExpired).

Tardis TSO L1 Cache - Implement Ports Behavior

```
machine(MachineType:L1Cache, "TARDIS TSO L1 Cache"):
{
  in_port(mandatory_in, RubyRequest, mandatoryQueue) {
    if (mandatory_in.isReady(clockEdge())) {
        peek(mandatory_in, RubyRequest, block_on="LineAddress") {
            DPRINTF(RubySlicc, "in_msg.isMemBarrier = %d\n",
               in_msg.isMemBarrier);
            if(in_msg.isMemBarrier == 1) {
                DPRINTF(RubySlicc, "[MEMBARRIER] Memory barrier
                   executed. Synchronizing timestampss\n");
                lts := max(lts, sts);
                sts := lts;
            }
            Entry cache_entry := getCacheEntry(in_msg.LineAddress);
            TBE tbe := TBEs[in_msg.LineAddress];
            DPRINTF(RubySlicc, "Received in mandatory_in %s with
               address 0x%x\n", in_msg.Type, in_msg.LineAddress);

            if (is_invalid(cache_entry) &&
               cacheMemory.cacheAvail(in_msg.LineAddress) == false ) {
                Addr addr :=
                   cacheMemory.cacheProbe(in_msg.LineAddress);
                Entry victim_entry := getCacheEntry(addr);
                TBE victim_tbe := TBEs[addr];
                trigger(Event:Eviction, addr, victim_entry,
                   victim_tbe);
            } else {
                if (in_msg.Type == RubyRequestType:IFETCH){
                    trigger(Event:Load, in_msg.LineAddress,
                       cache_entry, tbe);
                }
```

Tardis TSO L1 Cache - Implement Ports Behavior

```
                else if (in_msg.Type == RubyRequestType:LD) {
                    if(is_invalid(cache_entry)){
                        trigger(Event:Load, in_msg.LineAddress,
                           cache_entry, tbe);
                    }else{
                        DPRINTF(RubySlicc, "Address: 0x%x. Rts: %d.
                           Wts: %d. State: %s\n", in_msg.LineAddress,
                           cache_entry.rts, cache_entry.wts,
                           cache_entry.CacheState);
                        cache_entry.load_counter :=
                           cache_entry.load_counter +1;
                        if((cache_entry.load_counter)%
                           cache_entry.livelock_period == 0){
                            lts := lts + 1;
                            cache_entry.load_counter := 0;
                            cache_entry.livelock_period :=
                               cache_entry.livelock_period/2;
                            if(cache_entry.livelock_period == 0){
                                cache_entry.livelock_period := 1;
                            }
                        }

                        if(lts <= cache_entry.rts) {
                            trigger(Event:Load, in_msg.LineAddress,
                               cache_entry, tbe);
                        } else {
                            trigger(Event:LoadLeaseExpired,
                               in_msg.LineAddress, ache_entry, tbe);
                        }
                    }
                } else if (in_msg.Type == RubyRequestType:ST ||
                   in_msg.Type == RubyRequestType:ATOMIC) {
                    trigger(Event:Store, in_msg.LineAddress,
                       cache_entry, tbe);
                } else {
                    error("Unexpected type from processor");
                }
            }
        }
    }
}
```

4.3.5 Implmentazione Actions

Le actions altro non sono che una sequenza di operazioni, identificate da uno *shorthand*, che vengono eseguite durante le transizioni di stato per implementare il comportamento della macchina a stati finiti. Nel nostro caso, di rilevante importanza sono le operazioni sui timestamp, indicate col colore rosso nella fase di Design nella tabella dell'automa.
Le prime tre actions riguardano semplicemente l'invio del messaggio di richiesta alla directory, tramite la funzione enqueue (porta di uscita, struttura del messaggio, latenza di invio). In tale funzione è necessario popolare i diversi campi. Si noti che la variabile address già contiene il valore dell'indirizzo oggetto della transazione. Tra i campi del messaggio da inviare, bisogna anche specificare la dimensione del messaggio, che può essere di tipo Control se non contiene un blocco (come i messaggi di richiesta) o Data se lo contiene.

Tardis TSO L1 Cache - Implement Actions

```
action(sendSharedRequest, "sSR", desc="Send a SH_REQ to TM for a new
    block (load)"){
     DPRINTF(RubySlicc, "Send to dir SH_REQ with address 0x%x\n",
        address);
     enqueue(request_out, RequestMsg, issue_latency) {
         out_msg.addr := address;
         out_msg.Type := CoherenceRequestType:SH_REQ;
         out_msg.Destination.clear();
         out_msg.Destination.add(mapAddressToMachine(address,
            MachineType:Directory));
         out_msg.MessageSize := MessageSizeType:Control;
         out_msg.Requestor := machineID;
         out_msg.wts := 0;
         out_msg.lts := lts;
     }
}

action(sendExclusiveRequest, "sER", desc="Send a EX_REQ to TM for a
    new block (store)"){
     DPRINTF(RubySlicc, "Send to dir EX_REQ with address 0x%x\n",
        address);
     enqueue(request_out, RequestMsg, issue_latency) {
         out_msg.addr := address;
         out_msg.Type := CoherenceRequestType:EX_REQ;
         out_msg.Destination.clear();
         out_msg.Destination.add(mapAddressToMachine(address,
            MachineType:Directory));
         out_msg.MessageSize := MessageSizeType:Control;
         out_msg.Requestor := machineID;
         out_msg.wts := 0;
     }
}
```

L'action storeDatablock memorizza il blocco ricevuto sulla coda di risposta (response_in) nella cache. Qui viene utilizzata la funzione peek(), che popola i campi della struttura in_msg con i campi del messaggio presente in testa alla coda delle risposte. Invece, l'action storeTimestamps() copia i timestamp dal messaggio nella entry della cache, dopo aver asserito che la cache_entry sia valida. Infine, l'action updateLease semplicemente aggiorna il lease time (rts).

Tardis TSO L1 Cache - Implement Actions

```
action(storeDatablock, "gRB", desc="Get the datablock"){
    peek(response_in, ResponseMsg) {
        assert(is_valid(cache_entry));
        cache_entry.DataBlk := in_msg.DataBlk;
    }
}

action(storeTimestamps, "sTMs", desc="Store the timestamps"){
    peek(response_in, ResponseMsg){
        assert(is_valid(cache_entry));
        cache_entry.wts := in_msg.wts;
        cache_entry.rts := in_msg.rts;
    }
}

action(updateLease, "uL", desc="Update the lease time"){
    peek(response_in, ResponseMsg){
        cache_entry.rts := in_msg.rts;
    }
}
```

L'action sendShReqNewLease invia una richiesta di SH_REQ perchè è scaduto un blocco di cache ed è necessario rinnovare il lease o il blocco stesso qualora esso sia stato modificato da un'altra L1 Cache. L'action sendUpgradeReq invia una richiesta di EX_REQ per un blocco già presente nella L1 Cache che avanza tale messaggio alla directory, per ottenere anche i propri privilegi per la scrittura. In entrambi i casi la directory controllerà, tramite i timestamps, se i blocchi oggetto della richiesta sono già aggiornati o meno.

Tardis TSO L1 Cache - Implement Actions

```
action(sendShReqNewLease, "sSRNL", desc="Send a SH_REQ to TM to
    request a new Lease"){
     DPRINTF(RubySlicc, "Send to dir SH_REQ (expired) with address
        0x%x\n", address);
     enqueue(request_out, RequestMsg, issue_latency) {
         out_msg.addr := address;
         out_msg.Type := CoherenceRequestType:SH_REQ;
         out_msg.Destination.clear();
         out_msg.Destination.add(mapAddressToMachine(address,
            MachineType:Directory));
         out_msg.MessageSize := MessageSizeType:Control;
         out_msg.Requestor := machineID;
   out_msg.wts := cache_entry.wts;
         out_msg.sts := sts;
         out_msg.lts := lts;
     }
}

action(sendUpgradeReq, "sUR", desc="Send a EX_REQ to TM to request
    the upgrade"){
     DPRINTF(RubySlicc, "Send to dir EX_REQ (upgrade) with address
        0x%x\n", address);
     enqueue(request_out, RequestMsg, issue_latency) {
         out_msg.addr := address;
         out_msg.Type := CoherenceRequestType:EX_REQ;
   out_msg.Destination.clear();
         out_msg.Destination.add(mapAddressToMachine(address,
            MachineType:Directory));
         out_msg.MessageSize := MessageSizeType:Control;
         out_msg.Requestor := machineID;
         out_msg.wts := cache_entry.wts;
     }
}
```

L'action sendPutResponse invia semplicemente un blocco che deve essere sostituito a causa di un'Eviction da parte di una L1 Cache, allegando i propri timestamps. La directory acquisirà il blocco ed effettuerà il Writeback in memoria, ed aggiornerà il mts grazie ai timestamp ricevuti.

Tardis TSO L1 Cache - Implement Actions

```
action(sendPutResponse, "sPR", desc="Send a PUT message to TM due to
    an Eviction"){
     DPRINTF(RubySlicc, "Send to dir PUT_REP with address 0x%x\n",
         address);
     enqueue(response_out, ResponseMsg, cache_response_latency){
          out_msg.addr := address;
          out_msg.Type := CoherenceResponseType:PUT_REP;
          out_msg.Sender := machineID;
          out_msg.Destination.clear();
          out_msg.Destination.add(mapAddressToMachine(address,
             MachineType:Directory));
          out_msg.MessageSize := MessageSizeType:Data;
          out_msg.DataBlk := cache_entry.DataBlk;
          out_msg.wts := cache_entry.wts;
          out_msg.rts := cache_entry.rts;
     }
}
```

L'action sendWritebackResponse semplicemente invia il blocco aggiornato che era in stato Exclusive tramite Writeback alla Directory a causa della richiesta in lettura (SH_REQ) o scrittura (EX_REQ) da parte di un'altra L1 Cache.

Tardis TSO L1 Cache - Implement Actions

```
action(sendWritebackResponse, "sWR", desc="Send to TM the datablock
    due Writeback Response"){
    DPRINTF(RubySlicc, "Send to dir WB_REP with address 0x%x\n",
        address);
    peek(forward_in, RequestMsg){
        enqueue(response_out, ResponseMsg, cache_response_latency) {
            out_msg.addr := address;
            out_msg.Type := CoherenceResponseType:WB_REP;
            out_msg.Sender := machineID;
            out_msg.Destination.clear();
            out_msg.Destination.add(mapAddressToMachine(address,
                MachineType:Directory));
            out_msg.MessageSize := MessageSizeType:Data;
            out_msg.DataBlk := cache_entry.DataBlk;
            cache_entry.rts := max(cache_entry.rts, cache_entry.wts +
                lease, in_msg.rts);
            out_msg.wts := cache_entry.wts;
            out_msg.rts := cache_entry.rts;
        }
    }
}
```

Le successive action loadHit, externalLoadHit e loadExHit servono per attuare le operazioni di Load effettive. In questo caso esse implementano anche le operazioni di aggiornamento dei timestamp descritte in fase di Design.

Tardis TSO L1 Cache - Implement Actions

```
action(loadHit, "lH", desc="Load hit") {
    assert(is_valid(cache_entry));
    cacheMemory.setMRU(cache_entry);
    lts := max(lts, cache_entry.wts);
    DPRINTF(RubySlicc, "Load or Ifetch on 0x%x with data %s\n",
       address, cache_entry.DataBlk);
    sequencer.readCallback(address, cache_entry.DataBlk, false);
}

action(externalLoadHit, "xLH", desc="External load hit (was a miss)")
   {
    assert(is_valid(cache_entry));
    peek(response_in, ResponseMsg) {
        cacheMemory.setMRU(cache_entry);
        lts := max(lts, cache_entry.wts);
        DPRINTF(RubySlicc, "Load or Ifetch on 0x%x with data %s\n",
           address, cache_entry.DataBlk);
        sequencer.readCallback(address, cache_entry.DataBlk, true,
           machineIDToMachineType(in_msg.Sender));
    }
}

action(loadExHit, "lEH", desc="Load for EX state") {
    assert(is_valid(cache_entry));
    cacheMemory.setMRU(cache_entry);
    lts := max(lts, cache_entry.wts);
    cache_entry.rts := max(lts, cache_entry.rts);
    DPRINTF(RubySlicc, "Load or Ifetch (store state) hit on 0x%x with
       data %s\n", address, cache_entry.DataBlk);
    sequencer.readCallback(address, cache_entry.DataBlk, false);
}
```

Le successive action storeHit ed externalLoadHit servono per attuare le operazioni di Store effettive. In questo caso esse implementano anche le operazioni di aggiornamento dei timestamp descritte in fase di Design.

Tardis TSO L1 Cache - Implement Actions

```
action(storeHit, "sH", desc="Store hit"){
    assert(is_valid(cache_entry));
    cacheMemory.setMRU(cache_entry);
    sts := max(sts, cache_entry.rts + 1);
    cache_entry.wts := sts;
    cache_entry.rts := sts;
    DPRINTF(RubySlicc, "Store on 0x%x with data %s\n", address,
       cache_entry.DataBlk);
    sequencer.writeCallback(address, cache_entry.DataBlk, false);
}

action(externalStoreHit, "xSH", desc="External store hit (was a
   miss)") {
    assert(is_valid(cache_entry));
    peek(response_in, ResponseMsg) {
        cacheMemory.setMRU(cache_entry);
        sts := max(sts, cache_entry.rts + 1);
        cache_entry.wts := sts;
        cache_entry.rts := sts;
        DPRINTF(RubySlicc, "Store on 0x%x with data %s\n", address,
           cache_entry.DataBlk);
        sequencer.writeCallback(address, cache_entry.DataBlk, true,
           machineIDToMachineType(in_msg.Sender));
    }
}
```

Le due action successive servono per allocare e deallocare una entry nella cache al certo indirizzo fornito. Con gli assert si garantisce che il blocco che si sta per allocare, non sia già allocato o che il blocco da deallocare sia presente.

Tardis TSO L1 Cache - Implement Actions

```
action(allocateCacheBlock, "aCB", desc="Allocate a cache block") {
    assert(is_invalid(cache_entry));
    assert(cacheMemory.cacheAvail(address));
    set_cache_entry(cacheMemory.allocate(address, new Entry));
}

action(deallocateCacheBlock, "dCB", desc="Deallocate a cache block") {
    assert(is_valid(cache_entry));
    cacheMemory.deallocate(address);
    unset_cache_entry();
}
```

Mentre le due seguenti action riguardano l'allocazione e la deallocazione di una entry nel TBE al certo indirizzo fornito. Con gli assert si garantisce che la entry che si sta per allocare, non sia già allocata o che la entry da deallocare sia presente. Verranno utilizzate all'inizio e alla fine di una transazione di memoria (no transizione di stato).

Tardis TSO L1 Cache - Implement Actions

```
action(allocateTBE, "aT", desc="Allocate TBE") {
    assert(is_invalid(tbe));
    TBEs.allocate(address);
    set_tbe(TBEs[address]);
}

action(deallocateTBE, "dT", desc="Deallocate TBE") {
    assert(is_valid(tbe));
    TBEs.deallocate(address);
    unset_tbe();
}
```

Le ultime action popMandatoryQueue, popResponseQueue e popForwardQueue servono a rimuovere il messaggio dalle code appropriate. Vanno utilizzate come ultima operazione in ogni transizione. L'action stall non effettua operazioni, e pertanto il messaggio resta in attesa di essere gestito.

Tardis TSO L1 Cache - Implement Actions

```
action(popMandatoryQueue, "pQ", desc="Pop the mandatory queue") {
    mandatory_in.dequeue(clockEdge());
}

action(popResponseQueue, "pR", desc="Pop the response queue") {
    response_in.dequeue(clockEdge());
}

action(popForwardQueue, "pF", desc="Pop the forward queue") {
    forward_in.dequeue(clockEdge());
}

action(stall, "z", desc="Stall the incoming request") {

}
```

4.3.6 Implementazione Transizioni di Stato

Nella seguente sezione vedremo come implementare le transizioni di stato dei blocchi e l'aggiunta delle relative azioni definite in precedenza. Una transizione viene definita secondo la sintassi *transition(stato di partenza, eventi, stato finale)*. Per una singola transizione è possibile raggruppare più eventi e, se non è presente lo stato finale, esso non cambia. Generalmente, l'ultima operazione che effettua ogni transizione, è la pop della coda associata al messaggio che è stato processato.
Le prime transizioni riguardano l'acquisizione di un blocco, inizialmente nello stato di Invalid, fino allo stato stabile S_LSD o E, a seconda del tipo di accesso. Quando un blocco è Invalid e viene richiesto tramite Load o Store, bisogna dapprima allocare dello spazio nel banco della cache, allocare il TBE dato che sta cominciando una transazione di memoria, inviare un messaggio di richiesta SharedRequest alla Directory ed infine, dato che le Load e le Store vengono accodate nella mandatoryQueue, effettuare la pop su quest'ultima.
Quando viene ricevuta una risposta (SH_REP o EX_REP), viene scatenato un evento di ShRep o ExRep ed il blocco transita in uno degli stati stabili S_LSD o E. In entrambi i casi, viene acquisito il blocco (storeDatablock) ed acquisiti i timestamp (wts e rts con storeTimestamps). Dal momento che la transazione di memoria è terminata, bisogna anche deallocare il TBE (deallocateTBE) e rispondere al processore con un hit: externalLoadHit per Load, externalStoreHit per Store (l'external sta ad indicare che l'hit è avvenuto dopo un miss). Infine, rimuovere il messaggio dalla coda delle risposte (popResponseQueue).

Tardis TSO L1 Cache - Implement State Transitions

```
machine(MachineType:L1Cache, "TARDIS TSO L1 Cache"):
{
    transition(I, Load, IS){
        allocateCacheBlock;
        allocateTBE;
        sendSharedRequest;
        popMandatoryQueue;
    }

    transition(I, Store, IE){
        allocateCacheBlock;
        allocateTBE;
        sendExclusiveRequest;
        popMandatoryQueue;
    }

    transition(IS, ShRep, S_LSD){
        storeDatablock;
        storeTimestamps;
        deallocateTBE;
        externalLoadHit;
        popResponseQueue;
    }

    transition(IE, ExRep, E){
        storeDatablock;
        storeTimestamps;
        deallocateTBE;
        externalStoreHit;
        popResponseQueue;
    }
```

Qualora il blocco si trovi in stato di S_LSD:

- Un evento di Load semplicemente restituisce un hit.
- Un evento di LoadLeaseExpired indica che il blocco è scaduto, quindi è necessario richiedere il rinnovo del lease alla deirectory.
- Un evento di Store scatena l'invio di un messaggio di UPGR_REQ alla Directory, per ottenere i permessi di scrittura sul blocco già memorizzato.

Invece, nel caso in cui il blocco si trovasse nello stato di S_EXP:

- Un evento di risposta RenewRep causa il rinnovo del lease senza aggiornare il blocco.
- Un evento di risposta ShRep causa l'aggiornamento sia del blocco dati sia dei timestamps.

Tardis TSO L1 Cache - Implement State Transitions

```
transition(S_LSD, Load){
    loadHit;
    popMandatoryQueue;
}

transition(S_LSD, LoadLeaseExpired, S_EXP){
    sendShReqNewLease;
    popMandatoryQueue;
}

transition(S_EXP, ShRep, S_LSD){
    storeDatablock;
    storeTimestamps
    externalLoadHit;
    popResponseQueue;
}

transition(S_EXP, RenewRep, S_LSD){
    updateLease;
    externalLoadHit;
    popResponseQueue;
}

transition(S_LSD, Store, SE){
    sendUpgradeReq;
    popMandatoryQueue;
}
```

Quando un blocco si trova nello stato SE, indica che sta transitando dallo stato di Shared a Exclusive, pertanto la L1 Cache è in attesa di ricevere dalla directory uno dei seguenti messaggi di riposta, che hanno effetti diversi (e quindi, funzioni differenti):

- Se la L1 Cache riceve un messaggio di ExRep, allora vuol dire che il blocco in stato di Shared già posseduto è stato nel frattempo modificato da un'altra L1 Cache, pertanto va aggiornato anche il blocco stesso (storeDatablock).
- Se la L1 Cache riceve un messaggio di UpgRep, allora vuol dire che il blocco in stato di Shared già posseduto non è stato modificato da un'altra L1 Cache, pertanto risulta necessario solo aggiornare i timestamp (updateLease).

Se il blocco si trova nello stato di Exclusive (E), e riceve eventi di Load o LoadLeaseExpired da parte del processore, essi verranno effettuati senza scadere il blocco, dato che, se è in stato di Exclusive, il blocco è sicuramente il più aggiornato. Il caso di evento di Store è banale.

Tardis TSO L1 Cache - Implement State Transitions

```
transition(SE, ExRep, E){
    storeDatablock;
    externalStoreHit;
    popResponseQueue;
}

transition(SE, UpgRep, E){
    updateLease;
    externalStoreHit;
    popResponseQueue;
}

transition(E, {Load, LoadLeaseExpired}){
    loadExHit;
    popMandatoryQueue;
}

transition(E, Store){
    storeHit;
    popMandatoryQueue;
}
```

Se un blocco in stato di Exclusive (E), riceve un evento di FlushReq allora significa che è stato richiesto un Writeback (sendWritebackResponse) dalla Directory (conseguenza del fatto che manca un LLC e quindi da quest'ultimo non possono scatenarsi eventi di FlushReq dovuti a Eviction del LLC).
Mentre se viene scatenato un evento di Eviction, allora viene inviato un messaggio di PUT alla Directory (sendPutResponse), che provvederà ad acquisire il blocco. La directory, dopo aver acquisito il blocco, effettuerà un Writeback verso la memoria e, alla ricezione del messaggio di acknoledgment da parte di quest'ultima, inoltrerà lo stesso (AckRep) alla Cache che ha sostituito il blocco. La cache che ha sostituito il blocco, alla ricezione dell'AckRep, provvederà anche a deallocarlo (deallocateCacheBlock).
Un'ulteriore osservazione va fatta per il messaggio di FlushReq per lo stato EI: come già descritto, può accadere che nello stesso ciclo di transazione (assunzione interconnessione pipelined) la cache sostituisce il blocco inviando una PUT, mentre un'altra lo richiede causando l'invio da parte della directory il messaggio di FlushReq. Se non ci fosse questa transizione, di stato Directory e la L1 Cache entrerebbero in uno stato di Deadlock. In caso di Eviciton di un blocco in stato di Shared,

non c'è bisogno di inviare un messaggio di PUT alla directory dato che:

- Se il blocco è stato nativamente acquisito da Invalid, allora per certo non sarà stato modificato.
- Se il blocco era in stato di Exclusive prima di diventare Shared, allora la funzione di Writeback garantisce che il blocco è stato già acquisito ed inoltrato alla memoria dalla Directory.

Tardis TSO L1 Cache - Implement State Transitions

```
    transition(E, FlushReq, S_LSD){
        sendWritebackResponse;
        popForwardQueue;
    }

    transition(E, Eviction, EI){
        sendPutResponse;
    }

    transition(EI, FlushReq, I){
        deallocateCacheBlock;
        popForwardQueue;
    }

    transition(EI, AckRep, I){
        deallocateCacheBlock;
        popResponseQueue;
    }

    transition(S_LSD, Eviction, I){
        deallocateCacheBlock;
    }

    transition({IS, IE, S_EXP, SE, EI}, {Load, Store, Eviction}){
         stall;
    }

    transition({IS, IE}, {FlushReq}){
        stall;
    }
}
```

4.4 Directory

La Directory (o Timestamp Manager) in questo caso è implementata in maniera simile alla L1 Cache, seguendo gli stessi passi descritti in Figura 4.3. Tuttavia è bene ricordare che, il compito del Timestamp Manager è assunto dal Last Level Cache, e non necessariamente dalla Directory. Infatti se avessimo avuto una gerarchia di cache a 3 livelli, il ruolo di Timestamp Manager spettava al LLC, ovvero la L3 Cache. Nonostante ciò, è comunque possibile implementare il protocollo tenendo però presente che eventuali operazioni di Eviction o memorizzazione del Datablock sono assenti. Il sorgente implementato sarà: *gem5/src/learning_gem5/tardis_tso/TARDISTSO-dir.sm*.

Tardis TSO Directory

```
machine(MachineType:Directory, "TARDIS TSO Directory"):
    //Global variables, Network Topology and Port Connection
    ...
{
    //Define Event, States, Directory Entry and TBE Structure
    ...

    //Mandatory and Utility Functions
    ...

    //Ports Behavior
    ...

    //Actions
    ...

    //State Transitions
    ...
}
```

4.4.1 Variabili Globali, Rete e Porte

Anche in questo caso, l'implementazione prevede di dichiarare tutte le strutture e le variabili globali definite in fase di progettazione, abbiamo:

- DirectoryMemory *directory: oggetto che contiene le entry della directory. Qui sono contenute tutte le informazioni riguardanti i blocchi memorizzati nelle cache.
- bool send_evictions: variabile utilizzata dai processori Out of Order per supportare l'mwait. Pertanto, non è obbligatorio qualora non si utilizzi tale processore.
- mts: il memory timestamp, necessario per ripristinare il valore del timestamp di un blocco memorizzato in memoria centrale. Infatti, i blocchi presenti solo in memoria centrale, non

possiedono un timestamp: il memory timestamp viene assegnato al blocco che viene prelevato dalla memoria prima di inviarlo a una cache. Esso deve essere sempre aggiornato al valore più recente, altrimenti si avrebbe incosistenza tra i timestamp dei blocchi prelevati dalla memoria e quelli nelle cache.

- Cycles directory_latency e Cycles to_memory_controller_latency: corrispondono rispettivamente al tempo di di risposta della directory e per inviare una richiesta alla memoria.

Definiremo sei buffer, di cui quattro si interfacceranno con le interconnessioni verso le cache, mentre altri due obbligatori verso la memoria. 4.4. Si noti che i messaggi e i MessageBuffer, saranno i duali di quelli presenti nella L1 Cache, invertendo i loro ruoli (input - output sulle interconnessioni).

- SH_REP, EX_REP, RENEW_REP, UPGR_REP: risposte inviate dalla Directory alle Cache richiedenti, mediante la *response to cache*.
- FLUSH_REQ e WB_REQ: richieste inviate dalla Directory alla Cache proprietaria del blocco richiesto da un'altra Cache, mediante la *forward to cache*.
- SH_REQ, EX_REQ, RENEW_REP, UPGR_REP e ACK_REP: risposte di vario tipo inviate dalla Directory alla Cache, mediante la *request from cache*.
- FLUSH_REP, WB_REP e PUT_REP: risposte ricevute dalla Directory dovute a richieste da essa avanzate per ottenere un blocco, esclusa la PUT_REP, che fa riferimento a una Eviction spontanea da parte della L1 Cache. Tramite il MessageBuffer *response from cache*

A queste aggiungamo due ulteriore buffer speciali obbligatori, chiamati *requestToMemory* e *responseFromMemory*: il primo serve ad inoltrare richieste di tipo writeback o di acquisizione di un blocco verso la memoria da parte della directory, mentre il secondo raccoglie le risposte da parte della memoria (blocchi o acknoledgement).

Tardis TSO Directory - Global Variables Network and Port Connection

```
machine(MachineType:Directory, "Directory protocol"):
    DirectoryMemory * directory;
    int mts := 1;
    int lease;   //Lease to add to the timestamp of a requested block
    Cycles directory_latency := 12;
    Cycles to_memory_controller_latency := 1;

    MessageBuffer *forwardToCache, network="To", virtual_network="1",
       vnet_type="forward";
    MessageBuffer *responseToCache, network="To",
       virtual_network="2", vnet_type="response";

    MessageBuffer *requestFromCache, network="From",
       virtual_network="0", vnet_type="request";
    MessageBuffer *responseFromCache, network="From",
       virtual_network="2", vnet_type="response";

    MessageBuffer *requestToMemory;
    MessageBuffer *responseFromMemory;

{
```

4.4.2 Definizione di Eventi, Stati, Struttura Cache Entry e TBE

Tramite la keyword state_declaration, è possibie dichiarare gli stati definiti in fase di Design. Anche in questo caso, ogni stato è specificato tramite un identificatore univoco, un attributo che rappresenta i permessi di accesso (AccessPermission), e una descrizione testuale (desc) che chiarisce il significato o il contesto dello stato. Sebbene gli stati sono rappresentati in maniera cache-centrici, i permessi di accesso saranno memoria-centrici.
Gli stati stabili, nel caso della directory sono I, S ed E.

- Invalid: il blocco è in uno strato invalido di base (e.g. il sistema è appena stato avviato).
- Read_Only: il blocco è in modalità stato di sola lettura da parte della memoria. Questo significa che il blocco è condiviso tra le cache e la memoria.
- Read_Write: il blocco è in modlità lettura e scrittura da parte della memoria e dunque è invalido nelle cache.
- Busy: il blocco è in uno stato transitorio.

Nel caso della directory, la struttura state_declaration prevede di specificare anche lo stato iniziale delle entry, secondo la notazione *"Directory_State_<Stato iniziale>"*.

Tardis TSO Directory - States

```
machine(MachineType:Directory, "TARDIS TSO Directory"):
{
    state_declaration(State, desc="Directory states",
       default="Directory_State_I") {
        I, AccessPermission:Read_Write, desc="Invalid in the caches";

        S, AccessPermission:Read_Only, desc="At least one cache has
           the block in Shared state";
        E, AccessPermission:Invalid, desc="A cache has the block in
           Exclusive state";

        IS, AccessPermission:Busy, desc="Invalid to Shared. Waiting
           for Memory data";
        IE, AccessPermission:Busy, desc="Invalid to Exclusive.
           Waiting for Memory data";
        SE, AccessPermission:Busy, desc="A block is moving from
           Shared to Exclusive state";
        EE, AccessPermission:Busy, desc="A block is moving from
           Exclusive to Exclusive state of another cache";
        E_m,AccessPermission:Busy, desc="A block is moving to
           Exclusive in writeback phase to memory waiting the ACK
           from it";
        ES, AccessPermission:Busy, desc="A block is moving from
           Exclusive to Shared state";
        SS_m,AccessPermission:Busy, desc="A block is moving to EX in
           writeback phase to memory waiting the ACK from it";
        ES_m, AccessPermission:Busy, desc="A block is moving from
           Exclusive or Shared to Shared state waiting the ACK from
           memory";
    }
}
```

Per quanto riguarda gli eventi, la Directory ne utilizza sempre due aggiuntivi speciali per la gestione delle risposte dalla memoria:

- Memory_Data: evento scatenato quando la directory riceve il blocco richiesto.
- Memory_Ack: evento scatenato quando la directory riceve il messaggio di acknoledgement dalla memoria, a seguito di un writeback.

Tardis TSO Directory - States

```
machine(MachineType:Directory, "TARDIS TSO Directory"):
{
    enumeration(Event, desc="Directory events") {
        ShReq, desc="Request for read-only data from cache";
        ExReq, desc="Request for read-write data from cache";
        FlushRep, desc="Received a block from the owner";
        UpgReq, desc="Send to a cache an Upgrade Rep";
        Memory_Data, desc="Data from memory";
        Memory_Ack, desc="Ack from memory that write is complete";
    }
}
```

La Directory, così come le L1 Cache, dovrà dichiarare le Entry per memorizzare le informazioni dei blocchi memorizzati e una tabella TBE con le relative voci per gestire le transazioni di memoria e transazioni da e verso le cache (e quindi transizioni intermedie).
La Entry memorizza lo stato del blocco, il write timestamp, il read timestamp, e una maschera per determinare l'owner del blocco. Tuttavia, è possibile definire quest'ultimo campo anche attraverso il *MachineID* invece di *NetDest*, dal momento che può esserci sempre un solo proprietario per il blocco.
Il TBE deve mantenere tutte le informazioni necessarie delle transazioni intermedie, tra cui il DataBlock, il nodo richiedente che ha iniziato la transizione, l'indirizzo del blocco e i timestamp tra cui il lts, utilizzato nei messaggi di SH_REQ per calcolare il nuovo lease.

Tardis TSO Directory - Entry and TBE

```
machine(MachineType:Directory, "TARDIS TSO Directory"):
{
    structure(Entry, desc="...", interface="AbstractCacheEntry",
       main="false") {
        State DirState, desc="Directory state";
        int wts, default=0, desc="Write timestamp";
        int rts, default=0, desc="Read timestamp";
        NetDest Owner, desc="Owner of this block";
    }

    structure(TBE, desc="TBE entries requests") {
        State TBEState,        desc="Transient State";
        Addr PhysicalAddress, desc="physical address";
        DataBlock DataBlk,     desc="Data to be written";
        MachineID Requestor, desc="requestor";
        int wts,  default=0, desc="";
        int rts,  default=0, desc="";
        int lts,  default=0, desc="Requestor's Load timestamp.";
    }

    structure(TBETable, external="yes") {
        TBE lookup(Addr);
        void allocate(Addr);
        void deallocate(Addr);
        bool isPresent(Addr);
    }

    TBETable TBEs, template="<Directory_TBE>",
       constructor="m_number_of_TBEs";
}
```

4.4.3 Implementazione Funzioni di Utilità e Obbligatorie

Molti oggetti e funzioni utilizzati dalla L1 Cache, verrano riutilizzati anche per la directory, in quanto servono allo stesso scopo. L'unica nuova funzione da implementare è la *getDirectoryEntry()* che alloca una nuova entry nella directory se non è già presente. Nel caso della directory però, non è necessario deallocare le entry in quanto Ruby le alloca mediante un meccanismo "lazy", ovvero se non è presente ne viene allocata una nuova, mentre se è già presente viene restituita la entry esistente.

Tardis TSO Directory - Mandatory and Utility Functions

```
machine(MachineType:Directory, "TARDIS TSO Directory"):
{
    Tick clockEdge();
    Cycles ticksToCycles(Tick t);
    Tick cyclesToTicks(Cycles c);
    void set_tbe(TBE b);
    void unset_tbe();

    Entry getDirectoryEntry(Addr addr), return_by_pointer="yes" {
        Entry dir_entry := static_cast(Entry, "pointer",
           directory[addr]);
        if (is_valid(dir_entry)) {
            return dir_entry;
        }
        dir_entry :=  static_cast(Entry, "pointer",
        directory.allocate(addr, new Entry));
        return dir_entry;
    }

    State getState(TBE tbe, Addr addr) {
        if (is_valid(tbe)) { return tbe.TBEState; }
        else if (directory.isPresent(addr)) {
            return getDirectoryEntry(addr).DirState;
        } else { return State:I; }
    }

    void setState(TBE tbe, Addr addr, State state) {
        if (is_valid(tbe)) { tbe.TBEState := state; }
        if (directory.isPresent(addr)) {
            if (state == State:E) {
                DPRINTF(RubySlicc, "Owners %s. Count: %d. Address:
                   0x%x\n", getDirectoryEntry(addr).Owner,
                   getDirectoryEntry(addr).Owner.count(), addr);
                assert(getDirectoryEntry(addr).Owner.count() == 1);
            }
            getDirectoryEntry(addr).DirState := state;
            if (state == State:I)  {
                assert(getDirectoryEntry(addr).Owner.count() == 0);
            }
        }
    }
```

Tardis TSO Directory - Mandatory and Utility Functions

```
    AccessPermission getAccessPermission(Addr addr) {
        TBE tbe := TBEs[addr];
        if(is_valid(tbe)) {
            return Directory_State_to_permission(tbe.TBEState);
        }
        if(directory.isPresent(addr)) {
            return Directory_State_to_permission(
               getDirectoryEntry(addr).DirState );
        }
        return AccessPermission:NotPresent;
    }

    void setAccessPermission(Addr addr, State state) {
        if (directory.isPresent(addr)) {
            getDirectoryEntry(addr).changePermission(
               Directory_State_to_permission(state) );
        }
    }

    void functionalRead(Addr addr, Packet *pkt) {
        functionalMemoryRead(pkt);
    }

    int functionalWrite(Addr addr, Packet *pkt) {
        int num_functional_writes := 0;
        TBE tbe := TBEs[addr];
        if(is_valid(tbe)) {
            num_functional_writes := num_functional_writes +
            testAndWrite(addr, tbe.DataBlk, pkt);
        }
        num_functional_writes := num_functional_writes +
           functionalMemoryWrite(pkt);
        return num_functional_writes;
    }
```

Tardis TSO Directory - Mandatory and Utility Functions

```
int max(int i1, int i2){
  if(i1 > i2){
    return i1;
  }else{
    return i2;
  }
}

int max(int i1, int i2, int i3){
  if (i1 >= i2 && i1 >= i3) {
        return i1;
  } else if (i2 >= i1 && i2 >= i3) {
    return i2;
  } else {
    return i3;
  }
}
```

4.4.4 Implementazione Comportamento delle Porte

Per le porte, inizialmente definiamo le porte di uscita, tra cui quella verso la memoria denominata memQueue_out, con una struttura di messaggi definita come MemoryMsg, già dichiara in gem5. I campi più importanti di questa struttura definita nel sorgente *gem5/src/mem/ruby/protocol/RubySlicc_MemControl.sm* sono i seguenti:

- Addr addr: indirizzo fisico della richiesta;
- MemoryRequestType Type: tipo della richiesta di memoria, che può essere MEMORY_READ per acquisire un blocco, o MEMORY_WB per scrivere in memoria un blocco modificato dalle cache.
- MachineID Sender: quale componente invia la richiesta.
- MachineID OriginalRequestorMachId: quale nodo iniziale ha avviato la transazioni (in questo caso, sarà una L1 Cache). Questo campo può essere utilizzato in alternativa al campo Requestor del TBE.
- DataBlock DataBlk: blocco dati contenuto nel messaggio.
- MessageSizeType MessageSize: grandezza del messaggio.

Tardis TSO Directory - Implement Ports Behavior

```
machine(MachineType:Directory, "TARDIS TSO Directory"):
{
    out_port(forward_out, RequestMsg, forwardToCache);
    out_port(response_out, ResponseMsg, responseToCache);
    out_port(memQueue_out, MemoryMsg, requestToMemory);
```

Per le porte di ingresso, il loro ordine di dichiarazione stabilisce anche la loro priorità. Anche in questo caso, verrà preferita la terminazione delle transazioni, gestendo prima i messaggi di risposta dalla memoria e dalle cache (messaggi di Writeback).
Per la porta di ingresso che riceve risposte dalla memoria, bisogna determinare se il messaggio è di acquisizione di un blocco MEMORY_READ, o di MEMORY_WB che rappresenta un acknoledgement di un avvenuto writeback.

Tardis TSO Directory - Implement Ports Behavior

```
machine(MachineType:Directory, "TARDIS TSO Directory"):
{
    in_port(memQueue_in, MemoryMsg, responseFromMemory) {
        if (memQueue_in.isReady(clockEdge())) {
            peek(memQueue_in, MemoryMsg) {
                DPRINTF(RubySlicc, "Received in memQueue_in %s with
                   address 0x%x\n", in_msg.Type, in_msg.addr);
                TBE tbe := TBEs[in_msg.addr];

                if (in_msg.Type == MemoryRequestType:MEMORY_READ) {
                    trigger(Event:Memory_Data, in_msg.addr, tbe);
                } else if (in_msg.Type ==
                   MemoryRequestType:MEMORY_WB) {
                    trigger(Event:Memory_Ack, in_msg.addr, tbe);
                } else {
                    error("Invalid message");
                }
            }
        }
    }
```

Per i messaggi di riposta, l'evento che scatenato è FlushRep, questo per evitare possibili condizioni di Deadlock quando si sovrappongono messaggi di tipo diverso (FLUSH_REP, WB_REP o PUT_REP) ma che di fatto sono la stessa cosa, ovvero dei messaggi di risposta per ottenere il blocco aggiornato da parte della directory.

Tardis TSO Directory - Implement Ports Behavior

```
machine(MachineType:Directory, "TARDIS TSO Directory"):
{
    in_port(response_in, ResponseMsg, responseFromCache) {
        if (response_in.isReady(clockEdge())) {
            peek(response_in, ResponseMsg) {
                assert(machineIDToMachineType(in_msg.Sender) ==
                   MachineType:L1Cache);
                Entry dir_entry := getDirectoryEntry(in_msg.addr);
                DPRINTF(RubySlicc, "Received in response_in %s with
                   address 0x%x from cache %s and state %s\n",
                   in_msg.Type, in_msg.addr, in_msg.Sender,
                   dir_entry.DirState);
                TBE tbe := TBEs[in_msg.addr];

                mts := max(mts, in_msg.rts);
                if (in_msg.Type == CoherenceResponseType:FLUSH_REP) {
                    trigger(Event:FlushRep, in_msg.addr, tbe);
                } else if(in_msg.Type ==
                   CoherenceResponseType:WB_REP) {
                    trigger(Event:FlushRep, in_msg.addr, tbe);
                } else if(in_msg.Type ==
                   CoherenceResponseType:PUT_REP) {
                    trigger(Event:FlushRep, in_msg.addr, tbe);
                } else {
                    error("Unexpected message type.");
                }
            }
        }
    }
```

Per l'ultima porta di ingresso, il procedimento è il medesimo. A parte per il fatto di aggiornare il read timestamp per la entry presente nelal directory nel caso di una SH_REQ. Per la EX_-REQ bisogna verificare se il messaggio è Upgradabile confrontando il write timestamp del messaggio ricevuto e quello memorizzato nella entry della directory: se essi sono uguali, allora la L1 Cache che ha avanzato la richiesta di tipo EX_REQ possiede già il blocco aggiornato, e dunque le verrà inviato un messaggio UPGR_REP senza acquisirlo da memoria. Invece, se il blocco non è aggiornato, ne verrà richiesto uno nuovo dalla memoria ed inoltrato al richiedente mediante un messaggio EX_REP (evento ExReq).

Tardis TSO Directory - Implement Ports Behavior

```
machine(MachineType:Directory, "TARDIS TSO Directory"):
{
    in_port(request_in, RequestMsg, requestFromCache) {
        if (request_in.isReady(clockEdge())) {
            peek(request_in, RequestMsg) {
                Entry dir_entry := getDirectoryEntry(in_msg.addr);
                TBE tbe := TBEs[in_msg.addr];
                DPRINTF(RubySlicc, "Received in request_in %s with
                   address 0x%x (%s) from cache %s. %d. Owner: %s.
                   Requestor==Owner %d \n", in_msg.Type, in_msg.addr,
                   dir_entry.DirState, in_msg.Requestor, mts,
                   dir_entry.Owner,
                   dir_entry.Owner.isElement(in_msg.Requestor));

                if (in_msg.Type == CoherenceRequestType:SH_REQ) {
                    dir_entry.rts := max(dir_entry.rts, dir_entry.wts
                       + lease, in_msg.lts + lease);
                    trigger(Event:ShReq, in_msg.addr, tbe);
                } else if(in_msg.Type == CoherenceRequestType:EX_REQ)
                   {
                    if(in_msg.wts == dir_entry.wts && in_msg.wts > 0){
                        trigger(Event:UpgReq, in_msg.addr, tbe);
                    }else{
                        trigger(Event:ExReq, in_msg.addr, tbe);
                    }
                } else {
                    error("Unexpected message type.");
                }
            }
        }
    }
```

4.4.5 Implmentazione Actions

La prima action da implementare, riguarda l'invio alla memoria di una richiesta MEMORY_READ per ottenere un blocco. L'originalRequestorMachId è il richiedente del blocco, ovvero la L1 Cache che ha avanzato una richiesta SH_REQ o EX_REQ. La lunghezza del messaggio Len è impostata a 0, per significare che il messaggio è una semplice richiesta.

Tardis TSO Directory - Actions

```
action(sendMemoryRequest, "sMR", desc="Send a memory request to DRAM
    to get a block"){
     DPRINTF(RubySlicc, "Send to memory a request with address
        0x%x\n", address);
     peek(request_in, RequestMsg){
          enqueue(memQueue_out, MemoryMsg,
             to_memory_controller_latency) {
               out_msg.addr := address;
               out_msg.Type := MemoryRequestType:MEMORY_READ;
               out_msg.Sender := machineID;
               out_msg.OriginalRequestorMachId := in_msg.Requestor;
               out_msg.MessageSize := MessageSizeType:Request_Control;
               out_msg.Len := 0;
          }
     }
}
```

Tardis TSO Directory - Actions

```
action(sendShDataToReq, "sSDTR", desc="Send the block in read
    permission to the Cache Requestor (with a lease)"){
    peek(memQueue_in, MemoryMsg) {
        Entry dir_entry := getDirectoryEntry(address);
        DPRINTF(RubySlicc, "Send to %s block 0x%x [%s]\n",
            tbe.Requestor, address, in_msg.DataBlk);
        dir_entry.rts := max(dir_entry.rts, dir_entry.wts + lease,
            tbe.lts + lease);
        enqueue(response_out, ResponseMsg, 1) {
            out_msg.addr := address;
            out_msg.Sender := machineID;
            out_msg.Destination.clear();
            out_msg.Destination.add(tbe.Requestor);
            out_msg.rts := dir_entry.rts;
            if(tbe.wts == dir_entry.wts && dir_entry.wts > 0){
                out_msg.Type := CoherenceResponseType:RENEW_REP;
                out_msg.MessageSize := MessageSizeType:Control;
            }else{
                out_msg.Type := CoherenceResponseType:SH_REP;
                out_msg.MessageSize := MessageSizeType:Data;
                out_msg.DataBlk := in_msg.DataBlk;
                out_msg.wts := dir_entry.wts;
            }
        }
    }
}
```

Anche la seguente action invia un messaggio di risposta SH_REP al richiedente come la precedente, ma i campi del messaggio di risposta vengono popolati con quelli presenti nel TBE.

Tardis TSO Directory - Actions

```
action(sendShTBEDataToReq, "sSDTRTBE", desc=""){
    Entry dir_entry := getDirectoryEntry(address);
    DPRINTF(RubySlicc, "Send (WB) to %s block 0x%x [%s]\n",
       tbe.Requestor, address, tbe.DataBlk);
    enqueue(response_out, ResponseMsg, 1){
        out_msg.addr := tbe.PhysicalAddress;
        out_msg.Type := CoherenceResponseType:SH_REP;
        out_msg.Sender := machineID;
        out_msg.Destination.clear();
        out_msg.Destination.add(tbe.Requestor);
        out_msg.DataBlk := tbe.DataBlk;
        out_msg.MessageSize := MessageSizeType:Data;
        dir_entry.rts := max(dir_entry.rts, dir_entry.wts + lease,
           tbe.lts + lease);
        out_msg.wts := dir_entry.wts;
        out_msg.rts := dir_entry.rts;
    }
}
```

Le successive due action rispondono a messaggi di tipo EX_REQ. La sendExDataToReq inoltra il blocco acquisito dalla memoria, mentre sendExTBEDataToReq invia un messaggio di EX_REP popolato con i valori presenti nel TBE (quest'ultima è utilizzata quando viene richiesto il blocco in scrittura da un'altra L1 Cache).

Tardis TSO Directory - Actions

```
action(sendExDataToReq, "sEDTR", desc="Send the block in read-write
    permission to the Cache Requestor"){
    peek(memQueue_in, MemoryMsg) {
        Entry dir_entry := getDirectoryEntry(address);
        DPRINTF(RubySlicc, "Send to %s block 0x%x [%s]\n",
            tbe.Requestor, address, in_msg.DataBlk);
        enqueue(response_out, ResponseMsg, 1) {
            out_msg.addr := address;
            out_msg.Type := CoherenceResponseType:EX_REP;
            out_msg.Sender := machineID;
            out_msg.Destination.clear();
            out_msg.Destination.add(tbe.Requestor);
            out_msg.DataBlk := in_msg.DataBlk;
            out_msg.MessageSize := MessageSizeType:Data;
            out_msg.rts := dir_entry.rts;
            out_msg.wts := dir_entry.wts;
        }
    }
}
```

Tardis TSO Directory - Actions

```
action(sendExTBEDataToReq, "sETBEDTR", desc=""){
    assert(is_valid(tbe));
    enqueue(response_out, ResponseMsg, 1){
        out_msg.addr := address;
        out_msg.Type := CoherenceResponseType:EX_REP;
        out_msg.Sender := machineID;
        out_msg.Destination.clear();
        out_msg.Destination.add(tbe.Requestor);
        out_msg.DataBlk := tbe.DataBlk;
        out_msg.MessageSize := MessageSizeType:Data;
    }
}
```

L'action sendWritebackRequest serve ad inviare un messaggio di writeback ed invia un nuovo rts aggiornato sommando col valore del lts del richiedente aggiunta una costante lease. L'action sendFlushReq serve ad inviare un messaggio di Flush all'owner del blocco per soddisfare la richiesta di un'altra L1 Cache che lo richiede in scrittura.

Tardis TSO Directory - Actions

```
action(sendWritebackRequest, "sWR", desc="Send a WB_REQ to the Owner
    due to a Shared request of another Cache"){
     peek(request_in, RequestMsg){
  Entry dir_entry := getDirectoryEntry(address);
        DPRINTF(RubySlicc, "Send to %s block 0x%x [wb req]\n",
           dir_entry.Owner, address);
        enqueue(forward_out, RequestMsg, directory_latency){
            out_msg.addr := address;
            out_msg.Type := CoherenceRequestType:WB_REQ;
            out_msg.Requestor := machineID;
            out_msg.Destination.clear();
            out_msg.Destination := dir_entry.Owner;
            out_msg.MessageSize := MessageSizeType:Control;
            out_msg.rts := in_msg.lts + lease;
        }
     }
}

action(sendFlushReq, "sFR", desc="Send a FLUSH_REQ to the owner due
    to an Exclusive request of another Cache"){
     peek(request_in, RequestMsg){
        Entry dir_entry := getDirectoryEntry(address);
        DPRINTF(RubySlicc, "Send flush request to %s with address
           0x%x [%s]\n", dir_entry.Owner, address);
        enqueue(forward_out, RequestMsg, directory_latency){
            out_msg.addr := address;
            out_msg.Type := CoherenceRequestType:FLUSH_REQ;
            out_msg.Requestor := machineID;
            out_msg.Destination.clear();
            out_msg.Destination := dir_entry.Owner;
            out_msg.MessageSize := MessageSizeType:Control;
        }
     }
}
```

L'action sendBlockToMem viene utilizzata per effettuare il writeback in memoria di un blocco che è stato modificato (e.g. Eviction), mentre la sendUpgradeRep invia solo un nuovo timestamp al richiedente del blocco, qualora fosse già aggiornato.

Tardis TSO Directory - Actions

```
action(sendBlockToMem, "sBTM", desc="When the TM get the block, send
    it to Memory and wait for the Ack"){
    DPRINTF(RubySlicc, "Send WB to memory address 0x%x\n", address);
    peek(response_in, ResponseMsg){
        enqueue(memQueue_out, MemoryMsg,
            to_memory_controller_latency) {
            out_msg.addr := address;
            out_msg.Type := MemoryRequestType:MEMORY_WB;
            out_msg.Sender := machineID;
            out_msg.MessageSize := MessageSizeType:Writeback_Data;
            out_msg.DataBlk := in_msg.DataBlk;
        }
    }
}

action(sendUpgradeRep, "sURes", desc=""){
    Entry dir_entry := getDirectoryEntry(address);
    DPRINTF(RubySlicc, "Send to cache an Upgrade response address
        0x%x\n", address);
    peek(request_in, RequestMsg){
        enqueue(response_out, ResponseMsg, directory_latency) {
            out_msg.addr := address;
            out_msg.Type := CoherenceResponseType:UPGR_REP;
            out_msg.Sender := machineID;
            out_msg.Destination.clear();
            out_msg.Destination.add(in_msg.Requestor);
            out_msg.MessageSize := MessageSizeType:Control;
            out_msg.rts := dir_entry.rts;
        }
    }
}
```

Per la Directory è necessario implementare le action che gestiscono la memorizzazione dell'Owner di una entry. In particolare, verranno utilizzate due action per impostare l'Owner:

- setOwner: imposta l'owner del blocco il requestor salvato nel TBE. Viene utilizzata solo nel caso di Upgrade, ovvero quando una L1 Cache richiede un blocco già posseduto in lettura, in modalità scrittura.

- setOwner_ReqIn: imposta l'owner del blocco come il requestor del messaggio in ingresso sulla coda delle richieste. È la versione che viene normalmente utilizzata per gestire le transazioni di blocchi richiesti in scrittura.

L'action clearOwner, rimuove l'owner del blocco.

Tardis TSO Directory - Actions

```
action(setOwner_ReqIn, "sOR", desc="Set the owner") {
    peek(request_in, RequestMsg){
        assert(machineIDToMachineType(in_msg.Requestor) ==
           MachineType:L1Cache);
        assert(getDirectoryEntry(address).Owner.count() == 0);
        getDirectoryEntry(address).Owner.add(in_msg.Requestor);
        DPRINTF(RubySlicc, "New owner (req in): %s\n",
           in_msg.Requestor);
    }
}

action(setOwner, "sO", desc="Set the owner") {
    assert(is_valid(tbe));
    assert(machineIDToMachineType(tbe.Requestor) ==
       MachineType:L1Cache);
    getDirectoryEntry(address).Owner.clear();
    getDirectoryEntry(address).Owner.add(tbe.Requestor);
    DPRINTF(RubySlicc, "New owner: %s\n", tbe.Requestor);
}

action(clearOwner, "cO", desc="Clear the owner") {
    getDirectoryEntry(address).Owner.clear();
}
```

Le successive action riguardano la semplice memorizzazione e gestione dei timestamp tra messaggi ricevuti, entry della directory e TBE.

- storeDataBlockTBE: memorizza il blocco ricevuto da una L1 Cache a seguito di una Eviction, Writeback o Flush. Viene utilizzata ogni volta che la directory deve trasferire il blocco in memoria.
- setMemoryTimestamps: copia il valore del mts nei campi wts ed rts della entry della directory. Viene utilizzata quando quest'ultima riceve un blocco dalla memoria.
- storeLtsTBE_ReqIn: memorizza nel TBE il timestamp lts ricevuto dal messaggio in testa nella coda delle richieste.
- storeLtsTBE_RepIn: memorizza nel TBE il timestamp lts ricevuto dal messaggio in testa nella coda delle risposte.

Tardis TSO Directory - Actions

```
action(storeDataBlockTBE, "sDBTBE", desc="Store in TBE entry the
    datablock from response queue."){
    peek(response_in, ResponseMsg) {
        assert(is_valid(tbe));
        tbe.DataBlk := in_msg.DataBlk;
    }
}

action(setMemoryTimestamps, "sMTs", desc="Set the mts in wts and rts
    field of directory entry."){
    peek(memQueue_in, MemoryMsg){
        Entry dir_entry := getDirectoryEntry(address);
        dir_entry.wts := mts;
        dir_entry.rts := mts;
    }
}

action(storeLtsTBE_ReqIn, "sLTBEREQ", desc="Store in TBE entry the
    lts from request queue."){
    peek(request_in, RequestMsg){
        assert(is_valid(tbe));
        tbe.lts := in_msg.lts;
    }
}

action(storeLtsTBE_RepIn, "sLTBEREP", desc="Store in TBE entry the
    lts from response queue."){
    peek(response_in, ResponseMsg){
        assert(is_valid(tbe));
        tbe.lts := in_msg.lts;
    }
}
```

Ulteriori action vengono definite per la gestione del timestamp:

- storeTimestampsTBE_ReqIn: memorizza nel TBE i write timestamp e read timestamp acquisiti dal messaggio in testa alla coda delle richieste.
- storeTimestampsTBE_RepIn: memorizza nel TBE i write timestamp e read timestamp acquisiti dal messaggio in testa alla coda delle risposte. Viene utilizzata quando le L1 Cache effettuano Eviction o inviano il blocco aggiornato tramite i messaggi di Flush.

- storeTimestamps_RepIn: memorizza nel'entry della directory i write timestamp e read timestamp acquisiti dal messaggio in testa alla coda delle risposte.

Tardis TSO Directory - Actions

```
action(storeTimestampsTBE_ReqIn, "sTTBEREQ", desc="Store in TBE entry
    the wts and rts from request queue."){
    peek(request_in, RequestMsg){
        assert(is_valid(tbe));
        tbe.wts := in_msg.wts;
        tbe.rts := in_msg.rts;
    }
}

action(storeTimestampsTBE_RepIn, "sTTBEREP", desc="Store in TBE entry
    the wts and rts from response queue."){
    peek(response_in, ResponseMsg){
        assert(is_valid(tbe));
        tbe.wts := in_msg.wts;
        tbe.rts := in_msg.rts;
    }
}
action(storeTimestamps_RepIn, "sTs", desc=" desc="Store in Directory
    entry the wts and rts from response queue.""){
    Entry dir_entry := getDirectoryEntry(address);
    peek(response_in, ResponseMsg){
        dir_entry.wts := in_msg.wts;
        dir_entry.rts := in_msg.rts;
    }
}
```

Il TBE viene popolato con le informazioni del messaggio ricevuto o dalla coda delle richieste a seguito di una SH_REQ o EX_REQ (allocateTBE_ReqIn) oppure dalla coda delle risposte dovuto a FLUSH_REP o un'Eviction (allocateTBE_RepIn).

Tardis TSO Directory - Actions

```
action(allocateTBE_ReqIn, "aTBE", desc="Allocate TBE with Request
    message info.") {
    peek(request_in, RequestMsg) {
        TBEs.allocate(address);
        set_tbe(TBEs[address]);
        tbe.PhysicalAddress := in_msg.addr;
        tbe.Requestor := in_msg.Requestor;
        DPRINTF(RubySlicc, "Allocated TBE [Requestor: %s, Address:
            0x%x]\n", tbe.Requestor, tbe.PhysicalAddress);
    }
}

action(allocateTBE_RepIn, "aTBER", desc="Allocate TBE with Response
    message info.") {
    peek(response_in, ResponseMsg) {
        TBEs.allocate(address);
        set_tbe(TBEs[address]);
        tbe.PhysicalAddress := in_msg.addr;
        tbe.Requestor := in_msg.Sender;
        DPRINTF(RubySlicc, "Allocated TBE Response [Requestor: %s,
            Address: 0x%x]\n", tbe.Requestor, tbe.PhysicalAddress);
    }
}

action(deallocateTBE, "w", desc="Deallocate TBE") {
    assert(is_valid(tbe));
    TBEs.deallocate(address);
    unset_tbe();
}
```

L'action sendAckRepTBE viene utilizzata per inviare un messaggio di Ack Response al richiedente memorizzato nel TBE. Essa serve ad informare una L1 Cache che ha effettuato un'Eviction che il writeback in memoria è terminato.

Tardis TSO Directory - Actions

```
action(sendAckRepTBE, "sART", desc="Send to requestor in TBE the
    acknoledgement response."){
    assert(is_valid(tbe));
    enqueue(response_out, ResponseMsg, 1){
        out_msg.addr := tbe.PhysicalAddress;
        out_msg.Type := CoherenceResponseType:ACK_REP;
        out_msg.Destination.clear();
        out_msg.Destination.add(tbe.Requestor);
        out_msg.Sender := machineID;
        out_msg.MessageSize := MessageSizeType:Control;
    }
}
```

Le ultime action, servono per rimuovere il messaggio in testa appena gestito. Pertanto un'action di pop sarà sempre l'ultima operazione che verrà eseguita in una transizione di stato.

Tardis TSO Directory - Actions

```
action(popResponseQueue, "pR", desc="Pop the response queue") {
    response_in.dequeue(clockEdge());
}

action(popRequestQueue, "pQ", desc="Pop the request queue") {
    request_in.dequeue(clockEdge());
}

action(popMemQueue, "pM", desc="Pop the memory queue") {
    dequeueMemRespQueue();
}

action(stall, "z", desc="Stall the incoming request") {
}
```

4.4.6 Implementazione Transizioni di Stat

Gli stati di transizione implementati seguono l'automa progettato in fase di Design, con le action implementate nella fase precedente. Caso particolare è la transizione da S a IS a seguito di una Shared Request: è necessaria per evitare sovrapposizioni di più richieste di uno stesso blocco. Tuttavia, causa un peggioramento delle prestazioni perché se più L1 Cache richiedono un blocco in lettura, nello stato di IS solo una richiesta per volta verrà gestita, ponendo in attesa tutte le altre. È possibile ovviare a tale problema introducendo un ulteriore stato e bufferizzare più Shared Request.

Le prime transizioni presentate, sono inerenti alle richieste di blocchi in stato di Invalid, ovvero non presenti in alcuna cache.

Tardis TSO Directory - Implement State Transitions

```
transition(I, ShReq, IS){
    allocateTBE_ReqIn;
    storeLtsTBE_ReqIn;
    storeTimestampsTBE_ReqIn;
    sendMemoryRequest;
    popRequestQueue;
}

transition(I, ExReq, IE){
    allocateTBE_ReqIn;
    sendMemoryRequest;
    popRequestQueue;
}
```

Quando la directory riceve il messaggio dalla memoria, inoltra il messaggio ai richiedenti.

Tardis TSO Directory - Implement State Transitions

```
transition(IS, Memory_Data, S){
    setMemoryTimestamps;
    sendShDataToReq;
    deallocateTBE;
    popMemQueue;
}

transition(IE, Memory_Data, E){
    setMemoryTimestamps;
    sendExDataToReq;
    setOwner;
    deallocateTBE;
    popMemQueue;
}
```

Tardis TSO Directory - Implement State Transitions

```
transition(S, ShReq, IS){
    allocateTBE_ReqIn;
    storeLtsTBE_ReqIn;
    storeTimestampsTBE_ReqIn;
    sendMemoryRequest;
    popRequestQueue;
}

transition(S, ExReq, SE){
    allocateTBE_ReqIn;
    sendMemoryRequest;
    popRequestQueue;
}

transition(S, UpgReq, E){
    setOwner_ReqIn;
    sendUpgradeRep;
    popRequestQueue;
}
```

Tardis TSO Directory - Implement State Transitions

```
transition(SE, Memory_Data, E){
    setMemoryTimestamps;
    sendExDataToReq;
    setOwner;
    deallocateTBE;
    popMemQueue;
}

transition(E, FlushRep, ES_m){
    allocateTBE_RepIn;
    storeTimestamps_RepIn;
    sendBlockToMem;
    clearOwner;
    popResponseQueue;
}

transition(ES_m, Memory_Ack, S){
    sendAckRepTBE;
    deallocateTBE;
    popMemQueue;
}
```

Tardis TSO Directory - Implement State Transitions

```
transition(E, ShReq, ES){
    allocateTBE_ReqIn;
    sendWritebackRequest;
    clearOwner;
    popRequestQueue;
}

transition(ES, FlushRep, SS_m){
    storeDataBlockTBE;
    storeLtsTBE_RepIn;
    storeTimestampsTBE_RepIn;
    storeTimestamps_RepIn;
    sendBlockToMem;
    popResponseQueue;
}

transition(SS_m, Memory_Ack, S){
    sendShTBEDataToReq;
    deallocateTBE;
    popMemQueue;
}
```

Tardis TSO Directory - Implement State Transitions

```
transition(E, {ExReq,UpgReq}, EE){
    allocateTBE_ReqIn;
    sendFlushReq;
    clearOwner;
    popRequestQueue;
}

transition(EE, FlushRep, E_m){
    storeDataBlockTBE;
    storeLtsTBE_RepIn;
    storeTimestampsTBE_RepIn;
    storeTimestamps_RepIn;
    sendBlockToMem;
    popResponseQueue;
}

transition(E_m, Memory_Ack, E){
    sendExTBEDataToReq;
    setOwner;
    deallocateTBE;
    popMemQueue;
}
```

Tardis TSO Directory - Implement State Transitions

```
transition({IS, IE}, {ShReq, ExReq, UpgReq}){
    stall;
}

transition({ES, SE, EE, ES_m, E_m, SS_m}, {ShReq, ExReq, UpgReq}) {
    stall;
}
```

4.5 Stub DMA

Il sorgente dell'automa del DMA di stub è riutilizzabile dal seguente protocollo: *gem5/src/mem/ruby/protocol/example-dma.sm*. A patto di collegare i MessageBuffer su due virtual_network isolate (e.g. 3 e 4) per non interferire col protocollo Tardis TSO. Questo sarà necessario in fase di Full System emulation, in quanto il boot di Linux richiede un DMA.

Capitolo 5

Debugging

Per testare il protocollo implementato, è necessario prima compilare gem5 scegliendo l'architettura e il formato di build (dettagli che verranno spiegati in seguito); poi selezionare il protocollo progettato; infine, eseguire una simulazione utilizzando uno dei due modelli di esecuzione disponibili: System Call Emulation o Full System Emulation. Per maggiore completezza, esploreremo entrambe le opzioni. Tutti i comandi verranno eseguiti nella directory root *gem5/*.

5.1 Preparazione per la Compilazione

Per compilare gem5 con Tardis TSO, è necessario creare i seguenti file che tengono traccia ed istruiscono il compilatore riguardo i sorgenti del protocollo.

gem5/src/learning_gem5/tardis_tso/SConsopts

```
Import('*')
main.Append(PROTOCOL_DIRS=[Dir('.')])
```

gem5/src/learning_gem5/tardis_tso/Kconfig

```
config PROTOCOL
    default "TARDISTSO" if RUBY_PROTOCOL_TARDISTSO

cont_choice "Ruby protocol"
    config RUBY_PROTOCOL_TARDISTSO
        bool "TARDISTSO"
endchoice
```

gem5/src/learning_gem5/tardis_tso/TARDISTSO.slicc

```
protocol "TARDISTSO";
include "RubySlicc_interfaces.slicc";
include "TARDISTSO-msg.sm";
include "TARDISTSO-dma.sm";
include "TARDISTSO-cache.sm";
include "TARDISTSO-dir.sm";
```

Bisogna poi aggiungere il percorso dei sorgenti nel file *gem5/src/Kconfig*

gem5/src/Kconfig

```
rsource "learning_gem5/tardis_tso/Kconfig"
```

Infine, per poter eseguire il protocollo in Full System emulation, è necessario aggiungere Tardis TSO alla lista dei protocolli disponibili in gem5, nel sorgente *gem5/src/python/gem5/coherence_protocol.py*

Adding the new protocol

```
class CoherenceProtocol(Enum):
    MESI_THREE_LEVEL = 1
    #...
    TARDISTSO = 12
```

5.2 Build Gem5

Per compilare gem5 (per versioni gem5 >= 23.1), bisogna preparare la directory della build con il seguente comando:

Setup build

```
scons defconfig build/<ISA>_<CACHE_PROTOCOL> build_opts/<ISA>
```

Dove <ISA> è una delle architettura disponibili in gem5: X86, Arm, Riscv, Sparc, Power, Mips; mentre <CACHE_PROTOCOL> è il nome del protocollo che abbiamo assegnato nel file *tardis_tso/Kconfig*.
Poi bisogna impostare i parametri della build, questi riguardano il protocollo di coerenza implementato:

Set build parameters

```
scons setconfig build/<ISA>_<CACHE_PROTOCOL> \
RUBY_PROTOCOL_<CACHE_PROTOCOL>=y SLICC_HTML=y
```

Ed infine è possibile compilare gem5:

Compile gem5

```
scons build/<ISA>_<CACHE_PROTOCOL>/gem5.opt -j<N_THREADS>
```

Dunque, nel nostro specifico caso, per compilare gem5 per l'architettura X86 con il protocollo di coerenza Tardis TSO, i comandi saranno:

Tardis TSO Protocol - Build gem5

```
scons defconfig build/X86_TARDISTSO build_opts/X86

scons setconfig build/X86_TARDISTSO \
RUBY_PROTOCOL_TARDISTSO=y SLICC_HTML=y

scons build/X86_TARDISTSO/gem5.opt -j6
```

La fase di compilazione, quando eseguita per la prima volta, può richiedere diverse ore. Tuttavia, nelle esecuzioni successive, il tempo necessario si riduce notevolmente, poiché vengono ricompilati solo i sorgenti che sono stati modificati.

5.2.1 Formato Build

La compilazione produce l'eseguibile gem5.opt nella cartella build/X86_TARDISTSO/. L'estensione .opt specifica il tipo di build, che determina il bilanciamento tra velocità di esecuzione e tracciabilità. Esistono diversi tipi di build, tra cui:

- **.debug** Compilato senza ottimizzazioni e con simboli di debug. Questo binario è utile quando si utilizza un debugger per eseguire il debug, nel caso in cui le variabili necessarie siano ottimizzate e rimosse nella versione opt di gem5. L'esecuzione con debug è più lenta rispetto agli altri binari.
- **.opt** Questo binario è compilato con la maggior parte delle ottimizzazioni attivate (ad esempio, -O3), ma include comunque i simboli di debug. È molto più veloce rispetto a debug, ma contiene ancora abbastanza informazioni di debug per risolvere la maggior parte dei problemi.
- **.fast** Compilato con tutte le ottimizzazioni attivate (comprese le ottimizzazioni a tempo di collegamento sui sistemi supportati) e senza simboli di debug. Inoltre, tutti gli assert sono

rimossi, ma le istruzioni di panic e fatal sono ancora presenti. fast è il binario con le migliori prestazioni ed è molto più piccolo rispetto a opt. Tuttavia, è consigliato solo quando si è sicuri che il codice non contenga bug critici.

5.3 System Call Emulation Mode

La System Call Emulation permette di eseguire applicazioni nel simulatore gem5 senza il supporto di un sistema operativo. Infatti, verranno chiamate appunto delle system call simulate già implementate in gem5. Tuttavia, non sarà presente un vero e proprio scheduler che permette di assegnare in maniera dinamica i diversi thread di un'applicazione ai core della CPU. La Syscall Emulation richiede che la configurazione del sistema preveda un numero di core pari al numero di thread definiti nel workload da eseguire, più un core aggiuntivo dedicato alla gestione. Il core di gestione ha il compito esclusivo di allocare i thread del workload sugli altri core disponibili. Nel caso in cui non ci siano sufficienti core disponibili, il programma andrà in crash.

5.3.1 Configurazione

Per utilizzare questa modalità, è necessario prima configurare il sistema di cache mediante un file Python, nella cartella */gem5/configs/ruby/*. In essa, creiamo il file *TARDISTSO.py*, che conterrà le specifiche inerenti ai controller di: L1 Cache, Directory e DMA in cui verranno inizializzate le variabili globali, collegati i MessageBuffer alla rete di interconnessione.

/gem5/configs/ruby/TARDISTSO.py

```
import math

import m5
from m5.defines import buildEnv
from m5.objects import *

from .Ruby import (
    create_directories,
    create_topology,
    send_evicts,
)

class L1Cache(RubyCache):
    pass


def define_options(parser):
    return
```

/gem5/configs/ruby/TARDISTSO.py

```
def create_system(options, full_system, system, dma_ports,
   bootmem, ruby_system, cpus):
    if buildEnv["PROTOCOL"] != "TARDISTSO":
        fatal("This script requires the TARDISTSO protocol to
           be built.")

    cpu_sequencers = []

    l1_cntrl_nodes = []
    dma_cntrl_nodes = []
    lease = 90

    block_size_bits = int(math.log(options.cacheline_size, 2))
```

La parte successiva, crea ed inizializza, per ogni core del sistema, un banco L1 Cache, un L1 Cache Controller e un RubySequencer. Al banco verrà impostata la grandezza *l1d_size* ed il grado di associatività *l1d_assoc*, che verrano impostati all'avvio della simulazione, o se omessi, inizializzati a valori predefiniti. Al L1 Cache Controller assegna il banco appena creato, il dominio del clock,

il sistema ruby a cui fa riferimento, e se ha bisogno di inviare le eviction per i processori O3. Il RubySequencer è il risponsabile dell'inoltro delle richieste di Load e Store al L1 Cache Controller da parte del processore. Infine, si inizializzano le code con i relativi MessageBuffer che verrano collegati alla rete di interconnessione *ruby_system.network*.

/gem5/configs/ruby/TARDISTSO.py

```
    for i in range(options.num_cpus):
        cache = L1Cache(
            size=options.l1d_size,
            assoc=options.l1d_assoc,
            start_index_bit=block_size_bits,
        )
        clk_domain = cpus[i].clk_domain
```

/gem5/configs/ruby/TARDISTSO.py

```
        l1_cntrl = L1Cache_Controller(
            version=i,
            cacheMemory=cache,
            send_evictions=send_evicts(options),
            transitions_per_cycle=options.ports,
            clk_domain=clk_domain,
            ruby_system=ruby_system,
        )

        cpu_seq = RubySequencer(
            version=i,
            dcache=cache,
            clk_domain=clk_domain,
            ruby_system=ruby_system,
        )
        l1_cntrl.lease = lease
        l1_cntrl.sequencer = cpu_seq
        exec("ruby_system.l1_cntrl%d = l1_cntrl" % i)

        # Add controllers and sequencers to the appropriate
           lists
        cpu_sequencers.append(cpu_seq)
        l1_cntrl_nodes.append(l1_cntrl)

        # Connect the L1 controllers and the network
        l1_cntrl.mandatoryQueue = MessageBuffer()
        l1_cntrl.requestToDir = MessageBuffer()
        l1_cntrl.requestToDir.out_port =
           ruby_system.network.in_port
        l1_cntrl.responseToDir = MessageBuffer()
        l1_cntrl.responseToDir.out_port =
           ruby_system.network.in_port

        l1_cntrl.forwardFromDir = MessageBuffer()
        l1_cntrl.forwardFromDir.in_port =
           ruby_system.network.out_port
        l1_cntrl.responseFromDir = MessageBuffer()
        l1_cntrl.responseFromDir.in_port =
           ruby_system.network.out_port
```

La successiva parte serve a creare i componenti della memoria e della Directory.

/gem5/configs/ruby/TARDISTSO.py

```
    phys_mem_size = sum([r.size() for r in system.mem_ranges])
    assert phys_mem_size % options.num_dirs == 0
    mem_module_size = phys_mem_size / options.num_dirs
    ruby_system.memctrl_clk_domain = DerivedClockDomain(
        clk_domain=ruby_system.clk_domain, clk_divider=3
    )

    mem_dir_cntrl_nodes, rom_dir_cntrl_node =
       create_directories(
        options, bootmem, ruby_system, system
    )
    dir_cntrl_nodes = mem_dir_cntrl_nodes[:]
    if rom_dir_cntrl_node is not None:
        dir_cntrl_nodes.append(rom_dir_cntrl_node)
    for dir_cntrl in dir_cntrl_nodes:
        dir_cntrl.lease = lease
        dir_cntrl.requestFromCache = MessageBuffer(ordered=True)
        dir_cntrl.requestFromCache.in_port =
           ruby_system.network.out_port
        dir_cntrl.responseFromCache =
           MessageBuffer(ordered=True)
        dir_cntrl.responseFromCache.in_port =
           ruby_system.network.out_port

        dir_cntrl.responseToCache = MessageBuffer()
        dir_cntrl.responseToCache.out_port =
           ruby_system.network.in_port
        dir_cntrl.forwardToCache = MessageBuffer()
        dir_cntrl.forwardToCache.out_port =
           ruby_system.network.in_port
        dir_cntrl.requestToMemory = MessageBuffer()
        dir_cntrl.responseFromMemory = MessageBuffer()
```

Il successivo blocco di codice, inizializza le porte del DMA di stub.

/gem5/configs/ruby/TARDISTSO.py

```
    for i, dma_port in enumerate(dma_ports):
        dma_seq = DMASequencer(version=i,
           ruby_system=ruby_system)

        dma_cntrl = DMA_Controller(
            version=i,
            dma_sequencer=dma_seq,
            transitions_per_cycle=options.ports,
            ruby_system=ruby_system,
        )

        exec("ruby_system.dma_cntrl%d = dma_cntrl" % i)
        exec("ruby_system.dma_cntrl%d.dma_sequencer.in_ports =
           dma_port" % i)
        dma_cntrl_nodes.append(dma_cntrl)

        # Connect the dma controller to the network
        dma_cntrl.mandatoryQueue = MessageBuffer()
        dma_cntrl.responseFromDir = MessageBuffer(ordered=True)
        dma_cntrl.responseFromDir.in_port =
           ruby_system.network.out_port
        dma_cntrl.requestToDir = MessageBuffer()
        dma_cntrl.requestToDir.out_port =
           ruby_system.network.in_port

    all_cntrls = (
        l1_cntrl_nodes + dir_cntrl_nodes + dma_cntrl_nodes
    )
```

Per completezza se fosse necessario eseguire in Full System Emulation utilizzando questo script, è necessario collegare il DMA al controller I/O se il sistema viene avviato con una configurazione Full System Emulation. Tuttavia, la FS Emulation la utilizzeremo in modo differente.

/gem5/configs/ruby/TARDISTSO.py

```
    if full_system:
        io_seq = DMASequencer(version=len(dma_ports),
           ruby_system=ruby_system)
        ruby_system._io_port = io_seq
        io_controller = DMA_Controller(
            version=len(dma_ports),
            dma_sequencer=io_seq,
            ruby_system=ruby_system,
        )
        ruby_system.io_controller = io_controller

        # Connect the dma controller to the network
        io_controller.mandatoryQueue = MessageBuffer()
        io_controller.responseFromDir =
           MessageBuffer(ordered=True)
        io_controller.responseFromDir.in_port =
           ruby_system.network.out_port
        io_controller.requestToDir = MessageBuffer()
        io_controller.requestToDir.out_port =
           ruby_system.network.in_port

        all_cntrls = all_cntrls + [io_controller]
```

L'ultima parte di codice, imposta il numero delle rete virtuali, che in questo caso saranno 5, di cui 3 sono utilizzate realmente dal protocollo per interconnettere Directory e Cache, mentre le altre 2 sono isolate dal protocollo di coerenza per il DMA stub, qualora si usasse questo script per la Full System Emulation.

/gem5/configs/ruby/TARDISTSO.py

```
    ruby_system.network.number_of_virtual_networks = 5
    topology = create_topology(all_cntrls, options)
    return (cpu_sequencers, mem_dir_cntrl_nodes, topology)
```

5.3.2 Workloads

I workload da eseguire con la Syscall emulation saranno creati e compilati con un OS Linux con un kernel version 5.15 o maggiore e architettura x86. Se i workload vengono compilati con una versione del kernel precedente, gem5 restituirà l'errore *Kernel too old*. I sorgenti verranno inseriti

nella cartella *gem5/tests/test-progs/tardis_tso/x86/*. Se è necessario eseguire i workload su altre architetture, bisogna prima compilare gem5 per tali ISA, ed infine utilizzare un cross-compilatore (se si è su x86) per generare gli eseguibili del target.

5.3.2.1 mfence

Il workload di esempio che utilizzeremo sarà presente nella cartella
gem5/tests/test-progs/tardis_tso/x86/mfence.
Il workload utilizza la libreria pthread per creare e gestire 4 thread concorrenti, ognuno dei quali modifica una variabile globale condivisa (shared_variable). Ogni thread esegue una funzione che incrementa la variabile condivisa in base al proprio identificatore in un ciclo di 10 iterazioni. Per garantire l'ordine delle operazioni di memoria e sincronizzare correttamente le modifiche, viene utilizzata un'istruzione assembly (mfence) che introduce una barriera di memoria. Questo meccanismo assicura che le scritture e letture della variabile condivisa siano completate in modo ordinato, evitando potenziali effetti di riordino causati dall'architettura hardware. Dopo l'esecuzione di tutti i thread, il programma stampa il valore finale della variabile condivisa.

/gem5/tests/test-progs/tardis_tso/x86/mfence/mfence.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define N_THREADS 4

// Variabili condivise tra i thread
int shared_variable = 0;

// Funzione eseguita dai thread
void *thread_function(void *arg) {
    int thread_id = *(int *)arg;
    for (int i = 0; i < 10; ++i) {
        __asm__ __volatile__ ("mfence" ::: "memory");
        shared_variable += thread_id;
    }
    printf("Thread %d ha completato l'incremento.\n",
       thread_id);
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[N_THREADS];
    for(int i=0; i<N_THREADS; i++){
        if(pthread_create(&threads[i], NULL, thread_function,
           (void *)&i) != 0){
            printf("Failed to create thread\n");
            return 1;
        }
    }

    for(int i=0; i<N_THREADS; i++){
        if (pthread_join(threads[i], NULL) != 0) {
            printf("Failed to join thread\ns");
            return 1;
        }
    }
    printf("Valore finale della variabile condivisa: %d\n",
       shared_variable);
    return 0;
}
```

5.3.2.2 spinlock

Il programma implementa un meccanismo di sincronizzazione basato su spinlock per proteggere l'accesso concorrente a una variabile condivisa (counter) modificata da più thread. Ogni thread esegue una funzione che incrementa il contatore condiviso per un numero prefissato di iterazioni (NUM_-ITERATIONS). La protezione del contatore è garantita dall'acquisizione e rilascio dello spinlock, che utilizza operazioni atomiche per evitare race condition. Durante l'acquisizione dello spinlock, viene applicato un meccanismo di busy-waiting, in cui il thread rimane in attesa attiva finché il lock non è disponibile. Questo serve a sfruttare il meccanismo di livelock detection and corretion implementato nell'automa della L1 Cache. Grazie a queste operazioni atomiche, il programma assicura che ogni incremento al contatore avvenga in modo mutuamente esclusivo, evitando conflitti tra i thread. Al termine dell'esecuzione, il valore finale del contatore viene stampato.

/gem5/tests/test-progs/tardis_tso/x86/spinlock/spinlock.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

// Spinlock definition
typedef struct {
    volatile int flag;
} spinlock_t;

// Spinlock initialization
void spinlock_init(spinlock_t *lock) {
    lock->flag = 0;
}

// Spinlock acquire
void spinlock_acquire(spinlock_t *lock) {
    while (__sync_lock_test_and_set(&lock->flag, 1)) {
        // Spin-wait (busy-wait) until the lock is acquired
    }
}

// Spinlock release
void spinlock_release(spinlock_t *lock) {
    __sync_lock_release(&lock->flag);
}

#define NUM_THREADS 4
#define NUM_ITERATIONS 1000

spinlock_t lock;
int counter = 0;
```

/gem5/tests/test-progs/tardis_tso/x86/spinlock/spinlock.c

```
// Thread function to increment the counter
void* thread_function(void* arg) {
    for (int i = 0; i < NUM_ITERATIONS; ++i) {
        spinlock_acquire(&lock);
        ++counter;
        spinlock_release(&lock);
    }
    return NULL;
}

int main() {
    pthread_t threads[NUM_THREADS];

    // Initialize the spinlock
    spinlock_init(&lock);

    // Create threads
    for (int i = 0; i < NUM_THREADS; ++i) {
        if (pthread_create(&threads[i], NULL, thread_function,
           NULL) != 0) {
            printf("Failed to create thread");
            return 1;
        }
    }

    // Wait for all threads to finish
    for (int i = 0; i < NUM_THREADS; ++i) {
        if (pthread_join(threads[i], NULL) != 0) {
            printf("Failed to join thread");
            return 1;
        }
    }
    printf("Final counter value: %d\n", counter);
    return 0;
}
```

5.3.3 Avvio Syscall Emulation

Una volta aver compilato il simulatore gem5 con architettura x86 e protocollo Tardis TSO, e aver compilato i sorgenti dei workload con gcc (linux kernel 5.15), è possibile visualizzare l'*help* per avviare la Syscall Emulation.

```
./build/X86_TARDISTSO/gem5.opt configs/deprecated/example/se.py
   --help
```

Le opzioni più rilevanti sono le seguenti:

- **-n NUM_CPUS**
- –mem-type CfiMemory,**DDR3_1600_8x8**,DDR3_2133_8x8,... e molti altri
- –mem-size MEM_SIZE
- –caches: se presente, aggiunge al sistema la gerarchia delle cache (con i valori di default se non sono specificati negli altri parametri). ATTENZIONE: abilita solo una semplice gerarchia di cache senza il protocollo di coerenza implementato.
- –l2cache: se presente, aggiunge al sistema le cache di livello 2
- –num-dirs NUM_DIRS –num-l2caches NUM_L2CACHES –num-l3caches NUM_L3CACHES: specifica il numero delle directory, dei banchi di L2 Cache e dei banchi di L3 Cache.
- –l1d_size L1D_SIZE –l1i_size L1I_SIZE : specifica la grandezza della L1 Instruction Cache (l1i) e della L1 Data Cache (l1d).
- –l2_size L2_SIZE –l3_size L3_SIZE: specifica la grandezza delle L2 e L3 Cache.
- –l1d_assoc L1D_ASSOC –l1i_assoc L1I_ASSOC –l2_assoc L2_ASSOC –l3_assoc L3_ASSOC: specifica il grado di associatività rispettivamente della L1 Data Cache, L1 Instruction Cache, L2 Cache ed L3 Cache.
- –cacheline_size CACHELINE_SIZE: specifica la grandezza di un blocco di cache.
- **–ruby**: abilita il sottosistema ruby con il protocollo di coerenza con cui è stato compilato gem5.
- –cpu-type {X86AtomicSimpleCPU, **X86TimingSimpleCPU**, X86MinorCPU, X86KvmCPU, X86O3CPU}: riportate solo le più importanti, specifica che tipo di CPU utilizzare.
- –cpu-clock CPU_CLOCK: specifica il clock della CPU.
- –debug-flags FLAGS: specifica se eseguire in modalità di debug. I flag sono diversi, di cui rilevanti RubySlicc, ProtocolTrace ed Exec.

Nel nostro caso, il comando per eseguire un workload spinlock (4 thread) con 4 processori TimingSimple è

```
./build/X86_TARDISTSO/gem5.opt configs/deprecated/example/se.py
   -c tests/test-progs/tardis_tso/x86/spinlock/bin/spinlock -n 5
   --cpu-type X86TimingSimpleCPU --ruby
```

Osserviamo che il numero di CPU da utilizzare è dato dal numero di thread specificato dal workload + 1 di gestione delle altre CPU. L'output mostrerà una serie di Warning che indicano l'emulazione delle system call chiamate dal workload.
Con il seguente comandi è possibile visualizzare ulteriori opzioni per avviare la simulazione, in particolare in modalità di debug.

```
./build/X86_TARDISTSO/gem5.opt --help
```

Infatti, per mostrare il comportamento del protocollo, e la variazione dello stato dei blocchi di Cache, è necessario avviare il workload in modalità di debug. Per fare questo, è necessario utilizzare le seguenti opzioni:

- –debug-flags FLAGS: specifica se eseguire in modalità di debug. I flag sono diversi, di cui rilevanti RubySlicc (utilizzato nei sorgenti degli automi), ProtocolTrace e Exec.
- –debug-start=TICK: specifica da quale TICK (numero intero) avviare la modalità di debug con i flag specificati sopra.
- –debug-end=TICK: specifica da quale TICK (numero intero) terminare la modalità di debug.
- –debug-file=FILE: specifica il file di output sul quale salvare i log.

Il seguente comando, avvia gem5 in modalità di debug dal tick 10000 fino al tick 1000000, abilitando i flag RubySlicc e ProtocolTrace. I log di debug verranno salvati nel file *gem5/m5out/trace.out*. Questa modalità permette di generare dei file di log estremamente dettagliati, fondamentali per correggere eventuali errori di runtime presenti nel protocollo.

```
./build/X86_TARDISTSO/gem5.opt --debug-flags=RubySlicc,
   ProtocolTrace --debug-start=10000 --debug-end=1000000 --debug-
   file=trace.out configs/deprecated/example/se.py -c tests/test-
   progs/tardis_tso/x86/spinlock/bin/spinlock -n 5 --cpu-type
   X86TimingSimpleCPU --ruby
```

5.3.4 Risultati

I risultati in tabella mostrano il tempo di esecuzione espresso in Tick del simulatore gem5 per i workload mfence e Spinlock, eseguiti entrambi con 4 thread su 4 core differenti TimingSimple.

Tabella 5.1: Risultati

PROTOCOL	Mfence	Spinlock
TARDISTSO	6013738500	1086121000
MI	4568366500	1637962000
MSI	4569435500	Non termina

Nel caso del workload mfence, è stato osservato che il tempo di esecuzione utilizzando il protocollo TARDISTSO risulta essere comparabile a quello degli altri protocolli MI e MSI quando quest'ultimi sono configurati con una cache di 1 livello.
I tempi di esecuzione per il workload Mfence con il protocollo TARDISTSO, con MI e MSI sono simili, indicando che TARDISTSO introduce penalizzazioni sensibilmente maggiori rispetto agli altri protocolli in scenari con cache di livello 1. Infatti, è necessario notare che TARDISTSO fa uso di messaggi di Renew e Upgrade, che rallentano l'esecuzione. Tramite l'ottimizzazione di tali politiche è possibile migliorare il protocollo e dunque, i tempi di esecuzione.
Nonostante ciò, per quanto riguarda il workload Spinlock, TARDISTSO ha mostrato prestazioni superiori rispetto a MI. Questo miglioramento è attribuito alla gestione più efficiente delle variabili di spinlock attraverso il meccanismo di Livelock Prevention. Inoltre, si nota che per il protocollo MSI, il workload Spinlock non ha completato l'esecuzione, segnalando potenziali problemi di performance o incompatibilità con questo tipo di workload.
In definita, sebbene TARDISTSO possa essere vantaggioso per casi come lo Spinlock, è possibile ottimizzare il protocollo per migliorare altri scenari tipici, questo tramite un'ulteriore riduzione di messaggi superflui di Renew ed Upgrade, utilizzando tecniche di Lease prediction e altre strategie tipiche dei protocolli più elaborati come MESI.

5.4 Full System Emulation Mode

La modalità di emulazione Full System è la più fedele ad esecuzioni reali, dato che entrano in gioco altri fattori come interrupt, devices, sistema operativo e i suoi servizi in esecuzione. Se il workload è pensato per essere eseguito su un sistema operativo, è bene utilizzare questa modalità.
La Full System Emulation richiede la creazione di una board virtuale completa, comprensiva di tutti i componenti ad essa associati, come la dimensione e il tipo di memoria, i dispositivi, i processori, la gerarchia delle cache e il protocollo di coerenza (Tardis TSO). Questo comporta la simulazione dell'intero sistema hardware, con un impatto significativo sui tempi di esecuzione rispetto alla Syscall Emulation. Ad esempio, in modalità Full System, il processo di avvio di Linux può richiedere dalle 2 alle 8 ore, a seconda dell'hardware della macchina host.

Inoltre, poiché la Full System Emulation esegue un sistema operativo Linux, è necessario un componente DMA (Direct Memory Access). Tuttavia, in questo lavoro è stato utilizzato un DMA di stub, che non è collegato alle interconnessioni del protocollo di coerenza.

5.4.1 Board

Testeremo il sistema su una architettura X86, in cui dipendono anche l'allocazione dello spazio di memoria dei dispositivi. Gem5 mette a disposizione una board emulata *X86Board* con dispositivi mappati, interruzione, dma e layout di memoria già configurati. Creiamo una cartella di lavoro in *gem5/configs/tardis_tso/* per aggiungere a tale board la configurazione del processore, del protocollo di coerenza e del workload.
Bisogna creare i componenti Python ed istanziarli nel sistema della Board. Viene creata una classe L1Cache derivata da una generica cache di livello 1 astratta, AbstractL1Cache. Bisogna implementare i seguenti metodi:

- __init__(): il costruttore. Qui inizializzeremo le variabili globali definite nell'automa, come il lease time.
- connectQueues(): collegamento della rete ai MessageBuffer definiti nell'automa. L'ordinamento dei messaggi dei MessageBuffer può essere di tipo FIFO (opzione *ordered=True* o arbitrario).

/gem5/configs/tardis_tso/tardis_tso_one_level/l1cache.py

```
from m5.objects import (ClockDomain, MessageBuffer, RubyCache)

from gem5.isas import ISA
from gem5.utils.override import overrides
from gem5.components.processors.abstract_core import
   AbstractCore
from
   gem5.components.cachehierarchies.ruby.caches.abstract_l1_cache
   import AbstractL1Cache

class L1Cache(AbstractL1Cache):
    def __init__(self, size: str, assoc: int,
        network, core: AbstractCore, cache_line_size,
        target_isa: ISA, clk_domain: ClockDomain, lease: int,
    ):
        super().__init__(network, cache_line_size)
        self.cacheMemory = RubyCache(
            size=size, assoc=assoc,
               start_index_bit=self.getBlockSizeBits()
        )
        self.clk_domain = clk_domain
        self.send_evictions = core.requires_send_evicts()
        self.lease = lease

    @overrides(AbstractL1Cache)
    def connectQueues(self, network):
        self.mandatoryQueue = MessageBuffer()
        self.requestToDir = MessageBuffer(ordered=True)
        self.requestToDir.out_port = network.in_port
        self.responseToDir = MessageBuffer()
        self.responseToDir.out_port = network.in_port
        self.forwardFromDir = MessageBuffer(ordered=True)
        self.forwardFromDir.in_port = network.out_port
        self.responseFromDir = MessageBuffer()
        self.responseFromDir.in_port = network.out_port
```

La directory collega anche i buffer alle interconnessioni già determinate in precedenza, e importa anche il valore del lease, che può essere passato come parametro.

/gem5/configs/tardis_tso/tardis_tso_one_level/directory.py

```
from m5.objects import (MessageBuffer, RubyDirectoryMemory)

from gem5.utils.override import overrides
from
   gem5.components.cachehierarchies.ruby.caches.abstract_directory
   import AbstractDirectory

class Directory(AbstractDirectory):
    def __init__(self, network, cache_line_size, mem_range,
       port, lease):
        super().__init__(network, cache_line_size)
        self.addr_ranges = [mem_range]
        self.directory = RubyDirectoryMemory()
        self.memory_out_port = port
        self.lease = lease

    @overrides(AbstractDirectory)
    def connectQueues(self, network):
        self.requestFromCache = MessageBuffer()
        self.requestFromCache.in_port = network.out_port
        self.responseFromCache = MessageBuffer()
        self.responseFromCache.in_port = network.out_port

        self.forwardToCache = MessageBuffer()
        self.forwardToCache.out_port = network.in_port
        self.responseToCache = MessageBuffer()
        self.responseToCache.out_port = network.in_port
        self.requestToMemory = MessageBuffer()
        self.responseFromMemory = MessageBuffer()
```

Ed infine creiamo il DMA di stub che verrà connesso nel sistema su interconnessioni isolate dal protocollo.

/gem5/configs/tardis_tso/tardis_tso_one_level/dma_controller.py

```
from m5.objects import MessageBuffer

from gem5.utils.override import overrides
from gem5.components.cachehierarchies.ruby.caches.
   abstract_dma_controller import AbstractDMAController


class DMAController(AbstractDMAController):
    class DMAController(AbstractDMAController):
        def __init__(self, network, cache_line_size):
            super().__init__(network, cache_line_size)

    @overrides(AbstractDMAController)
    def connectQueues(self, network):
        self.mandatoryQueue = MessageBuffer()
        self.requestToDir = MessageBuffer()
        self.requestToDir.out_port = network.in_port
        self.responseFromDir = MessageBuffer(ordered=True)
        self.responseFromDir.in_port = network.out_port
```

Il passaggio successivo consiste nel creare un ulteriore sorgente python che mette assieme i 3 componenti definiti, e formare così una singola classe che rappresenta il nostro protocollo. Pertanto procediamo alla creazione istanziando i diversi componenti utilizzando una rete Point2Point.

/gem5/configs/tardis_tso/tardis_tso_one_level_cache_hierarchy.py

```
from m5.objects import (
    DMASequencer,
    RubyPortProxy,
    RubySequencer,
    RubySystem,
)

from gem5.coherence_protocol import CoherenceProtocol
from gem5.isas import ISA
from gem5.utils.override import overrides
from gem5.utils.requires import requires
from gem5.components.boards.abstract_board import AbstractBoard
from gem5.components.cachehierarchies.abstract_cache_hierarchy
   import AbstractCacheHierarchy
from gem5.components.cachehierarchies.
   ruby.abstract_ruby_cache_hierarchy import
   AbstractRubyCacheHierarchy
from tardis_tso_one_level.directory import Directory
from tardis_tso_one_level.dma_controller import DMAController
from tardis_tso_one_level.l1_cache import L1Cache
from
   gem5.components.cachehierarchies.ruby.topologies.simple_pt2pt
   import SimplePt2Pt
```

/gem5/configs/tardis_tso/tardis_tso_one_level_cache_hierarchy.py

```
class
   TARDISTSOOneLevelCacheHierarchy(AbstractRubyCacheHierarchy):

    def __init__(self, size: str, assoc: str):
        super().__init__()
        self._size = size
        self._assoc = assoc

    @overrides(AbstractCacheHierarchy)
    def incorporate_cache(self, board: AbstractBoard) -> None:
        requires(
        coherence_protocol_required=CoherenceProtocol.TARDISTSO)

        self.ruby_system = RubySystem()

        # Ruby's global network.
        self.ruby_system.network = SimplePt2Pt(self.ruby_system)


        self.ruby_system.number_of_virtual_networks = 5
        self.ruby_system.network.number_of_virtual_networks = 5

        self._controllers = []
        for i, core in
           enumerate(board.get_processor().get_cores()):
            cache = L1Cache(
                size=self._size,
                assoc=self._assoc,
                network=self.ruby_system.network,
                core=core,
                cache_line_size=board.get_cache_line_size(),
                target_isa=board.get_processor().get_isa(),
                clk_domain=board.get_clock_domain(),
                lease=10,
            )
```

/gem5/configs/tardis_tso/tardis_tso_one_level_cache_hierarchy.py

```
            cache.sequencer = RubySequencer(
                version=i,
                dcache=cache.cacheMemory,
                clk_domain=cache.clk_domain,
            )

            if board.has_io_bus():
                cache.sequencer.connectIOPorts(board.get_io_bus())

            cache.ruby_system = self.ruby_system

            core.connect_icache(cache.sequencer.in_ports)
            core.connect_dcache(cache.sequencer.in_ports)

            core.connect_walker_ports(
                cache.sequencer.in_ports,
                   cache.sequencer.in_ports
            )

            # Connect the interrupt ports
            if board.get_processor().get_isa() == ISA.X86:
                int_req_port =
                   cache.sequencer.interrupt_out_port
                int_resp_port = cache.sequencer.in_ports
                core.connect_interrupt(int_req_port,
                   int_resp_port)
            else:
                core.connect_interrupt()

            cache.ruby_system = self.ruby_system
            self._controllers.append(cache)
```

/gem5/configs/tardis_tso/tardis_tso_one_level_cache_hierarchy.py

```
        # Create the directory controllers
        self._directory_controllers = []
        for range, port in board.get_mem_ports():
            dir = Directory(
                self.ruby_system.network,
                board.get_cache_line_size(),
                range,
                port,
                10,
            )
            dir.ruby_system = self.ruby_system
            self._directory_controllers.append(dir)

        # Create the DMA Controllers, if required.
        self._dma_controllers = []
        if board.has_dma_ports():
            dma_ports = board.get_dma_ports()
            for i, port in enumerate(dma_ports):
                ctrl = DMAController(
                    self.ruby_system.network,
                       board.get_cache_line_size()
                )
                ctrl.dma_sequencer = DMASequencer(version=i,
                   in_ports=port)

                ctrl.ruby_system = self.ruby_system
                ctrl.dma_sequencer.ruby_system =
                   self.ruby_system

                self._dma_controllers.append(ctrl)

        self.ruby_system.num_of_sequencers =
           len(self._controllers) + len(
            self._dma_controllers
        )
```

/gem5/configs/tardis_tso/tardis_tso_one_level_cache_hierarchy.py

```
        # Connect the controllers.
        self.ruby_system.controllers = self._controllers
        self.ruby_system.directory_controllers =
           self._directory_controllers

        if len(self._dma_controllers) != 0:
            self.ruby_system.dma_controllers =
               self._dma_controllers

        self.ruby_system.network.connectControllers(
            self._controllers
            + self._directory_controllers
            + self._dma_controllers
        )
        self.ruby_system.network.setup_buffers()

        # Set up a proxy port for the system_port. Used for
           load binaries and
        # other functional-only things.
        self.ruby_system.sys_port_proxy = RubyPortProxy()
        board.connect_system_port(self.ruby_system.
           sys_port_proxy.in_ports)
```

5.4.2 Workloads

Gem5 mette a disposizione immagini di sistemi operativi e programmi di benchmark pronti all'uso, chiamati *Risorse*. Queste possono essere utilizzate tramite la chiamata a funzione obtain_resource() in python, all'atto di creazione del sistema, nel quale va specificato il nome di una delle risorse messe già a disposizione da gem5. Essa provvederà a scaricare automaticamente tale risorsa. L'elenco delle risorse disponibili pre-compilate sono disponibili su *https://resources.gem5.org/*

5.4.2.1 Linux Boot

Il seguente codice istanzia una Board ad-hoc per il protocollo Tardis, che fa uso della classe creata in precedenza, ed fa uso di un processore dual-core TimingSimpleCPU con protocollo TARDISTSO.

/gem5/configs/tardis_tso/x86_ubuntu_run.py

```
from m5.util import warn

from gem5.coherence_protocol import CoherenceProtocol
from gem5.components.boards.x86_board import X86Board
from tardis_tso_one_level_cache_hierarchy import
   TARDISTSOOneLevelCacheHierarchy
from gem5.components.memory.single_channel import
   SingleChannelDDR3_1600
from gem5.components.processors.cpu_types import CPUTypes
from gem5.components.processors.simple_processor import
   SimpleProcessor
from gem5.isas import ISA
from gem5.utils.requires import requires
from gem5.resources.resource import obtain_resource
from gem5.simulate.simulator import Simulator
```

/gem5/configs/tardis_tso/x86_ubuntu_run.py

```
class X86TardisBoard(X86Board):
    def __init__(self):
        requires(
            isa_required=ISA.X86,
            coherence_protocol_required =
               CoherenceProtocol.TARDISTSO,
        )

        memory = SingleChannelDDR3_1600(size="3GB")
        processor = SimpleProcessor(cpu_type=CPUTypes.TIMING,
           isa=ISA.X86, num_cores=2)

        cache_hierarchy = TARDISTSOOneLevelCacheHierarchy(size
           = "32kB", assoc = "8", )

        super().__init__(
            clk_freq="2GHz",
            processor=processor,
            memory=memory,
            cache_hierarchy=cache_hierarchy,
        )
board = X86TardisBoard()

board.set_workload(obtain_resource("x86-ubuntu-18.04-boot"))

simulator = Simulator(board=board)
simulator.run()
```

Per avviare il workload, basta eseguire il comando specificando questo file python come parametro:

Run Full System Emulation

```
./build/X86_TARDISTSO/gem5.opt configs/tardis_tso/x86_ubuntu_run
   .py
```

In una seconda shell, per visualizzare l'output del boot di Linux:

Visualize Linux Boot

```
tail -f m5out/board.pc.com_1.device
```

È importante notare che, per eseguire il debugging del protocollo di coerenza in una simulazione Full System, è necessario impostare un TICK tramite l'apposita opzione di debug. Questo valore definisce il momento a partire dal quale la simulazione entrerà in modalità di debugging. Se invece si utilizzasse solo il debug-flag, la modalità di debugging verrebbe attivata fin dall'inizio, introducendo un significativo overhead che renderebbe impraticabile il debugging del protocollo. Ad esempio, supponiamo che X sia il TICK corrispondente all'istante in cui la simulazione va in crash. In questo caso, possiamo impostare debug-start a X * 0.999, in modo che la modalità di debugging inizi poco prima del crash. Questo permette di analizzare esclusivamente i momenti immediatamente precedenti l'errore, facilitando il debugging del protocollo in modo più mirato ed efficiente.

Capitolo 6

Lavori Futuri

Il protocollo così presentato, possiede ancora diverse limitazioni e, pertanto sono possibili molti spazi di manovra per migliorarlo. In particolari, di seguito sono presentati i possibili spunti per lavori futuri:

- Ottimizzazione dei parametri di Lease e Livelock Period: è possibile affinare questi parametri per trovare il giusto equilibrio tra il numero di messaggi di Renew sull'interconnessione e la rapidità di aggiornamento delle variabili di Spinlock.

- Separazione delle cache per istruzioni e dati: attualmente, il protocollo utilizza un unico banco di cache L1 privata per ogni core, in cui vengono memorizzati sia blocchi contenenti istruzioni sia blocchi contenenti dati. Per ridurre l'interferenza tra i due tipi di blocchi, si potrebbe introdurre un secondo banco di cache dedicato, separando la gestione delle Instruction Cache e Data Cache.

- Estensione a un'architettura multi-livello: una delle principali limitazioni dell'attuale protocollo è l'assenza di una Last Level Cache (LLC) condivisa. Attualmente, ogni volta che viene richiesto un blocco, la Directory deve recuperarlo direttamente dalla memoria, con un impatto significativo sulle prestazioni. Un'evoluzione naturale sarebbe l'implementazione del protocollo su due livelli di cache: L1 privata, L2 condivisa (LLC), e Directory, con il LLC che funge da Timestamp Manager. Questo permetterebbe di ridurre drasticamente l'incidenza dei miss lease e dei conseguenti messaggi di Renew, migliorando le prestazioni rispetto ai protocolli basati su Invalidate (come MSI e MESI).

- Espansione dei workload di test: attualmente, i benchmark utilizzati sono limitati. Un possibile sviluppo futuro prevede l'esecuzione di workload più complessi, includendo il supporto a un sistema operativo per valutare il comportamento del protocollo in scenari più realistici.

- Un altro aspetto da considerare riguarda il tipo di blocchi prelevati. I blocchi contenenti solo istruzioni sono, di fatto, di sola lettura, poiché non includono variabili che vengono aggiornate durante l'esecuzione dei workload. Un possibile miglioramento consiste nell'evitare che questi blocchi scadano, poiché non subiscono modifiche e, di conseguenza, i messaggi di Renew risulterebbero superflui. Questo approccio permetterebbe di applicare il meccanismo di

lease esclusivamente ai blocchi contenenti dati effettivi, riducendo così il traffico di controllo e migliorando l'efficienza complessiva del protocollo.

Capitolo 7

References

[1] L. Lamport, “How to make a multiprocessor computer that correctly executes multiprocess programs,” Computers, IEEE Transactions on, vol. 100
[2] gem5, "gem5 bootcamp 2022: CPU models" - https://www.youtube.com/watch?v=cDv-g-c0XCY
[3] Xiangyao Yu and Srinivas Devadas, "Tardis: Time Traveling Coherence Algorithm for Distributed Shared Memory" on IEEE
[4] Xiangyao Yu, Hongzhe Liu, Ethan Zou and Srinivas Devadas, "Tardis 2.0: Optimized Time Traveling Coherence for Relaxed Consistency Models"
[5] Learning gem5, Introduction to Ruby https://www.gem5.org/documentation/learning_gem5/part3/MSIin
[6] Increase speed of emulation in gem5 - https://stackoverflow.com/questions/59860091/how-to-increase-the-simulation-speed-of-a-gem5-run
[7] Building an Ubuntu image in gem5 - https://github.com/gem5/gem5-resources/blob/stable/src/ubuntu-generic-diskimages/BUILDING.md
[8] Resources repository for gem5 - https://github.com/gem5/gem5-resources
[9] Use of gem5’s resources - https://www.gem5.org/documentation/general_docs/gem5_resources/
[10] Gem5’s resources page - https://resources.gem5.org/
[11] X86 Full System Emulation in gem5 - https://www.gem5.org/documentation/gem5-stdlib/x86-full-system-tutorial